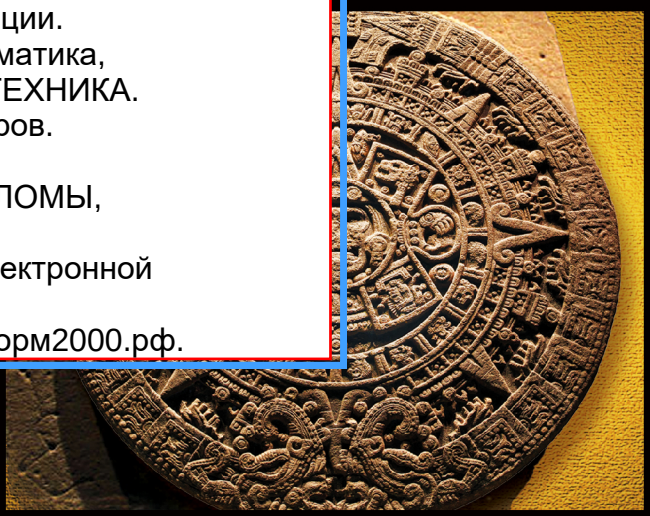
КЛАССИКА COMPUTER SCIENCE

АЛГОРИТМЫ
Разработка и применение



ДЖ. КЛЕЙНБЕРГ Е. ТАРДОС

PEARSON Addison Wesley

ПИТЕР®

JON KLEINBERG, ÉVA TARDOS

ALGORITHM DESIGN

Cornell University

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

КЛАССИКА COMPUTER SCIENCE

ДЖОН КЛЕЙНБЕРГ, ЕВА ТАРДОС

АЛГОРИТМЫ
Разработка и применение

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Киев · Екатеринбург · Самара · Минск

2016

Дж. Клейнберг, Е. Тардос

Алгоритмы: разработка и применение. Классика Computers Science

Серия «Классика computer science»

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Художник	*С. Маликова*
Корректоры	*Н. Викторова, И. Мивриньш*
Верстка	*А. Шляго*

ББК 32.973.2-018
УДК 004.3

Клейнберг Дж., Тардос Е.

К48 Алгоритмы: разработка и применение. Классика Computers Science / Пер. с англ. Е. Матвеева. — СПб.: Питер, 2016. — 800 с.: ил. — (Серия «Классика computer science»).

ISBN 978-5-496-01545-5

Впервые на русском языке выходит одна из самых авторитетных книг по разработке и использованию алгоритмов. Алгоритмы — это основа программирования, определяющая, каким образом программное обеспечение будет использовать структуры данных.

Вы познакомитесь с базовыми аспектами построения алгоритмов, основными понятиями и определениями, структурами данных, затем перейдете к основным методам построения алгоритмов, неразрешимости и методам решения неразрешимых задач, и, наконец, изучите рандомизацию при проектировании алгоритмов.

Самые сложные темы объясняются на четких и простых примерах, поэтому книга может использоваться как для самостоятельного изучения студентами, так и учеными-исследователями или профессионалами в области компьютерных технологий, которые хотят получить представление о применении тех или иных методов проектирования алгоритмов.

Алгоритмический анализ состоит из двух фундаментальных компонентов: выделения математически чистого ядра задачи и выявления методов проектирования подходящего алгоритма на основании структуры задачи. И чем лучше аналитик владеет полным арсеналом возможных методов проектирования, тем быстрее он начинает распознавать «чистые» формулировки, лежащие в основе запутанных задач реального мира.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-0132131087 англ.
ISBN 978-5-496-01545-5

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.066 —
Книги печатные профессиональные, технические и научные.
Подписано в печать 28.01.16. Формат 70×100/16. Бумага писчая. Усл. п. л. 64,500. Тираж 1000. Заказ 0000.
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Оглавление

Предисловие

Алгоритмические идеи вездесущи, а широта их применения наглядно проявляется в примерах как из области информатики, так и за ее пределами. Некоторые серьезные изменения стандартов маршрутизации в Интернете произошли вследствие обсуждения недостатков одного алгоритма кратчайшего пути и относительных преимуществ другого. Базовые понятия, используемые биологами для выражения сходства между генами и геномами, имеют алгоритмические определения. Озабоченность, высказываемая экономистами в контексте практической приемлемости комбинаторных аукционов, отчасти обусловлена тем фактом, что особыми случаями таких аукционов являются вычислительно трудноразрешимые задачи поиска. При этом алгоритмические концепции не ограничиваются хорошо известными, устоявшимися задачами; эти идеи регулярно проявляются в совершенно новых проблемах, возникающих в самых разных областях. Ученый из Yahoo!, однажды рассказавший нам за обедом о системе поставки рекламы пользователям, описывал совокупность задач, которые, по сути, можно было смоделировать как задачу нахождения потока в сети. То же произошло в общении с нашим бывшим студентом (ныне консультантом по вопросам управления, занимающимся правилами подбора персонала для крупных больниц), с которым мы встретились во время поездки в Нью-Йорк.

Дело даже не в том, что алгоритмы находят разнообразные применения. Другое, более глубокое соображение заключается в том, что тема алгоритмов превращается в мощную линзу, через которую можно рассматривать область вычислительных технологий вообще. Алгоритмические задачи образуют ядро компьютерной науки, однако они редко появляются в виде аккуратно упакованных, математически точных вопросов. Чаще они отягощаются множеством хлопотных подробностей, привязанных к конкретному случаю; одни из этих подробностей важны, другие избыточны. В результате алгоритмический анализ состоит из двух фундаментальных компонентов: выделения математически чистого ядра задачи и выявления методов проектирования подходящего алгоритма на основании структуры задачи. Эти два компонента взаимосвязаны: чем лучше аналитик владеет полным арсеналом возможных методов проектирования, тем быстрее он начинает распознавать «чистые» формулировки, лежащие в основе запутанных задач реального мира. Таким образом, при использовании с максимальной эффективностью алгоритмические идеи не просто предоставляют решения четко поставленных задач — они формируют язык для четкого выражения вопросов, заложенных в их основу.

Цель этой книги — донести до читателя этот подход к алгоритмам в форме процесса проектирования, который начинается с задач, встречающихся по всему

диапазону вычислительных приложений, использует хорошее понимание методов проектирования алгоритмов и конечным результатом которого является разработка эффективных решений таких задач. Мы будем изучать роль алгоритмических идей в компьютерной науке вообще и постараемся связать эти идеи с диапазоном точно сформулированных задач, для решения которых проектируются и анализируются алгоритмы. Иначе говоря, какие причины заставляют нас искать решение этих задач и как мы выбираем конкретные способы их формулировки? Как мы узнаем, какие принципы проектирования уместны в той или иной ситуации?

Исходя из сказанного, мы постарались показать, как выявлять «чистые» формулировки алгоритмических задач в сложных вычислительных областях и как на базе этих формулировок проектировать эффективные алгоритмы для решения полученных задач. Чтобы разобраться в сложном алгоритме, часто бывает проще всего реконструировать последовательность идей (включая неудачные заходы и тупики), которые привели от исходных упрощенных методов к выработанному со временем решению. Так сформировался стиль изложения, не приводящий читателя от постановки задачи сразу к алгоритму и, на наш взгляд, лучше отражающий то, как мы и наши коллеги подходим к решению подобных задач.

Общие сведения

Книга предназначена для студентов, прошедших двухсеместровый вводный курс вычислительных технологий (стандартный курс «CS1/CS2»), в ходе которого они писали программы для реализации базовых алгоритмов, и работы с такими структурами данных, как деревья и графы, а также с базовыми структурами данных (массивы, списки, очереди и стеки). Так как переход между этим курсом и первым курсом по алгоритмам еще не стал стандартным, книга открывается с изложения тем, которые знакомы студентам некоторых образовательных учреждений по курсу CS1/CS2, но в других учреждениях включаются в учебный план первого курса по алгоритмам. Соответственно, эта часть может рассматриваться либо как обзор, либо как новый материал; включая ее, мы надеемся, что книга может быть использована в более широком диапазоне учебных курсов и с большей гибкостью в отношении исходных знаний, обязательных для читателя.

В соответствии с описанным подходом мы разрабатываем базовые методы проектирования алгоритмов, изучая задачи из многих областей компьютерной науки и сопутствующих областей. В частности, приводятся довольно подробные описания возможных применений из области систем и сетей (кэширование, коммутация, междоменная маршрутизация в Интернете), искусственного интеллекта (планирование, игры, сети Хопфилда), распознавание образов (сегментация изображений), извлечение информации (обнаружение точки перехода, кластеризация), исследование операций (планирование воздушного движения) и вычислительная биология (выравнивание последовательностей, вторичная структура РНК).

Понятие вычислительной неразрешимости и *NP*-полноты в частности играет значительную роль в книге. Это соответствует нашему подходу к общему процессу

проектирования алгоритма. Иногда интересная задача, возникающая в практической области, имеет эффективное решение, а иногда она оказывается доказуемо *NP*-полной; для полноценного подхода к решению новой алгоритмической задачи вы должны уметь исследовать оба варианта с равной уверенностью. Поскольку очень многие естественные задачи в компьютерной науке являются *NP*-полными, разработка механизмов работы с неразрешимыми задачами стала ключевым фактором изучения алгоритмов, и эта тема широко представлена в книге. Вывод о том, что задача является *NP*-полной, следует воспринимать не как конец исследования, а как приглашение к поиску алгоритмов приближения, методов эвристического локального поиска или разрешимых особых случаев. Каждый из этих трех подходов широко рассматривается в книге.

Чтобы упростить работу с такими задачами, в каждую главу включается раздел «Упражнения с решениями», в котором мы берем одну или несколько задач и подробно описываем процесс поиска решения. По этой причине обсуждение каждого упражнения с решением получается намного длиннее, чем простая запись полного, правильного решения (другими словами, намного длиннее, чем было необходимо для полноценного решения, если бы эта задача была задана студенту для самостоятельного решения). Как и в остальном тексте книги, обсуждения в этих разделах должны дать читателю представление о более масштабных процессах, применяемых для решения задач такого типа и в конечном итоге приводящих к точному решению.

Стоит упомянуть еще два обстоятельства, связанных с использованием этих задач для самостоятельной работы в учебных курсах. Во-первых, задачи упорядочены приблизительно по возрастанию сложности, но это всего лишь приблизительный ориентир, и мы не советуем уделять ему слишком много внимания; так как основная часть задач была разработана для самостоятельной работы студентов, большие подмножества задач в каждой главе довольно тесно связаны в отношении сложности. Во-вторых, если не считать начальных глав, над решением этих задач придется потрудиться — как для того, чтобы связать описание задачи с алгоритмическими методами, описанными в главе, так и для непосредственного проектирования алгоритма. В своем учебном курсе мы назначали около трех таких задач на неделю.

Учебные аспекты и дополнения

Кроме задач и упражнений с решениями, в этой книге используется ряд других учебных аспектов, а также дополнения, упрощающие ее применение в учебных целях.

Как упоминалось ранее, многие разделы в книге посвящены формулировке алгоритмической задачи (включая ее историю и мотивацию для поиска решения), проектированию и анализу алгоритма для ее решения. В соответствии с этим стилем такие разделы имеют единую структуру, включающую последовательность подразделов: «Задача» с описанием задачи и ее точной формулировкой; «Проектирование алгоритма» с применением подходящих методов для разработки

алгоритма; и «Анализ алгоритма» с доказательством свойств алгоритма и анализом его эффективности. В тех случаях, где рассматриваются расширения задачи или приводится дополнительный анализ алгоритма, включаются дополнительные подразделы. Целью такой структуры является введение относительно общего стиля изложения, который переходит от исходного обсуждения проблемы, возникающей в вычислительной области, к подробному анализу методов ее решения.

Наконец, мы будем благодарны читателям за обратную связь. Как и в любой книге такого объема, в ней наверняка встречаются ошибки, которые остались и в печатном варианте. Комментарии и сообщения об ошибках можно присылать по электронной почте *algbook@cs.cornell.edu*; пожалуйста, включите в строку темы слово «feedback».

Краткое содержание

Глава 1 начинается с представления некоторых типичных алгоритмических задач. Мы начнем с задачи устойчивых паросочетаний, потому что она позволяет обозначить базовые аспекты разработки алгоритмов более конкретно и элегантно, чем любые абстрактные рассуждения: поиск устойчивых паросочетаний обусловлен естественной, хотя и сложной практической областью, на базе которой можно абстрагировать интересную формулировку задачи и неожиданно эффективный алгоритм ее решения. В оставшейся части главы 1 рассматриваются пять «типичных задач», которые предопределяют темы из оставшейся части курса. Эти пять задач взаимосвязаны в том смысле, что все они могут рассматриваться как вариации и/или особые случаи задачи поиска независимого множества; но одна задача может быть решена при помощи «жадного» алгоритма, другая — методом динамического программирования, третья — методом нахождения потока в сети, четвертая (собственно задача независимого множества) является *NP*-полной, а пятая — PSPACE-полной. Тот факт, что тесно связанные задачи могут значительно отличаться по сложности, играет важную роль в книге, и эти пять задач служат ориентирами, к которым мы неоднократно возвращаемся по мере изложения материала.

Главы 2 и 3 помогают связать материал с учебным курсом CS1/CS2, о котором говорилось выше. В главе 2 вводятся ключевые математические определения и понятия, используемые при анализе алгоритмов, а также мотивация для их применения. Глава начинается с неформального обзора того, что же следует понимать под вычислительной разрешимостью задачи, а также концепцией полиномиального времени как формального критерия эффективности. Затем вопросы скорости роста функций и асимптотического анализа рассматриваются более формально, приводится информация по функциям, часто встречающимся при анализе алгоритмов, а также по их стандартным применениям. В главе 3 рассматриваются базовые определения и алгоритмические примитивы, необходимые для работы с графами и занимающие центральное место во многих задачах книги. К концу учебного курса CS1/CS2 студенты реализуют многие базовые алгоритмы теории графов, но этот материал полезно представить здесь в более широком контексте

проектирования алгоритмов. В частности, мы рассмотрим базовые определения графов, методы обхода графов (обход в ширину и обход в глубину) и основные концепции направленных графов, включая сильную связность и топологическое упорядочение.

В главах 2 и 3 также представлены многие базовые структуры данных, используемые при реализации алгоритмов в книге; более сложные структуры данных описаны в последующих главах. Наш подход к структурам данных заключается в том, чтобы представлять их тогда, когда они потребуются для реализации алгоритмов, описанных в книге. Таким образом, хотя многие структуры данных уже будут знакомы студентам по курсу CS1/CS2, мы будем рассматривать их в более широком контексте проектирования и анализа алгоритмов.

В главах 4–7 рассматриваются четыре основных метода проектирования алгоритмов: жадные алгоритмы, принцип «разделяй и властвуй», динамическое программирование и нахождение потока в сети. С жадными алгоритмами важно понять, когда они работают, а когда нет; наше рассмотрение этой темы базируется на способе классификации алгоритмов, используемых для доказательства правильности работы жадных алгоритмов. Глава завершается обзором основных применений жадных алгоритмов для поиска кратчайших путей, направленных и ненаправленных связующих деревьев, группировки и сжатия. Обсуждение метода «разделяй и властвуй» начинается с обзора стратегий рекуррентных соотношений как границ времени выполнения; затем мы покажем, как знание этих рекуррентных соотношений может управлять проектированием алгоритмов, превосходящих прямолинейные решения многих базовых задач, включая сравнение оценок, нахождение ближайших пар точек на плоскости и быстрое преобразование Фурье. Знакомство с динамическим программированием начинается с интуитивно понятной рекурсии, на которой оно базируется, с последующей разработкой все более и более выразительных формул через приложения, в которых они естественным образом встречаются. Наконец, мы рассмотрим алгоритмы для задач нахождения потока в сети; большая часть этой главы будет посвящена разнообразным практическим применениям этих потоков. В том объеме, в котором потоки в сети рассматриваются в алгоритмических учебных курсах, студенты часто недостаточно хорошо представляют широкий спектр задач, к которым они могут применяться; мы постараемся воздать должное их гибкости и представим применение потоков для распределения загрузки, планирования, сегментации изображений, а также ряда других задач.

Главы 8 и 9 посвящены понятию вычислительной неразрешимости. Основное внимание в них уделяется *NP*-полноте; базовые *NP*-полные задачи разбиваются на категории, чтобы помочь студентам в идентификации возможных сокращений при обнаружении новых задач. Мы дойдем до достаточно сложных доказательств *NP*-полноты, с рекомендациями относительно того, как следует подходить к построению сложных сокращений. Также будут рассмотрены типы вычислительной сложности, выходящие за рамки *NP*-полноты, особенно в области PSPACE-полноты. Эта полезная концепция подчеркивает, что неразрешимость не кончается на *NP*-полноте; PSPACE-полнота также закладывает фундамент для центральных понятий из об-

ласти искусственного интеллекта (планирования и ведения игр), которые без нее не нашли бы отражения в рассматриваемой алгоритмической области.

В главах с 10-й по 12-ю рассматриваются три основных метода работы с вычислительно неразрешимыми задачами: идентификация структурированных особых случаев, алгоритмы аппроксимации и эвристики локального поиска. Глава, посвященная разрешимым особым случаям, показывает, что экземпляры *NP*-полных задач, встречающиеся на практике, могут быть не столь сложны в худших случаях, потому что нередко они содержат структуру, которая может быть использована при проектировании эффективного алгоритма. Мы покажем, что *NP*-полные задачи часто удается эффективно решить, если ограничиться входными данными, имеющими структуру дерева. Тема завершается подробным обсуждением декомпозиций графов в дерево. Хотя эта тема больше подходит для выпускных учебных курсов, она имеет существенную практическую ценность, для которой трудно найти более доступный материал. В главе, посвященной алгоритмам аппроксимации, рассматривается процесс проектирования эффективных алгоритмов и задача анализа оптимального решения для построения хороших оценок. Что касается методов проектирования алгоритмов аппроксимации, мы сосредоточимся на жадных алгоритмах, линейном программировании и третьем методе, который будет называться «определение цены» (pricing), использующим идеи первых двух. Наконец, мы рассмотрим эвристики локального поиска, включая алгоритм Метрополиса и метод модельной «закалки». Эта тема часто не рассматривается в алгоритмических курсах среднего уровня, потому что о доказуемых гарантиях таких алгоритмов известно очень мало; однако, учитывая их широкое практическое распространение, мы считаем, что студентам будет полезно знать о них. Также включаются описания случаев, для которых существуют доказуемые гарантии.

В главе 13 рассматривается применение рандомизации при проектировании алгоритмов. На эту тему написано несколько хороших книг. Здесь мы постараемся предоставить более компактное введение в некоторые способы применения методов рандомизации, основанные на информации, которая обычно входит в учебные курсы дискретной математики среднего уровня.

Как пользоваться книгой

Книга создавалась прежде всего для вводных учебных курсов по алгоритмам, но она также может использоваться как основа для ознакомительной части основных учебных курсов.

Используя книгу на вводном уровне, мы проводим примерно одну лекцию для каждого нумерованного раздела; если объем материала превышает продолжительность одной лекции (например, если в разделе присутствуют дополнительные примеры), дополнительный материал рассматривается как приложение, с которым студенты могут ознакомиться самостоятельно. Разделы со звездочками пропускаются; хотя в них рассматриваются важные темы, они не столь принципиальны. Также мы обычно пропускаем один-два раздела каждой главы из первой половины

книги (например, часто пропускаются разделы 4.3, 4.7–4.8, 5.5–5.6, 6.5, 7.6 и 7.11). В главах 11–13 рассматривается примерно половина разделов.

Последнее обстоятельство стоит подчеркнуть: вместо того, чтобы рассматривать последние главы как «высокоуровневый материал», недоступный для вводных алгоритмических курсов, мы написали их так, чтобы несколько начальных разделов каждой главы были доступны для аудитории начального уровня. Наши вводные учебные курсы включают материал из всех этих глав, так как мы считаем, что все эти темы занимают достаточно важное место на вводном уровне.

Главы 2 и 3 рассматриваются в основном как обзор материала более ранних курсов; но, как упоминалось выше, использование этих двух глав серьезно зависит от связи каждого конкретного курса с предшествующими курсами.

Итоговый учебный план выглядит примерно так: глава 1; главы 4–8 (исключая разделы 4.3, 4.7–4.9, 5.5–5.6, 6.5, 6.10, 7.4, 7.6, 7.11 и 7.13); глава 9 (кратко); глава 10, разделы 10.1 и 10.2; глава 11, разделы 11.1, 11.2, 11.6 и 11.8; глава 12, разделы 12.1–12.3; глава 13, разделы 13.1–13.5.

Книга также может использоваться в ознакомительной части основных учебных курсов. В нашем представлении такой курс должен познакомить студентов, которым предстоит заниматься исследованиями во многих разных областях, с важными современными вопросами проектирования алгоритмов. В таком случае повышенное внимание формулировке задач тоже приносит пользу, потому что студенты скоро начнут определять собственные исследовательские задачи в различных подобластях. На учебных курсах такого типа мы рассматриваем материал глав 4 и 6 (разделы 4.5–4.9 и 6.5–6.10), весь материал главы 7 (начальные разделы излагаются в более высоком темпе), быстро излагаем тему *NP*-полноты из главы 8 (потому что многие студенты-старшекурсники уже знакомы с этой темой), а все оставшееся время тратится на знакомство с материалом глав 10–13.

Наконец, книга может использоваться для самостоятельного изучения студентами, учеными-исследователями или профессионалами в области компьютерных технологий, которые хотят иметь представление о применении тех или иных методов проектирования алгоритмов в контексте их работы. Многие наши выпускники и коллеги использовали материалы книги таким образом.

Благодарности

Эта книга была написана на основе серии алгоритмических учебных курсов, которые мы вели в Корнелльском университете. За прошедшие годы эти курсы развивались, как развивалась и сама область, и в них отражено влияние преподавателей Корнелла, которые помогали сформировать их в нынешнем виде; это Юрис Хартманис (Juris Hartmanis), Моника Хензингер (Monika Henzinger), Джон Хопкрофт (John Hopcroft), Декстер Козен (Dexter Kozen), Ронитт Рубинфельд (Ronitt Rubinfeld) и Сэм Туэг (Sam Toueg). Кроме того, мы бы хотели поблагодарить всех своих коллег по Корнеллу за бесчисленные обсуждения представленного материала и более широкого круга вопросов, связанных со спецификой области.

Учебный персонал, участвовавший в преподавании курса, оказал огромную помощь в разработке материала. Мы благодарны нашим ассистентам, в числе которых были: Сиддхарт Александр (Siddharth Alexander), Ри Андо (Rie Ando), Эллиот Аншелевич (Elliot Anshelevich), Ларс Бекстром (Lars Backstrom), Стив Бейкер (Steve Baker), Ральф Бензингер (Ralph Benzinger), Джон Бикет (John Bicket), Дуг Бурдик (Doug Burdick), Майк Коннор (Mike Connor), Владимир Дижур (Vladimir Dizhoor), Шаддин Догми (Shaddin Doghmi), Александр Друян (Alexander Druyan), Бовэй Ду (Bowei Du), Саша Евфимиевски (Sasha Evfimievski), Арифул Гани (Ariful Gani), Вадим Гриншпан (Vadim Grinshpun), Ара Хайрапетян (Ara Hayrapetyan), Крис Джуэл (Chris Jeuell), Игорь Кац (Igor Kats), Омар Хан (Omar Khan), Михаил Кобяков (Mikhail Kobyakov), Алексей Копылов (Alexei Kopylov), Брайан Кулис (Brian Kulis), Амит Кумар (Amit Kumar), Ёнви Ли (Yeongwee Lee), Генри Лин (Henry Lin), Ашвин Мачанаваджала (Ashwin Machanavajjhala), Аян Мандал (Ayan Mandal), Бил Макклоски (Bill McCloskey), Леонид Мейергуз (Leonid Meyerguz), Эван Моран (Evan Moran), Нираджан Нагараджан (Niranjan Nagarajan), Тина Нолт (Tina Nolte), Трэвис Ортогеро (Travis Ortogero), Мартин Пал (Martin Pál), Джон Пересс (Jon Peress), Мэтт Пиотровски (Matt Piotrowski), Джо Поластре (Joe Polastre), Майк Прискотт (Mike Priscott), Син Ци (Xin Qi), Вену Рамасубрамьян (Venu Ramasubramanian), Адитья Рао (Aditya Rao), Дэвид Ричардсон (David Richardson), Брайан Сабино (Brian Sabino), Рачит Сиамвалла (Rachit Siamwalla), Себастьян Силгардо (Sebastian Silgardo), Алекс Сливкинс (Alex Slivkins), Чайтанья Свами (Chaitanya Swamy), Перри Там (Perry Tam), Надя Травинин (Nadya Travinin), Сергей Васильвицкий (Sergei Vassilvitskii), Мэтью Уокс (Matthew Wachs), Том Векслер (Tom Wexler), Шан-Люн Мэверик Ву (Shan-Leung Maverick Woo), Джастин Ян (Justin Yang) и Миша Зацман (Misha Zatsman). Многие из них предоставили полезную информацию, поделились замечаниями и предложениями по тексту. Мы также благодарим всех студентов, поделившихся своим мнением по поводу ранних вариантов книги.

За несколько последних лет нам сильно помогали наши коллеги, использовавшие черновые версии материалов для обучения. Анна Карлин (Anna Karlin) бесстрашно взяла предварительную версию за основу своего курса в Вашингтонском университете, когда книга еще находилась на ранней стадии работы; за ней последовали многие люди, использовавшие книгу как учебник или ресурс для обучения: Пол Бим (Paul Beame), Аллан Бородин (Allan Borodin), Девдатт Дубхаши (Devdatt Dubhashi), Дэвид Кемпе (David Kempe), Джин Клейнберг (Gene Kleinberg), Декстер Козен (Dexter Kozen), Амит Кумар (Amit Kumar), Майк Моллой (Mike Molloy), Ювал Рабани (Yuval Rabani), Тим Рафгарден (Tim Roughgarden), Алекса Шарп (Alexa Sharp), Шанхуа Тен (Shanghua Teng), Аравинд Шринивасан (Aravind Srinivasan), Дитер ван Мелькебек (Dieter van Melkebeek), Кевин Уэйн (Kevin Wayne), Том Векслер (Tom Wexler) и Сью Уайтсайдз (Sue Whitesides). Мы глубоко благодарны им за их мнения и советы, которые помогли нам улучшить книгу. Хотим дополнительно поблагодарить Кевина Уэйна за предоставленные материалы, с которыми книга становится еще более полезной для преподавателя.

В ряде случаев в нашем подходе к некоторым темам в книге отразилось влияние конкретных коллег. Наверняка многие из этих предложений прошли мимо нас, но

некоторых людей мы хотим поблагодарить особо: это Юрий Бойков (Yuri Boykov), Рон Элбер (Ron Elber), Дэн Хаттенлокер (Dan Huttenlocher), Бобби Клейнберг (Bobby Kleinberg), Эви Клейнберг (Evie Kleinberg), Лиллиан Ли (Lillian Lee), Дэвид Макаллестер (David McAllester), Марк Ньюман (Mark Newman), Прабхакар Рагхаван (Prabhakar Raghavan), Барт Селман (Bart Selman), Дэвид Шмойс (David Shmoys), Стив Строгац (Steve Strogatz), Ольга Векслер (Olga Veksler), Данкан Уоттс (Duncan Watts) и Рамин Забих (Ramin Zabih).

Нам было приятно работать с Addison Wesley за прошедший год. Прежде всего, спасибо Мэтту Голдстейну (Matt Goldstein) за все его советы и рекомендации и за помощь в преобразовании огромного объема обзорного материала в конкретный план, который улучшил книгу. Наши ранние беседы о книге с Сьюзен Хартман (Susan Hartman) тоже принесли огромную пользу. Мы благодарим Мэтта и Сьюзен, а также Мишель Браун (Michelle Brown), Мэрилин Ллойд (Marilyn Lloyd), Патти Махтани (Patty Mahtani) и Мейта Суарес-Ривас (Maite Suarez-Rivas) из издательства Addison Wesley и Пола Анагностопулоса (Anagnostopoulos) и Жаки Скарлотт (Jacqui Scarlott) из Windfall Software за работу по редактированию, подготовке к выпуску и управлению проектом. Хотим особо поблагодарить Пола и Жаки за мастерскую верстку книги. Спасибо Джойсу Уэллсу (Joyce Wells) за дизайн обложки, Нэнси Мерфи (Nancy Murphy) из Dartmouth Publishing — за работу над иллюстрациями; Теду Локсу (Ted Laux) за составление алфавитного указателя, Каролу Лейбу (Carol Leyba) и Дженнифер Маккейн (Jennifer McClain) за стилевое редактирование и корректуру.

Ансельм Блумер (Anselm Blumer) из Университета Тафта, Ричард Чанг (Richard Chang) из Мэрилендского университета, Кевин Комптон (Kevin Compton) из университета Мичигана, Диана Кук (Diane Cook) из Техасского университета в Арлингтоне, Сариэл Хар-Пелед (Sariel Har-Peled) из Университета Иллинойса в Урбана-Шампейн, Санджив Ханна (Sanjeev Khanna) из Университета Пенсильвании, Филип Клейн (Philip Klein) из Университета Брауна, Дэвид Маттиас (David Matthias) из Университета штата Огайо, Адам Мейерсон (Adam Meyerson) из Калифорнийского университета в Лос-Анджелесе, Майкл Митценмахер (Michael Mitzenmacher) из Гарвардского университета, Стивен Олариу (Stephan Olariu) из Университета Олд-Доминион, Мохан Патури (Mohan Paturi) из Калифорнийского университета в Сан-Диего, Эдгар Рамос (Edgar Ramos) из Университета Иллинойса в Урбана-Шампейн, Санджай Ранка (Sanjay Ranka) из Университета Флориды в Гейнсвилле, Леон Резник (Leon Reznik) из Рочестерского технологического института, Субхаш Сури (Subhash Suri) из Калифорнийского университета в Санта-Барбаре, Дитер ван Мелькебек (Dieter van Melkebeek) из Университета Висконсина в Мэдисоне, Булент Йенер (Bulent Yener) из Ренсслерского политехнического университета не пожалели своего времени и предоставили подробные и содержательные рецензии на рукопись; их комментарии привели к многочисленным усовершенствованиям (как большим, так и малым) в окончательной версии текста.

Наконец, мы хотим поблагодарить наши семьи — Лиллиан, Алису, Дэвида, Ребекку и Эми. Мы ценим их поддержку, терпение и все остальное сильнее, чем можно выразить в тексте.

История этой книги началась в иррациональном восторге конца 90-х, когда область компьютерных технологий, как показалось многим из нас, ненадолго прошла через область, которую традиционно занимали знаменитости и другие обитатели сегмента поп-культуры. (Возможно, это происходило только в нашем воображении). Теперь, через несколько лет после того, как шумиха и цены на акции опустились, мы видим, что в некоторых отношениях этот период навсегда изменил компьютерную науку, и хотя во многом она осталась прежней, энтузиазм, характерный для этой области с самых ранних дней, остается таким же сильным, общее увлечение информационными технологиями еще не остыло, а вычислительные технологии продолжают проникать в новые дисциплины. Итак, мы надеемся, что эта книга будет полезной всем студентам, изучающим эту тему.

Джон Клейнберг
Ева Тардос
Итака, 2005

Глава 1
Введение: некоторые типичные задачи

1.1. Первая задача: устойчивые паросочетания

Начнем с рассмотрения алгоритмической задачи, неплохо демонстрирующей многие темы, которым будет уделяться особое внимание в этой книге. Эта задача обусловлена вполне естественными и практическими соображениями, на основе которых будет сформулирована простая и элегантная постановка задачи. Алгоритм решения задачи также очень элегантен, поэтому большая часть нашей работы будет потрачена на доказательство того, что алгоритм работает правильно, а время, необходимое для его завершения и получения ответа, лежит в допустимых пределах. Происхождение самой задачи, известной как *задача о поиске устойчивых паросочетаний* (далее «задача устойчивых паросочетаний»), имеет несколько источников.

Задача

Задача устойчивых паросочетаний была частично сформулирована в 1962 году, когда два экономиста-математика, Дэвид Гейл и Ллойд Шепли, задались вопросом: можно ли спланировать процесс поступления в колледж (или приема на работу), который был бы *саморегулируемым* (self-enforcing)? Что они имели в виду?

Давайте сначала неформально рассмотрим, какие ситуации могут возникнуть, если группа студентов колледжа начинает подавать заявки в компании на летнюю практику. Ключевым фактором в процессе обработки заявок становится взаимодействие двух разных сторон: компаний (нанимателей) и студентов (кандидатов). Каждый кандидат упорядочивает список компаний в порядке своих предпочтений, а каждая компания — после поступления заявок — формирует свой порядок предпочтений для кандидатов, подавших заявки. На основании этих предпочтений компании обращаются с предложениями к некоторым из своих кандидатов.

Кандидаты решают, какое из полученных предложений стоит принять, и студенты отправляются на свою летнюю практику.

Гейл и Шепли рассмотрели всевозможные сбои, которые могут произойти в этом процессе при отсутствии каких-либо механизмов, обеспечивающих должный ход событий. Допустим, к примеру, что ваш друг Радж только что принял предложение от крупной телекоммуникационной компании CluNet. Через несколько дней начинающая компания WebExodus, которая тянула с принятием нескольких окончательных решений, связывается с Раджем и тоже предлагает ему летнюю практику. Вообще-то, с точки зрения Раджа, вариант с WebExodus предпочтительнее CluNet — скажем, из-за непринужденной атмосферы и творческого азарта. Этот поворот заставляет Раджа отказаться от предложения CluNet и пойти в WebExodus. Лишившись практиканта, CluNet предлагает работу одному из запасных кандидатов, который мгновенно отменяет свое предыдущее согласие на предложение мегакорпорации Babelsoft. Ситуация набирает обороты и постепенно выходит из-под контроля.

С другого направления все выглядит ничуть не лучше, а то и хуже. Допустим, подруге Раджа по имени Челси было назначено отправиться в Babelsoft. Но услышав историю Раджа, она звонит в WebExodus и говорит: «Знаете, я бы предпочла провести это лето в вашей фирме, а не в Babelsoft». Отдел кадров WebExodus охотно верит; более того, заглянув в заявку Челси, они понимают, что она перспективнее другого студента, у которого уже запланирована летняя практика. И если компания WebExodus не отличается особой деловой принципиальностью, она найдет способ отозвать свое предложение другому студенту и возьмет Челси на его место.

Такие ситуации быстро порождают хаос, и многие участники — как кандидаты, так и наниматели — могут оказаться недовольны как самим процессом, так и его результатом. Что же пошло не так? Одна из основных проблем заключается в том, что процесс не является саморегулируемым; если участникам разрешено произвольно действовать, исходя из их собственных интересов, весь процесс может быть нарушен.

Пожалуй, вы бы предпочли следующую, более устойчивую ситуацию, в которой сами личные интересы участников предотвращают отмену и перенаправление предложений. Допустим, другой студент, который должен был провести лето в CluNet, звонит в WebExodus и сообщает, что он бы предпочел поработать на них. Но на основании уже принятых предложений ему отвечают: «Нет, каждый из принятых нами кандидатов предпочтительнее вас, поэтому мы ничего не сможем сделать». Или возьмем нанимателя, который преследует лучших кандидатов, распределенных по другим фирмам, но получает от каждого из них ответ: «Спасибо, но мне и здесь хорошо». В таком случае все результаты устойчивы — никакие дальнейшие «перетасовки» невозможны.

Итак, вот как выглядел вопрос, заданный Гейлом и Шепли: можно ли для имеющегося набора предпочтений по кандидатам и нанимателям распределить кандидатов по нанимателям так, чтобы для каждого нанимателя E и каждого кандидата A, который не был принят на работу к E, выполнялось по крайней мере одно из следующих двух условий?

1. Каждый из кандидатов, принятых E на работу, с его точки зрения, предпочтительнее A; или
2. С точки зрения A, его текущая ситуация предпочтительнее работы на нанимателя E.

Если этот критерий выполняется, то результат устойчив (стабилен): личные интересы сторон предотвратят закулисные дополнительные договоренности между кандидатами и нанимателями.

Гейл и Шепли разработали блестящее алгоритмическое решение для задачи, которым мы сейчас займемся. Но сначала стоит заметить, что это не единственный источник происхождения задачи устойчивых паросочетаний. Оказывается, еще за десять лет до работы Гейла и Шепли очень похожая процедура, основанная на тех же соображениях, использовалась Национальной программой распределения студентов-медиков по больницам. Более того, эта система с относительно незначительными изменениями продолжает применяться и в наши дни.

Формулировка задачи

Чтобы добраться до сути концепции, задачу следует по возможности очистить от всего лишнего. В мире компаний и кандидатов существует асимметрия, которая только отвлекает от главного. Каждый кандидат подбирает одну компанию, но каждая компания подбирает нескольких кандидатов; более того, количество кандидатов может оказаться больше (или, как это бывает, меньше) количества доступных вакансий. Наконец, в реальной жизни каждый кандидат редко подает заявки во все имеющиеся компании.

По крайней мере на начальной стадии будет полезно устранить эти сложности и перейти к «упрощенной» постановке задачи: каждый из n кандидатов подает заявки в каждую из n компаний, а каждая компания хочет принять на работу *одного* кандидата. Вскоре вы увидите, что при этом сохраняются фундаментальные аспекты задачи; в частности, наше решение упрощенной версии можно будет напрямую расширить для более общего случая.

Вслед за Гейлом и Шепли мы заметим, что этот частный случай можно рассматривать как задачу разработки системы, в которой n мужчин и n женщин могут подыскать себе пару. В нашей задаче существует естественная аналогия для двух «полов» (кандидаты и компании), а в рассматриваемом случае каждый участник подыскивает себе ровно одного участника противоположного пола[1].

Итак, имеется множество $M = \{m_1, ..., m_n\}$ из n мужчин и множество $W = \{w_1, ..., w_n\}$ из n женщин. Пусть запись $M \times W$ обозначает множество всех возможных

[1] Гейл и Шепли также рассматривали задачу однополых устойчивых паросочетаний. У нее тоже имеются свои практические применения, но на техническом уровне возникают достаточно заметные отличия. С учетом схемы «кандидат–наниматель», которая здесь рассматривается, мы будем придерживаться версии с двумя полами.

упорядоченных пар в форме (m, w), где $m \in M$ и $w \in W$. *Паросочетание*[1] S представляет собой множество упорядоченных пар, каждая из которых входит в $M \times W$, обладающее тем свойством, что каждый элемент M и каждый элемент W встречается не более чем в одной паре в S. Идеальным паросочетанием S' называется паросочетание, при котором каждый элемент M и каждый элемент W встречается ровно в одной паре из S'.

Мы еще не раз в книге вернемся к концепциям паросочетаний и идеальных паросочетаний; они естественным образом возникают при моделировании широкого диапазона алгоритмических задач. На текущий момент термин «идеальное паросочетание» соответствует способу формирования пар из мужчин и женщин таким способом, чтобы каждый оказался с кем-то в паре и никто не был включен более чем в одну пару — ни одиночество, ни полигамия.

Теперь в эту схему можно добавить понятие *предпочтений*. Каждый мужчина $m \in M$ формирует оценки всех женщин; мы говорим, что m предпочитает w женщине w', если m присваивает w более высокую оценку, чем w'. Мы будем называть упорядоченную систему оценок m его *списком предпочтений*. «Ничьи» в оценках запрещены. Аналогичным образом каждая женщина назначает оценки всем мужчинам.

Какие могут возникнуть проблемы с имеющимся идеальным паросочетанием S? В контексте исходной задачи с нанимателями и кандидатами возможна следующая неприятная ситуация: в S присутствуют две пары (m, w) и (m', w') (рис. 1.1), обладающие таким свойством, что m предпочитает w' женщине w, а w' предпочитает m мужчине m'. В таком случае ничто не помешает m и w' бросить своих партнеров и создать новую пару; такой набор пар не является саморегулируемым. В нашей терминологии такая пара (m, w') является *неустойчивой* по отношению к S: пара (m, w') не принадлежит S, но каждый из участников m и w' предпочитает другого своему текущему партнеру в S.

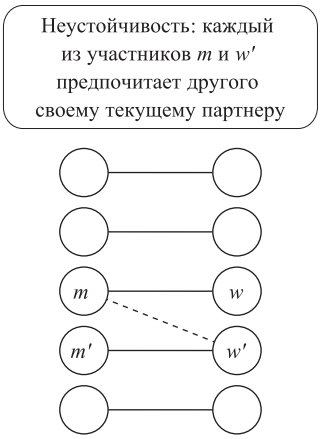


Рис. 1.1. Идеальное паросочетание S с неустойчивостью (m, w')

[1] Также встречается термин «марьяж». — *Примеч. пер.*

Итак, нашей целью является создание паросочетания без неустойчивых пар. Паросочетание S называется устойчивым, если оно (1) идеально и (2) не содержит неустойчивости в отношении S. Немедленно возникают два вопроса:

- Существует ли устойчивое паросочетание для каждого набора списков предпочтений?
- Можно ли эффективно построить устойчивое паросочетание для имеющегося списка предпочтений (если оно существует)?

Примеры

Для демонстрации этих определений будут рассмотрены два очень простых примера задачи устойчивых паросочетаний.

Для начала допустим, что имеется множество из двух мужчин $\{m, m'\}$ и множество из двух женщин $\{w, w'\}$. Списки предпочтений выглядят так:

- m предпочитает w женщине w';
- m' предпочитает w женщине w';
- w предпочитает m мужчине m';
- w' предпочитает m мужчине m'.

Если рассматривать этот набор списков предпочтений на интуитивном уровне, он представляет ситуацию полного согласия: мужчины сходятся во мнениях относительно порядка женщин, а женщины — относительно порядка мужчин. В такой ситуации существует уникальное устойчивое паросочетание, состоящее из пар (m, w) и (m', w'). Другое идеальное паросочетание, состоящее из пар (m', w) и (m, w'), не является устойчивым, потому что пара (m, w) образует неустойчивость по отношению к этому паросочетанию. (Как m, так и w предпочли бы бросить своих партнеров и образовать новую пару.)

В следующем примере ситуация немного усложняется. Предположим, действуют следующие предпочтения:

- m предпочитает w женщине w';
- m' предпочитает w' женщине w;
- w предпочитает m' мужчине m;
- w' предпочитает m мужчине m'.

Что же происходит на этот раз? Предпочтения двух мужчин противоположны (они ставят на первое место разных женщин); то же самое можно сказать и о предпочтениях двух женщин. Однако предпочтения мужчин полностью противоречат предпочтениям женщин.

Во втором примере существуют два разных устойчивых паросочетания. Сочетание, состоящее из пар (m, w) и (m', w'), является устойчивым — оба мужчины довольны, насколько это возможно, и ни один из них не покинет своего партнера. Однако паросочетание из пар (m', w) и (m, w') тоже устойчиво, так как обе женщины довольны, насколько это возможно. Запомните этот важный момент, прежде чем двигаться дальше, — в заданной ситуации может существовать более одного устойчивого паросочетания.

Проектирование алгоритма

Теперь мы продемонстрируем, что для каждого набора списков предпочтений среди мужчин и женщин существует устойчивое паросочетание. Более того, способ демонстрации заодно ответит на второй вопрос, который был задан выше: мы сформулируем эффективный алгоритм, который получает списки предпочтений и строит по ним устойчивое паросочетание.

Рассмотрим некоторые базовые идеи, заложенные в основу алгоритма.

- Изначально ни один из участников не имеет пары. Допустим, неженатый мужчина m выбирает женщину w с наивысшей оценкой в его списке предпочтений и делает ей предложение. Можно ли немедленно объявить, что пара (m, w) войдет в итоговое устойчивое паросочетание? Не обязательно; в какой-то момент в будущем мужчина m', более предпочтительный с точки зрения женщины w, может сделать ей предложение. С другой стороны, для w будет рискованно немедленно отказывать m; она может не получить ни одного предложения от участника с более высокой оценкой в ее списке предпочтений. Таким образом, возникает естественная идея перевести пару (m, w) в промежуточное состояние *помолвки*.
- Допустим, в текущем состоянии некоторые мужчины и женщины *свободны* (то есть не помолвлены), а другие участвуют в помолвке. Следующий шаг может выглядеть примерно так: произвольный свободный мужчина m выбирает женщину с наивысшей оценкой w, которой он еще не делал предложение, и обращается к ней с предложением. Если женщина w тоже свободна, то m и w вступают в состояние помолвки. В противном случае w уже помолвлена с другим мужчиной m'; тогда она определяет, кто из m и m' имеет более высокую оценку в ее списке предпочтений; этот мужчина вступает в помолвку с w, а другой становится свободным.
- Наконец, алгоритм в какой-то момент должен определить, что ни одного свободного участника не осталось; тогда все помолвки объявляются окончательными и возвращается полученное идеальное паросочетание.

Ниже приводится конкретное описание алгоритма Гейла–Шепли; его промежуточное состояние изображено на рис. 1.2.

```
В начальном состоянии все m ∈ M и w ∈ W свободны.
Пока существует свободный мужчина m, который еще не успел сделать предложение
каждой женщине
    Выбрать такого мужчину m.
    Выбрать w – женщину с наивысшей оценкой в списке предпочтений m,
        которой m еще не делал предложения.
    Если w свободна, то
        пара (m, w) вступает в состояние помолвки.
    Иначе w в настоящее время помолвлена с m'.
        Если w предпочитает m' мужчине m, то
            m остается свободным.
```

```
        Иначе w предпочитает m мужчине m'
            пара (m, w) вступает в состояние помолвки.
            m' становится свободным.
        Конец Если
    Конец Если
Конец Пока
Вернуть множество S помолвленных пар
```

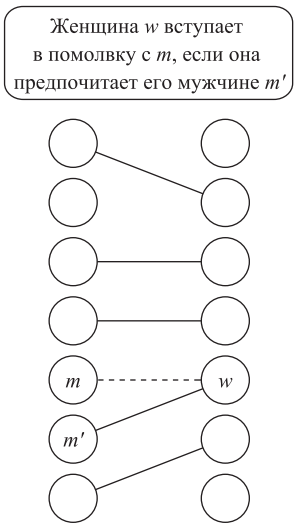


Рис. 1.2. Промежуточное состояние алгоритма Гейла–Шепли: свободный мужчина m делает предложение женщине w

Хотя формулировка алгоритма Гейла–Шепли выглядит достаточно просто, на первый взгляд не очевидно, что он возвращает устойчивое паросочетание — или хотя бы идеальное паросочетание. Сейчас мы докажем устойчивость результата через серию промежуточных фактов.

Анализ алгоритма

Для начала рассмотрим ситуацию с точки зрения женщины w во время выполнения алгоритма. Какое-то время она не получает предложений и остается свободной. Потом мужчина m может сделать ей предложение, и она становится помолвленной. Со временем она может получить дополнительные предложения и принять те из них, в которых повышается оценка ее партнера. Итак, мы приходим к следующим выводам:

(1.1) *Женщина w остается помолвленной с момента получения первого предложения*; *партнеры*, *с которыми она вступает в помолвку*, *становятся все лучше и лучше* (*в контексте ее списка предпочтений*).

С точки зрения мужчины m, во время выполнения алгоритма все происходит иначе. Он остается свободным до тех пор, пока не сделает предложение женщине

с наивысшей оценкой в его списке; в этот момент он может вступить в помолвку — а может и не вступить. С течением времени он может переходить из свободного состояния в состояние помолвки и обратно; однако при этом продолжает действовать следующее утверждение.

(1.2) Последовательность женщин, которым m делает предложение, постоянно ухудшается (в контексте его списка предпочтений).

Теперь мы покажем, что алгоритм завершается, и оценим максимальное количество итераций, необходимых для завершения.

(1.3) Алгоритм Гейла–Шепли завершается после выполнения максимум n^2 итераций цикла «Пока».

Доказательство. Полезная стратегия определения верхней границы времени выполнения основана на поиске метрики продвижения, а именно: следует найти какую-то точную формулировку того, насколько каждый шаг, выполняемый алгоритмом, приближает его к завершению.

В рассматриваемом алгоритме каждая итерация состоит из предложения, которое некий мужчина делает (не более одного раза) женщине, к которой он еще не обращался ранее. Итак, если обозначить $P(t)$ множество пар (m, w), в которых мужчина m делал предложение w к моменту завершения итерации t, мы видим, что для всех t размер $P(t + 1)$ строго больше размера $P(t)$. Однако всего существует только n^2 возможных пар, так что за время выполнения алгоритма значение $P(\cdot)$ может возрасти не более n^2 раз. Следовательно, выполнение алгоритма может потребовать не более n^2 итераций. ■

В приведенном утверждении и его доказательстве заслуживают внимания два обстоятельства. Во-первых, некоторые выполнения алгоритма (с некоторыми списками предпочтений) могут потребовать около n^2 итераций, так что анализ достаточно близок к лучшей возможной оценке. Во-вторых, существует много других метрик, которые не так хорошо подходят для оценки продвижения алгоритма, поскольку они не обязательно строго возрастают при каждой итерации. Например, количество свободных участников (а также количество помолвленных пар) может оставаться постоянным между итерациями. Следовательно, эти величины не могут использоваться непосредственно для определения верхней границы максимально возможного количества итераций, как в предыдущем абзаце.

Теперь докажем, что множество S, возвращаемое при завершении алгоритма, действительно является идеальным паросочетанием. Почему это не очевидно? По сути, нужно показать, что ни один мужчина не остановится в конце своего списка предпочтений; выход из цикла возможен только в том случае, если ни одного свободного мужчины не осталось. В этом случае набор помолвленных пар действительно будет идеальным паросочетанием.

Итак, необходимо доказать следующее:

(1.4) Если мужчина m свободен в какой-то точке выполнения алгоритма, то существует женщина, которой он еще не делал предложение.

Доказательство. Предположим, существует точка, в которой мужчина m свободен, но уже сделал предложение всем женщинам. Тогда из (1.1) следует, что каждая

из n женщин на этот момент уже помолвлена. Так как набор помолвленных пар образует паросочетание, при этом также должно быть n помолвленных мужчин. Но мужчин всего n, а согласно исходному предположению, m не помолвлен; возникает противоречие. ■

(1.5) Множество S, возвращаемое при завершении, является идеальным паросочетанием.

Доказательство. Множество помолвленных пар всегда образует паросочетание. Предположим, что алгоритм завершается со свободным мужчиной m. При завершении m должен был уже сделать предложение всем женщинам, в противном случае цикл бы не завершился. Но это противоречит положению (1.4), согласно которому не может быть свободного мужчины, который уже сделал предложение всем женщинам. ■

Наконец, мы доказали главное свойство алгоритма, а именно то, что он приводит к устойчивому паросочетанию.

(1.6) Возьмем выполнение алгоритма Гейла–Шепли, возвращающее множество пар S. Это множество S является устойчивым паросочетанием.

Доказательство. Из (1.5) мы уже знаем, что S является идеальным паросочетанием. Таким образом, чтобы доказать, что S является устойчивым паросочетанием, мы предположим, что существует неустойчивость по отношению к S, и придем к противоречию. Согласно приведенному ранее определению, при такой неустойчивости S будет содержать две пары (m, w) и (m', w'), в которых:

- m предпочитает w' женщине w;
- w' предпочитает m мужчине m'.

В процессе выполнения алгоритма, приведшего к S, последнее предложение m по определению было обращено к w. Вопрос: делал ли m предложение w' в некоторой предшествующей точке выполнения? Если не делал, то женщина w должна находиться в списке предпочтений m выше, чем w', что противоречит нашему допущению о том, что m предпочитает w' женщине w. Если делал, то w' отказала ему ради другого мужчины m'', которого w' предпочитает m. Однако m' является итоговым партнером w', а это значит, что либо $m'' = m'$, либо, согласно (1.1), w' предпочитает своего итогового партнера m' мужчине m''; в любом случае это противоречит предположению о том, что w' предпочитает m мужчине m'.

Следовательно, S является устойчивым паросочетанием. ■

Расширения

Мы начали с определения устойчивого паросочетания и только что доказали, что алгоритм Гейла–Шепли действительно строит его. Теперь будут рассмотрены некоторые дополнительные вопросы относительно поведения алгоритма Гейла–Шепли и его связи со свойствами различных устойчивых паросочетаний.

Прежде всего вернемся к тому, что было продемонстрировано в одном из предыдущих примеров с несколькими устойчивыми паросочетаниями. Вспомните, что списки предпочтений в этом примере выглядели так:

- m предпочитает w женщине w';
- m' предпочитает w' женщине w;
- w предпочитает m' мужчине m;
- w' предпочитает m мужчине m'.

Теперь при любом выполнении алгоритма Гейла–Шепли m будет помолвлен с w, m' будет помолвлен с w' (возможно, в обратном порядке), и на этом все остановится. Следовательно, другое устойчивое паросочетание, состоящее из пар (m', w) и (m, w'), недостижимо при выполнении алгоритма Гейла–Шепли, в котором мужчины делают предложение. С другой стороны, оно станет достижимым при выполнении версии алгоритма, в которой предложение делают женщины. А в более крупных примерах, в которых с каждой стороны задействовано более двух участников, может существовать еще большее количество возможных устойчивых паросочетаний, многие из которых не могут быть сгенерированы никаким естественным алгоритмом.

Этот пример демонстрирует некоторую «несправедливость» алгоритма Гейла–Шепли, «подыгрывающего» мужчинам. Если предпочтения мужчин идеально распределены (то есть все они ставят на первое место разных женщин), то при любом выполнении алгоритма Гейла–Шепли каждый мужчина окажется в паре со своей главной кандидатурой независимо от предпочтений женщин. Если же предпочтения женщин полностью расходятся с предпочтениями мужчин (как в приведенном примере), то полученное устойчивое паросочетание будет насколько возможно плохим для женщин. Итак, этот простой набор списков предпочтений компактно описывает мир, в котором кому-то суждено оказаться несчастным: женщины несчастны, если предложения делают мужчины, а мужчины несчастны, если предложения делают женщины.

Давайте проанализируем алгоритм Гейла–Шепли более подробно и попробуем понять, насколько общий характер имеет эта «несправедливость».

Прежде всего, наш пример подчеркивает то обстоятельство, что формулировка алгоритма Гейла–Шепли не является полностью детерминированной: каждый раз, когда имеется свободный мужчина, можно выбрать *любого* свободного мужчину, который сделает следующее предложение. Различные варианты выбора приводят к разным путям выполнения алгоритма; вот почему в формулировке (1.6) используется осторожное «Возьмем выполнение алгоритма Гейла–Шепли, возвращающее множество пар S», а не «Возьмем множество пар S, возвращаемое алгоритмом Гейла–Шепли».

Возникает другой, тоже очень естественный вопрос: приводят ли все выполнения алгоритма Гейла–Шепли к одному паросочетанию? Вопросы такого рода возникают во многих областях теории обработки данных: имеется алгоритм, который работает асинхронно, с множеством независимых компонентов, которые выполняют операции с возможностью сложных взаимодействий. Требуется определить, какую изменчивость вносит такая асинхронность в окончательный результат.

Или совершенно другой пример: независимыми компонентами могут быть не мужчины и женщины, а электронные устройства, активизирующие части крыла самолета; последствия асинхронности в их поведении могут быть очень серьезными.

В текущем контексте ответ на этот вопрос оказывается на удивление простым: все выполнения алгоритма Гейла–Шепли приводят к одному паросочетанию. Докажем это утверждение.

Все выполнения приводят к одному паросочетанию

Существует много разных способов доказательства подобных утверждений, нередко требующих довольно сложной аргументации. Как выясняется, самый простой и наиболее содержательный метод заключается в том, чтобы уникальным образом *охарактеризовать* полученное паросочетание, а потом показать, что все выполнения приводят к паросочетанию, обладающему такой характеристикой.

Что же это за характеристика? Мы покажем, что каждый мужчина в конечном итоге оказывается «лучшим из возможных партнеров» в конкретном смысле. (Вспомните, что это утверждение истинно, если все мужчины предпочитают разных женщин.) Для начала мы будем считать, что женщина w является *действительным* партнером мужчины m, если существует устойчивое паросочетание, содержащее пару (m, w). Женщина w будет называться *лучшим действительным партнером* m, если w является действительным партнером m и никакая женщина, которой m назначает более высокую оценку, чем w, не является его действительным партнером. Для обозначения лучшего действительного партнера m будет использоваться запись $best(m)$.

Пусть S^* обозначает множество пар $\{(m, best(m)) : m \in M\}$. Докажем следующий факт:

(1.7) Каждое выполнение алгоритма Гейла–Шепли приводит к множеству S^*.

Это утверждение выглядит удивительно сразу на нескольких уровнях. Прежде всего, в его текущей формулировке не существует причины полагать, что S^* вообще является паросочетанием, не говоря уже об устойчивости. В конце концов, почему у двух мужчин не может быть единого лучшего действительного партнера? Во-вторых, оно фактически заявляет, что алгоритм Гейла–Шепли обеспечивает лучший возможный результат для каждого мужчины одновременно; не существует устойчивого паросочетания, в котором кто-нибудь из мужчин мог бы рассчитывать на более высокий результат. И наконец, оно отвечает на приведенный выше вопрос, показывая, что порядок предложений в алгоритме Гейла–Шепли абсолютно не влияет на конечный результат.

Но, несмотря на все, доказать его не так уж сложно.

Доказательство. Пойдем от обратного: предположим, что некоторое выполнение $\mathcal{E}$ алгоритма Гейла–Шепли порождает паросочетание S, в котором некий мужчина находится в одной паре с женщиной, не являющейся его лучшим действительным партнером. Так как мужчины делают предложение в порядке убывания предпочтений, это означает, что он был отвергнут действительным партнером в ходе выполнения $\mathcal{E}$ алгоритма. Возьмем первый момент в ходе выполнения $\mathcal{E}$, когда некоторый мужчина (скажем, m) отвергается действительным партнером w.

Поскольку мужчины делают предложение по убыванию предпочтений, а предложение отклоняется впервые, неизбежно следует, что женщина w является лучшим действительным партнером m, то есть $best(m)$.

Отказ w мог произойти либо из-за того, что предложение m было отклонено в пользу существующей помолвки w, либо потому, что женщина w разорвала помолвку с m в пользу улучшенного предложения. Но в любом случае в этот момент w образует или продолжает помолвку с мужчиной m', которого она предпочитает m.

Поскольку w является действительным партнером m, должно существовать устойчивое паросочетание S', содержащее пару (m, w). Теперь зададимся вопросом: кто является партнером m' в этом паросочетании? Допустим, это женщина $w' \neq w$.

Так как отказ w от предложения m был первым отказом, полученным мужчиной от действительного партнера при выполнении $\mathcal{E}$, из этого следует, что m' не был отвергнут никаким действительным партнером на момент выполнения $\mathcal{E}$, в котором он был помолвлен с w. Поскольку предложения делаются в порядке убывания предпочтений, а w' очевидно является действительным партнером m', из этого следует, что m' предпочитает w женщине w'. Но нам уже известно, что w предпочитает m' мужчине m, поскольку в ходе выполнения E она отвергла m в пользу m'. Так как $(m', w) \notin S'$, можно сделать вывод, что пара (m', w) является неустойчивой по отношению к S'.

Но это противоречит требованию об устойчивости S', а следовательно, исходное предположение было неверным. ■

Итак, для мужчин алгоритм Гейла–Шепли идеален. К сожалению, о женщинах того же сказать нельзя. Для женщины w мужчина m является действительным партнером, если существует устойчивое паросочетание, содержащее пару (m, w). Мужчина m будет называться *худшим действительным партнером* для w, если m является действительным партнером w и ни один мужчина с оценкой ниже, чем m, в списке предпочтений w не является ее действительным партнером.

(1.8) В устойчивом паросочетании S^* каждая женщина оказывается в паре со своим худшим действительным партнером.

Доказательство. Предположим, в S^* существует пара (m, w), в которой m не является худшим действительным партнером w. Тогда должно существовать устойчивое паросочетание S', в котором w находится в паре с мужчиной m', имеющим более низкую оценку, чем m. В S' мужчина m находится в паре с женщиной $w' \neq w$; поскольку w является лучшим действительным партнером m и w' является действительным партнером m, мы видим, что m предпочитает w женщине w'.

Но из сказанного следует, что пара (m, w) является неустойчивой по отношению к S', а это противоречит требованию об устойчивости S', а следовательно, исходное предположение было неверным. ■

Итак, рассмотренный пример, в котором предпочтения мужчин вступали в противоречие с предпочтениями женщин, дает представление о чрезвы-

чайно общем явлении: для любых входных данных сторона, которая делает предложение в алгоритме Гейла–Шепли, оказывается в паре с лучшим из возможных устойчивых партнеров (с ее точки зрения), тогда как сторона, которая не делает предложение, соответственно оказывается с худшим из возможных устойчивых партнеров.

1.2. Пять типичных задач

Задача устойчивых паросочетаний служит превосходным примером процесса проектирования алгоритма. Для многих задач этот процесс состоит из нескольких ключевых шагов: постановка задачи с математической точностью, достаточной для того, чтобы сформулировать конкретный вопрос и начать продумывать алгоритмы ее решения; планирование алгоритма для задачи; и анализ алгоритма, который доказывает его точность и позволяет получить граничное время выполнения для оценки эффективности.

Для практической реализации этой высокоуровневой стратегии применяются фундаментальные приемы проектирования, чрезвычайно полезные для оценки присущей ей сложности и формулирования алгоритма для ее решения. Как и в любой другой области, на освоение этих приемов проектирования потребуется некоторое время; но при наличии достаточного опыта вы начнете распознавать задачи как принадлежащие к уже известным областям и ощутите, как незначительные изменения в формулировке задачи могут иметь колоссальные последствия для ее вычислительной сложности.

Освоение этой темы будет полезно начать с некоторых типичных ключевых точек, которые встретились нам в процессе нашего изучения алгоритмов: четко сформулированных задач, похожих друг на друга на общем уровне, но значительно различающихся по сложности и методам, которые будут задействованы для их решения. Первые три задачи могут быть эффективно решены последовательностью алгоритмических приемов нарастающей сложности; четвертая задача станет поворотной точкой в нашем обсуждении — в настоящее время считается, что эффективного алгоритма для ее решения не существует. Наконец, пятая задача дает представление о классе задач, которые считаются еще более сложными.

Все эти задачи самодостаточны, а побудительные причины для их изучения лежат в области вычислительных приложений. Тем не менее для описания некоторых из них будет удобно воспользоваться терминологией *графов*. Графы — достаточно стандартная тема на ранней стадии изучения вычислительных технологий, но в главе 3 они будут рассматриваться на довольно глубоком уровне; кроме того, ввиду огромной выразительной силы графов они будут широко использоваться в книге. В контексте текущего обсуждения граф G достаточно рассматривать просто как способ представления попарных отношений в группе объектов. Таким образом, G состоит из пары множеств (V, E) — набора *узлов* V и набора *ребер* E, каждое из которых связывает два узла. Таким образом, ребро $e \in E$ представляет двухэлементное подмножество V: $e = \{u, v\}$ для некоторых $u, v \in V$; при этом u и v называются концами e.

Графы обычно изображаются так, как показано на рис. 1.3; узлы изображаются маленькими кружками, а ребра — отрезками, соединяющими два конца.

Перейдем к обсуждению пяти типичных задач.

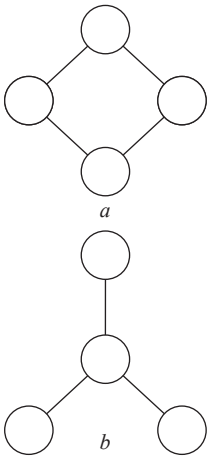


Рис. 1.3. Примеры графов с четырьмя узлами

Интервальное планирование

Рассмотрим очень простую задачу планирования. Имеется некий ресурс — аудитория, суперкомпьютер, электронный микроскоп и т. д.; множество людей обращается с заявками на использование ресурса в течение периода времени. *Заявка* имеет вид: «Можно ли зарезервировать ресурс, начиная с времени s и до времени f?» Будем считать, что ресурс в любой момент времени может использоваться только одним человеком. Планировщик выбирает подмножество заявок, отклоняя все остальные, чтобы принятые заявки не перекрывались по времени. Целью планирования является максимизация количества принятых заявок.

Более формальная постановка: имеется n заявок с номерами 1, ..., n; в каждой заявке i указывается начальное время s_i и конечное время f_i. Естественно, $s_i < f_i$ для всех i. Две заявки i и j называются *совместимыми*, если запрашиваемые интервалы не перекрываются, то есть заявка i приходится либо на более ранний интервал времени, чем у j ($f_i \le s_j$), либо на более поздний интервал времени, чем у j ($f_j \le s_i$). В более общем смысле подмножество A заявок называется совместимым, если все пары запросов $i, j \in A$, $i \ne j$ являются совместимыми. Целью алгоритма является выбор совместимого подмножества заявок максимально возможного размера.

Пример задачи интервального планирования представлен на рис. 1.4. На диаграмме представлено одно совместимое множество размера 4, и оно является самым большим совместимым множеством.

Вскоре мы увидим, что эта задача может быть решена с применением очень естественного алгоритма, который упорядочивает множество заявок по некоторой эвристике, а затем «жадно» обрабатывает их за один проход, выбирая совмести-

мое подмножество максимально возможного размера. Это типично для категории «жадных» (greedy) алгоритмов, которые мы будем рассматривать для различных задач, — «недальновидных» правил, которые обрабатывают входные данные по одному элементу без сколько-нибудь очевидных опережающих проверок. Когда «жадный» алгоритм находит оптимальное решение для любых конфигураций задачи, это часто выглядит совершенно неожиданно. Впрочем, сам факт оптимальности столь простого решения обычно что-то говорит о структуре нижележащей задачи.

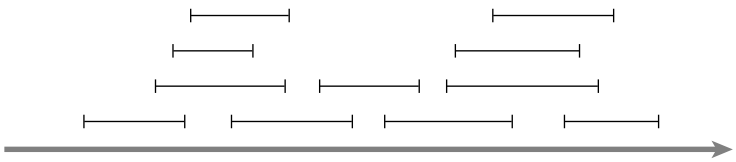


Рис. 1.4. Пример задачи интервального планирования

Взвешенное интервальное планирование

В задаче интервального планирования мы стремились максимизировать количество одновременно принимаемых заявок. Теперь рассмотрим более общую постановку задачи: интервалу каждой заявки i присваивается соответствующий коэффициент (*вес*) $v_i > 0$; можно рассматривать его как оплату, которую владелец ресурса получит от i-го претендента при удовлетворении его заявки. Наша цель — найти совместимое подмножество интервалов с максимальным общим весом.

Частный случай, при котором $v_i = 1$ для каждого i, представляет собой базовую задачу интервального планирования; однако введение произвольных весов существенно влияет на природу задачи максимизации. Например, если значение v_1 превышает сумму всех остальных v_i, то оптимальное решение должно включать интервал 1 независимо от конфигурации полного набора интервалов. Итак, любой алгоритм для решения этой задачи должен быть крайне чувствителен к значениям весов, но при этом вырождаться в метод решения (невзвешенной) задачи интервального планирования, если все значения равны 1.

Вероятно, не существует простого «жадного» правила, которое однократно перебирает все интервалы, принимая правильное решение для произвольных значений. Вместо этого будет применен метод *динамического программирования*, который строит оптимальное значение по всем возможным решениям в компактной табличной форме, обеспечивающей высокую эффективность алгоритма.

Двудольные паросочетания

При рассмотрении задачи устойчивых паросочетаний мы определили паросочетание как множество упорядоченных пар мужчин и женщин, обладающее тем свойством, что каждый мужчина и каждая женщина принадлежат не более чем одной из упорядоченных пар. Затем идеальное паросочетание было определено как паросочетание, в котором каждый мужчина и каждая женщина принадлежат некоторой паре.

У этих концепций существует более общее выражение в понятиях теории графов. Для этого будет полезно определить понятие *двудольного графа*. Граф $G = (V, E)$ называется двудольным (bipartite), если множество узлов V может быть разбито на множества X и Y таким способом, что у каждого ребра один конец принадлежит X, а другой конец — Y. Двудольный граф изображен на рис. 1.5; часто граф, двудольность которого требуется подчеркнуть, изображается именно таким образом — узлы X и Y выстраиваются в два параллельных столбца. Впрочем, обратите внимание на то, что два графа на рис. 1.3 также являются двудольными.

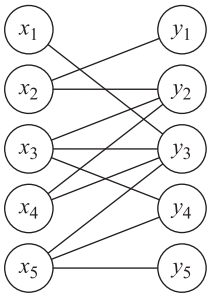


Рис. 1.5. Двудольный граф

В задаче поиска устойчивых паросочетаний результат строился из пар мужчин и женщин. В случае двудольных графов ребра представляются парами узлов, поэтому мы можем сказать, что паросочетание в графе $G = (V, E)$ представляет собой множество ребер $M \subseteq E$ с тем свойством, что каждый узел входит не более чем в одно ребро M. Паросочетание M является идеальным, если каждый узел входит ровно в одно ребро M.

Чтобы убедиться в том, что это представление отражает ту же ситуацию, которая была описана в задаче устойчивых паросочетаний, рассмотрим двудольный граф G' с множеством X из n мужчин, множеством Y из n женщин и ребрами из каждого узла X в каждый узел Y. Тогда паросочетания и идеальные паросочетания G' в точности соответствуют паросочетаниям и идеальным паросочетаниям в множествах мужчин и женщин.

В задаче устойчивых паросочетаний в ситуацию была добавлена система предпочтений. Здесь предпочтения не рассматриваются, но сама природа задачи для произвольных двудольных графов добавляет другой источник сложности: не гарантировано существование ребра из каждого узла $x \in X$ в каждый узел $y \in Y$, так что множество возможных паросочетаний имеет весьма сложную структуру. Другими словами, все выглядит так, словно только некоторые комбинации мужчин и женщин желают образовать пары, и мы хотим определить, как составить множество пар в соответствии с этими ограничениями. Для примера рассмотрим двудольный граф G на рис. 1.5; в G существует много паросочетаний, но только одно идеальное паросочетание (а вы его видите?).

Паросочетания в двудольных графах позволяют моделировать ситуации, в которых объекты *связываются* с другими объектами. Например, узлы X могут представлять задания, узлы Y — машины, а ребра (x_i, y_j) — показывать, что машина y_j

способна обработать задание x_i. Тогда идеальное паросочетание описывает такой способ назначения каждого задания машине, которая может его обработать, при котором каждой машине назначается ровно одно задание. Весной на кафедрах информатики часто рассматриваются двудольные графы, в которых X — множество профессоров, Y — множество предлагаемых курсов, а ребро (x_i, y_j) означает, что профессор x_i может преподавать курс y_j. Идеальное паросочетание в таком графе представляет собой связывание каждого профессора с курсом, который он может преподавать, таким образом, что для каждого курса будет назначен преподаватель.

Итак, *задача поиска паросочетаний в двудольном графе* формулируется следующим образом: для имеющегося произвольного двудольного графа G найти паросочетание максимального размера. Если $|X| = |Y| = n$, то идеальное паросочетание существует в том и только в том случае, если максимальное паросочетание имеет размер n. Как выясняется, алгоритмические методы, рассматривавшиеся ранее, не позволяют сформулировать эффективный алгоритм для этой задачи. Однако существует очень элегантный и эффективный алгоритм поиска максимального паросочетания; он индуктивно строит паросочетания все большего размера с избирательным возвратом в ходе выполнения.

Этот процесс, называемый *приращением* (augmentation), является центральным компонентом в большом классе эффективно решаемых задач, называемых задачами *нахождения потока в сети*.

Независимое множество

А теперь перейдем к чрезвычайно общей задаче, которая включает большинство более ранних задач как частные случаи. Для заданного графа $G = (V, E)$ множество узлов $S \subseteq V$ называется *независимым*, если никакие два узла, входящие в S, не соединяются ребром. Таким образом, задача поиска независимого множества формулируется так: для заданного графа G найти независимое множество максимально возможного размера. Например, максимальный размер независимого множества для графа на рис. 1.6 равен 4: оно состоит из четырех узлов {1, 4, 5, 6}.

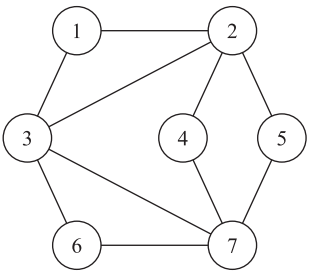


Рис. 1.6. Размер наибольшего независимого множества в этом графе равен 4

Задача поиска независимого множества представляет произвольную ситуацию, в которой вы пытаетесь выбрать некоторые объекты из набора, в то время как между некоторыми объектами существуют попарные конфликты. Допустим, у вас есть

n друзей, но некоторые пары терпеть друг друга не могут. Сколько друзей можно пригласить на обед, если вы хотите избежать лишней напряженности? По сути, речь идет о поиске наибольшего независимого множества в графе, узлы которого представляют ваших друзей, а ребра — конфликты в парах.

Задачи интервального планирования и паросочетаний в двудольном графе могут рассматриваться как особые случаи задачи независимого множества. Для задачи интервального планирования определите граф $G = (V, E)$, в котором узлы соответствуют интервалам, а каждая перекрывающаяся пара соединяется ребром; независимые множества в G соответствуют совместимым подмножествам интервалов. С представлением задачи паросочетаний в двудольном графе ситуация чуть менее тривиальна. Для заданного двудольного графа $G' = (V', E')$ выбираемые объекты соответствуют ребрам, а конфликты возникают между двумя ребрами, имеющими общий конец (такие пары ребер не могут принадлежать общему паросочетанию). Соответственно мы определяем граф $G = (V, E)$, в котором множество узлов V эквивалентно множеству ребер E' в G'. Мы определяем ребро между каждой парой элементов V, соответствующих ребрам G' с общим концом. Теперь можно проверить, что независимые множества G точно соответствуют паросочетаниям G'.

Проверка не так уж сложна, но чтобы уследить за преобразованиями «ребра в узлы, узлы в ребра», придется как следует сосредоточиться[1].

С учетом общего характера задачи независимого множества эффективный алгоритм ее решения должен получиться весьма впечатляющим. Он должен неявно включать алгоритмы для задач интервального планирования, паросочетаний в двудольном графе, а также ряда других естественных задач оптимизации.

Текущее состояние задачи независимого множества таково: эффективные алгоритмы для ее решения неизвестны, и есть гипотеза, что такого алгоритма не существует. Очевидный алгоритм, действующий методом «грубой силы», перебирает все подмножества узлов, проверяет каждое из них на независимость, после чего запоминает самое большое из обнаруженных подмножеств. Возможно, это решение близко к лучшему из того, что возможно сделать в этой задаче. Позднее в книге будет показано, что задача независимого множества принадлежит к большому классу задач, называемых *NP-полными*. Ни для одной из этих задач не известен эффективный алгоритм решения, но все они эквивалентны в том смысле, что решение любой такой задачи будет точно подразумевать существование такого решения у всех.

Естественно задать вопрос: можно ли сказать хоть что-нибудь положительное о сложности задачи независимого множества? Кое-что можно: если имеется граф G из 1000 узлов и мы захотим убедить вас в том, что он содержит независимое множество S с размером 100, сделать это будет несложно. Достаточно показать граф G,

[1] Для любознательных читателей заметим, что не каждая конфигурация задачи поиска независимых множеств может быть получена из задач интервального планирования или двудольных паросочетаний: полная задача независимых множеств является более общей. Граф на рис. 1.3, *а* не может быть сгенерирован как «граф конфликтов» в задаче интервального планирования, а граф на рис. 1.3, *б* — как граф конфликтов в задаче двудольных паросочетаний.

выделить узлы S красным цветом и позволить вам убедиться в том, что никакие два узла не соединены ребром. Итак, между проверкой того, что множество действительно является большим независимым, и непосредственным поиском большого независимого множества существует огромная разница. На первый взгляд замечание кажется тривиальным (и действительно является таковым), но оно играет ключевую роль в понимании этого класса задач. Более того, как вы вскоре увидите, в некоторых особо сложных задачах даже может не быть простого способа «проверки» решений в этом смысле.

Задача конкурентного размещения

Наконец, мы приходим к пятой задаче, которая базируется на следующей игре для двух участников. Допустим, имеются две крупные компании, которые открывают свои сети кофеен по всей стране (назовем их JavaPlanet и Queequeg's Coffee); в настоящее время они соревнуются за свою долю рынка в некотором географическом регионе. Сначала JavaPlanet открывает свое заведение; затем Queequeg's Coffee открывает свое; потом JavaPlanet; снова Queequeg's, и т. д. Допустим, они должны выполнять градостроительные нормы, согласно которым две кофейни не должны располагаться слишком близко друг к другу, и каждая компания пытается разместить свои заведения в самых удобных точках. Кто выиграет?

Сделаем правила «игры» более конкретными. Географический регион, о котором идет речь, разделен на n зон с номерами 1, 2, ..., n. Каждой зоне i присваивается оценка b_i — доход, который получит любая из компаний, открывшая кофейню в этой зоне. Наконец, некоторые пары зон (i, j) являются смежными, а местные градостроительные нормы запрещают открывать кофейни в двух смежных зонах независимо от того, какой компании они принадлежат (а также запрещают открывать две кофейни в одной зоне). Для моделирования этих конфликтов мы воспользуемся графом $G = (V, E)$, где V — множество зон, а (i, j) — ребро в E, если зоны i и j являются смежными. Таким образом, градостроительные нормы требуют, чтобы полный набор кофеен образовал независимое множество в G.

В нашей игре два игрока P_1 и P_2 поочередно выбирают узлы в G, игрок P_1 делает первый ход. Набор всех выбранных узлов в любой момент времени должен образовать независимое множество в G. Предположим, игрок P_2 имеет целевую границу B, и мы хотим узнать: существует ли для P_2 такая стратегия, чтобы независимо от выбора ходов P_1 игрок P_2 мог выбрать множество узлов с суммарной оценкой не ниже B?

Мы назовем эту формулировку *задачей конкурентного размещения*.

Для примера возьмем набор данных на рис. 1.7; предположим, игроку P_2 установлена целевая граница $B = 20$. В этом случае у P_2 имеется выигрышная стратегия. С другой стороны, при $B = 25$ у игрока P_2 такой стратегии нет.

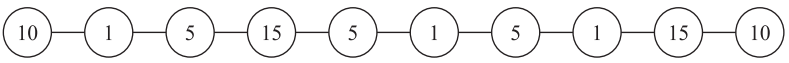


Рис. 1.7. Экземпляр задачи конкурентного размещения

В этом нетрудно убедиться, изучая диаграмму в течение некоторого времени; но для этого потребуется проверка разных сценариев в виде «Если P_1 ходит так, то P_2 пойдет вот так; но если P_1 походит так, то P_2 ответит так...» И похоже, это обстоятельство является неотъемлемой особенностью задачи: мало того что проверка наличия у P_2 выигрышной стратегии имеет высокую вычислительную сложность; в графе сколько-нибудь значительного размера нам будет трудно даже убедить вас в наличии у P_2 выигрышной стратегии. Не существует короткого доказательства, которое можно было бы продемонстрировать; вместо этого приходится приводить объемистый анализ множества всех возможных ходов.

В этом отношении задача отличается от задачи независимого множества, в которой, насколько нам известно, найти большое решение сложно, зато предложенное большое решение проверяется легко. Эти отличия формализуются в классе *PSPACE-полных задач*, к которому принадлежит задача конкурентного размещения. Считается, что PSPACE-полные задачи строго сложнее *NP*-полных задач, и это предполагаемое отсутствие коротких «доказательств» их решений является признаком этой повышенной сложности. Концепция PSPACE-полноты отражает широкий спектр задач, связанных с играми и планированием; многие из них считаются фундаментальными проблемами в области искусственного интеллекта.

Упражнения с решениями

Упражнение с решением 1

Имеется город, в котором n мужчин и n женщин пытаются вступить в брак. У каждого мужчины имеется список предпочтений, содержащий оценки всех женщин, а у каждой женщины — список предпочтений с оценками всех мужчин.

Множество всех $2n$ людей разделено на две категории: *хорошие* люди и *плохие* люди. Предположим, для некоторого числа k ($1 \le k \le n - 1$) имеются k хороших мужчин и k хороших женщин; следовательно, существует $n - k$ плохих мужчин и $n - k$ плохих женщин.

Каждый участник предпочитает любого хорошего человека любому плохому человеку. Формально каждый список предпочтений обладает тем свойством, что оценка любого хорошего человека противоположного пола выше оценки любого плохого человека противоположного пола: первые k позиций в нем занимают хорошие люди (противоположного пола) в некотором порядке, а затем идут $n - k$ плохих людей (противоположного пола) в некотором порядке.

Покажите, что в каждом устойчивом паросочетании каждый хороший мужчина вступает в брак с хорошей женщиной.

Решение

Естественный подход к подобным задачам — предположить, что утверждение ложно, и попытаться прийти к противоречию. Что будет означать ложность исходной

предпосылки в данном случае? Что существует некоторое устойчивое паросочетание M, в котором хороший мужчина m находится в паре с плохой женщиной w.

Теперь нужно понять, как выглядят остальные пары из M. Всего есть k хороших мужчин и k хороших женщин. Может ли при этом каждая хорошая женщина находиться в паре с хорошим мужчиной в этом паросочетании M? Нет: один из хороших мужчин (а именно m) уже выбрал плохую женщину, поэтому осталось всего $k - 1$ других хороших мужчин. Следовательно, даже если все остальные находятся в паре с хорошими женщинами, все равно остается некоторая плохая женщина, которая находится в паре с плохим мужчиной.

Обозначим ее w'. Теперь в M легко выявляется неустойчивость: возьмем пару (m, w'). Каждый из ее участников хорош, но находится в паре с плохим партнером. Получается, что каждый из участников потенциальной пары m и w' предпочитает другого своему текущему партнеру, а следовательно, пара (m, w') создает неустойчивость. Это противоречит исходному допущению об устойчивости M, что и требовалось доказать. ■

Упражнение с решением 2

Можно рассмотреть обобщение задачи устойчивых паросочетаний, в котором какие-то пары «мужчина-женщина» явно запрещены. В случае с нанимателями и кандидатами можно представить, что у некоторых кандидатов отсутствуют необходимые сертификаты и их не примут на работу в некоторые компании, как бы перспективно они ни выглядели. В аналогии с браком имеется множество M из n мужчин, множество W из n женщин и множество $F \subseteq M \times W$ пар, брак между которыми попросту *запрещен*. Каждый мужчина m определяет оценки для всех женщин w, для которых $(m, w) \notin F$, и каждая женщина w' определяет оценки для всех мужчин m', для которых $(m', w') \notin F$.

В этой более общей постановке задачи паросочетание S называется *устойчивым*, если в нем не проявляются никакие из следующих признаков неустойчивости:

1. В S присутствуют две пары (m, w) и (m', w'), у которых $(m, w') \notin F$, m предпочитает w' женщине w, а w' предпочитает m мужчине m' (обычная разновидность нестабильности).
2. Существует пара $(m, w) \in S$ и мужчина m', такие что m' не входит ни в одну пару в сочетании, $(m', w) \notin F$ и w предпочитает m' мужчине m. (Одинокий мужчина, более предпочтительный и не запрещенный.)
3. Существует пара $(m, w) \in S$ и женщина w', такие что w' не входит ни в одну пару в сочетании, $(m, w') \notin F$ и m предпочитает w' женщине w. (Одинокая женщина, более предпочтительная и не запрещенная.)
4. Существует мужчина m и женщина w, ни один из которых не входит ни в одну пару в паросочетании, так что $(m, w) \notin F$. (Существуют два одиноких человека, которым ничто не мешает вступить в брак друг с другом.)

Обратите внимание: по этим более общим определениям устойчивое паросочетание не обязано быть идеальным.

Теперь можно задать вопрос: всегда ли существует устойчивое паросочетание для каждого множества списков предпочтений и каждого множества запрещенных пар? Чтобы ответить на этот вопрос, можно пойти по одному из двух путей: 1) предоставить алгоритм, который для любого множества списков предпочтений и запрещенных пар будет создавать устойчивое паросочетание, или 2) привести пример множества списков предпочтений и запрещенных пар, для которых не существует устойчивого паросочетания.

Решение

Алгоритм Гейла–Шепли в высшей степени надежен по отношению к вариациям задачи устойчивого паросочетания. Итак, если вы имеете дело с новой разновидностью задачи и не можете найти контрпример для устойчивости, часто бывает полезно проверить, действительно ли прямая адаптация алгоритма Гейла–Шепли породит устойчивое паросочетание.

Перед нами один из таких случаев. Мы покажем, что устойчивое паросочетание существует всегда, даже в более общей модели с запрещенными парами, и для этого мы адаптируем алгоритм Гейла–Шепли. Мы рассмотрим, почему исходный алгоритм Гейла–Шепли не может использоваться напрямую. Конечно, сложность связана с тем, что алгоритму Гейла–Шепли ничего не известно о запретных парах и условие цикла *Пока*:

```
Пока существует свободный мужчина m, который еще не успел сделать предложение
каждой женщине
```

не сработает; m не должен делать предложение женщине w, для которой пара (m, w) является запрещенной.

Рассмотрим модификацию алгоритма Гейла–Шепли, в которую будет внесено только одно изменение: мы изменим условие цикла *Пока* и приведем его к виду:

```
Пока существует свободный мужчина m, который еще не успел сделать предложение
каждой женщине w, для которой (m, w) ∉ F
```

А теперь приведем полный алгоритм:

```
В начальном состоянии все m ∈ M и w ∈ W свободны.
Пока существует свободный мужчина m, который еще не успел сделать предложение
каждой женщине w, для которой (m, w) ∉ F
    Выбрать такого мужчину m.
    Выбрать w – женщину с наивысшей оценкой в списке предпочтений m,
        которой m еще не делал предложения.
    Если w свободна, то
        пара (m, w) вступает в состояние помолвки.
    Иначе w в настоящее время помолвлена с m'.
        Если w предпочитает m' мужчине m, то
            m остается свободным.
        Иначе w предпочитает m мужчине m'
            пара (m, w) вступает в состояние помолвки.
            m' становится свободным.
```

```
        Конец Если
    Конец Если
Конец Пока
Вернуть множество S помолвленных пар
```

Сейчас мы докажем, что этот алгоритм дает устойчивое паросочетание в соответствии с новым определением устойчивости.

Прежде всего, утверждения (1.1), (1.2) и (1.3) из текста остаются истинными (в частности, завершение алгоритма потребует не более n^2 итераций). Кроме того, нам не нужно доказывать, что полученное паросочетание S идеально (на самом деле оно может таковым не быть). Также следует обратить внимание на дополнительные факты. Если m — мужчина, который не входит в пару из S, то m должен был сделать предложение всем не запрещенным женщинам; и если w — женщина, которая не входит в пару из S, то из этого следует, что ни один мужчина не делал ей предложение.

Наконец, осталось доказать последнее утверждение:

(1.9) В отношении возвращаемого паросочетания S не существует неустойчивости.

Доказательство. Наше общее определение неустойчивости состоит из четырех частей, таким образом, нам нужно убедиться в том, что не встречается ни одно из четырех нежелательных явлений.

Для начала предположим, что существует неустойчивость типа (1), состоящая из пар (m, w) и (m', w') в S и обладающая тем свойством, что $(m, w') \notin F$, m предпочитает w' женщине w, а w' предпочитает m мужчине m'. Из этого следует, что m должен был сделать предложение w'; получается, что w' ему отказала, а значит, она предпочитает своего окончательного партнера m — противоречие.

Далее предположим, что существует неустойчивость типа (1), состоящая из пары $(m, w) \in S$ и мужчины m' и обладающая тем свойством, что m' не входит ни в одну пару, $(m', w) \notin F$ и w предпочитает m' мужчине m. Из этого следует, что m' должен был сделать предложение w и получить отказ; и снова это означает, что w предпочитает своего окончательного партнера m' — противоречие.

Если существует неустойчивость типа (3), состоящая из пары $(m, w) \in S$ и женщины w' и обладающая тем свойством, что w' не входит ни в одну пару, $(m, w') \notin F$ и m предпочитает w' женщине w. В этом случае ни один мужчина вообще не делал предложения w'; в частности, m не делал предложения w', поэтому он должен предпочитать w женщине w' — противоречие.

Наконец, предположим, что существует неустойчивость типа (4), состоящая из мужчины m и женщины w, ни один из которых не является частью ни одной пары в паросочетании, так что $(m, w') \notin F$. Но чтобы m не имел пары, он должен был сделать предложение каждой женщине, брак с которой ему не запрещен; в частности, он должен был сделать предложение w, но тогда она не была бы одинока — противоречие. ■

Упражнения

1. Решите, является ли приведенное ниже утверждение истинным или ложным. Если оно истинно, приведите короткое объяснение; если ложно — приведите контрпример.

   Истинно или ложно? В любой конфигурации задачи устойчивых паросочетаний существует устойчивое паросочетание, содержащее пару (m, w), в котором m находится на первом месте в списке предпочтений w, а w находится на первом месте в списке предпочтений m.

2. Решите, является ли приведенное ниже утверждение истинным или ложным. Если оно истинно, приведите короткое объяснение; если ложно — приведите контрпример.

   Истинно или ложно? Допустим, имеется конфигурация задачи устойчивых паросочетаний с мужчиной m и женщиной w, для которых m находится на первом месте в списке предпочтений w, а w находится на первом месте в списке предпочтений m. В этом случае в каждом устойчивом паросочетании S для этой конфигурации пара (m, w) принадлежит S.

3. Существует много других ситуаций, в которых можно задавать вопросы, относящиеся к некой разновидности принципа «устойчивости». В следующем примере используется формулировка, основанная на конкуренции между двумя компаниями.

   Допустим, имеются две телевизионные сети; назовем их $\mathcal{A}$ и $\mathcal{B}$. Также доступны n программных окон в течение времени массового просмотра, а каждая сеть проводит n телевизионных передач. Каждая сеть хочет создать *расписание* — схему распределения передач по разным окнам, чтобы обеспечить себе как можно большую долю рынка.

   Следующий способ поможет сравнить, насколько хорошо две сети справляются со своей задачей, исходя из их расписания. У каждой передачи имеется фиксированный рейтинг, основанный на количестве людей, посмотревших передачу в прошлом году; будем считать, что рейтинги двух передач никогда не совпадают. Сеть выигрывает заданное окно, если передача, которую она ставит в это окно, обладает более высоким рейтингом, чем передача в расписании других сетей для этого окна. Цель каждой сети — захватить как можно больше окон.

   Допустим, в первую неделю осеннего сезона сеть $\mathcal{A}$ публикует расписание S, а сеть $\mathcal{B}$ — расписание T. На основе этой пары расписаний каждая сеть захватывает некоторые окна в соответствии с приведенным выше правилом. Пара расписаний (S, T) будет называться *устойчивой*, если ни одна из сетей не может в одностороннем порядке изменить свое расписание и получить дополнительные окна. Иначе говоря, не существует расписания S' такого, что сеть $\mathcal{A}$ с парой (S', T) получит больше окон, чем с парой (S, T). По соображениям симметрии также не существует расписания T', с которым сеть $\mathcal{B}$ со своей парой (S, T') захватит больше окон, чем с парой (S, T).

Аналогия вопроса Гейла и Шепли для подобной устойчивости выглядит так: можно ли утверждать, что для каждого множества телепередач и оценок всегда существует устойчивая пара расписаний? Чтобы ответить на этот вопрос, сделайте одно из двух:

(а) приведите алгоритм, который для любого множества передач и связанных с ними оценок строит устойчивую пару расписаний; или

(б) приведите пример множества передач и связанных с ними оценок, для которого не существует устойчивой пары расписаний.

4. Гейл и Шепли опубликовали свою статью о задаче устойчивых паросочетаний в 1962 году; однако разновидность их алгоритма к тому времени уже почти 10 лет использовалась Национальной программой распределения студентов-медиков по больницам.

   По сути, ситуация выглядела так: было m больниц, в каждой из которых имелось некоторое количество вакансий. Также было n студентов-медиков, которые получали диплом и были заинтересованы в поступлении в одну из больниц. Каждая больница строила оценки студентов в порядке предпочтения, и каждый студент строил оценки больниц в порядке предпочтения. Будем считать, что количество выпускников превышает количество доступных вакансий в m больницах.

   Естественно, задача заключалась в поиске способа связывания каждого студента не более чем с одной больницей, при котором были бы заполнены все вакансии во всех больницах. (Так как предполагается, что количество студентов превышает количество вакансий, некоторые студенты не будут приняты ни в одну из больниц.)

   Распределение студентов по больницам называется *устойчивым*, если не встречается ни одна из следующих ситуаций:

- Неустойчивость первого типа: имеются студенты s и s', а также больница h, для которых:
  - s назначается в h,
  - s' не назначается ни в одну больницу,
  - h предпочитает s' студенту s.
- Неустойчивость второго типа: имеются студенты s и s', а также больницы h и h', для которых:
  - s назначается в h,
  - s' назначается в h',
  - h предпочитает s' студенту s,
  - s предпочитает h студенту h'.

  Фактически ситуация повторяет задачу устойчивых паросочетания, если не считать того, что 1) в больницах обычно открыто более одной вакансии и 2) студентов больше, чем вакансий.

Покажите, что всегда существует устойчивое распределение студентов по больницам, и приведите алгоритм поиска такого распределения.

5. Задача устойчивых паросочетаний в том виде, в каком она рассматривалась в тексте, подразумевает, что у всех мужчин и женщин имеются полностью упорядоченные списки предпочтений. В этом упражнении будет рассмотрена версия задачи, в которой мужчины и женщины могут считать некоторые варианты равноценными. Как и прежде, имеется множество M из n мужчин и множество W из n женщин. Допустим, каждый мужчина и каждая женщина оценивают представителей противоположного пола, но в списке предпочтений допускаются совпадения. Например (с $n = 4$) женщина может сказать, что m_1 находится на первом месте; второе место делят m_2 и m_3, а последнее место занимает m_4. Мы говорим, что w предпочитает m мужчине m', если m занимает в ее списке предпочтений более высокое место, чем m' (при отсутствии равенства).

   При наличии совпадений в предпочтениях возможны две естественные концепции устойчивости. И в каждом случае можно задаться вопросом о существовании устойчивых паросочетаний.

   (a) *Сильная неустойчивость* в идеальном паросочетании S состоит из мужчины m и женщины w, каждый из которых предпочитает другого своему текущему партнеру в S. Всегда ли существует идеальное паросочетание без сильной неустойчивости? Либо приведите пример множества мужчин и женщин со списками предпочтений, в котором в каждом идеальном паросочетании имеется сильная неустойчивость, либо приведите алгоритм, который гарантированно находит идеальное паросочетание без сильной неустойчивости.

   (b) *Слабая неустойчивость* в идеальном паросочетании S состоит из мужчины m и женщины w, находящихся в паре с w' и m' соответственно, при выполнении одного из следующих условий:

   - m предпочитает w женщине w' и w либо предпочитает m мужчине m', либо считает эти варианты равноценными; или
   - w предпочитает m женщине m' и m либо предпочитает w женщине w', либо считает эти варианты равноценными.

   Другими словами, пара, в которой участвуют m и w, либо предпочтительно для обоих, либо предпочтительна для одного, тогда как другому все равно. Всегда ли существует идеальное паросочетание без слабой неустойчивости? Либо приведите пример множества мужчин и женщин со списками предпочтений, в котором в каждом идеальном паросочетании имеется слабая неустойчивость, либо приведите алгоритм, который гарантированно находит идеальное паросочетание без слабой неустойчивости.

6. Транспортной компании Peripatetic Shipping Lines, Inc., принадлежат n кораблей, которые обслуживают n портов. У каждого корабля имеется расписание, в котором для каждого дня месяца указано, в каком порту он стоит в настоящий

момент (или находится в море). (Можно считать, что месяц состоит из m дней, $m > n$.) Каждый корабль посещает каждый порт ровно на один день в месяц. По соображениям безопасности в PSL Inc. действует следующее жесткое требование:

† Никакие два корабля не могут находиться в одном порту в один день.

Компания хочет провести техническое обслуживание всех своих кораблей по следующей схеме. Расписание каждого корабля S_i *сокращается*: в какой-то день он прибывает в назначенный порт и остается здесь до конца месяца. Это означает, что корабль S_i в этом месяце не будет посещать остальные порты по расписанию, но это нормально. Итак, сокращенный вариант расписания S_i состоит из исходного расписания до заданной даты, в которой корабль приходит в порт P; всю оставшуюся часть расписания он просто остается в порту P.

Компания хочет получить ответ на следующий вопрос: Как при заданных расписаниях каждого корабля построить сокращенную версию каждого расписания, чтобы условие (†) продолжало выполняться: никакие два корабля никогда не должны оказываться в одном порту в один день. Покажите, что такое множество сокращений всегда может быть найдено, и приведите алгоритм для его построения.

Пример. Допустим, имеются два корабля и два порта и «месяц» состоит из четырех дней. Расписание первого корабля выглядит так:

порт P_1; *в море*; *порт* P_2; *в море*

а расписание второго корабля выглядит так:

в море; *порт* P_1; *в море*; *порт* P_2

Существует только один вариант выбора сокращенных расписаний — первый корабль остается в порту P_2, начиная с дня 3, а второй корабль остается в порту P_1, начиная с дня 2.

7. Ваши друзья работают в компании CluNet, занимающейся созданием крупных коммуникационных сетей. Им нужен алгоритм, управляющий работой конкретного типа коммутатора.

   Существует n входных проводов и n выходных проводов, каждый из которых направлен от источника к приемнику данных. Каждый входной провод стыкуется с выходным через специальное устройство, называемом *соединительным блоком*. Точки на проводах естественным образом упорядочиваются в направлении от источника к приемнику; для двух разных точек x и y на одном проводе точка x расположена *выше* y, если x находится ближе к источнику, чем y. В противном случае говорят, что точка x находится *ниже* точки y. Порядок, в котором один входной провод контактирует с выходными проводами, не обязательно совпадает с порядком, в котором с ними контактирует другой входной провод. (Это относится и к порядку, в котором выходные провода контактируют со входными.) На рис. 1.8 приведен пример такой конфигурации входных и выходных проводов.

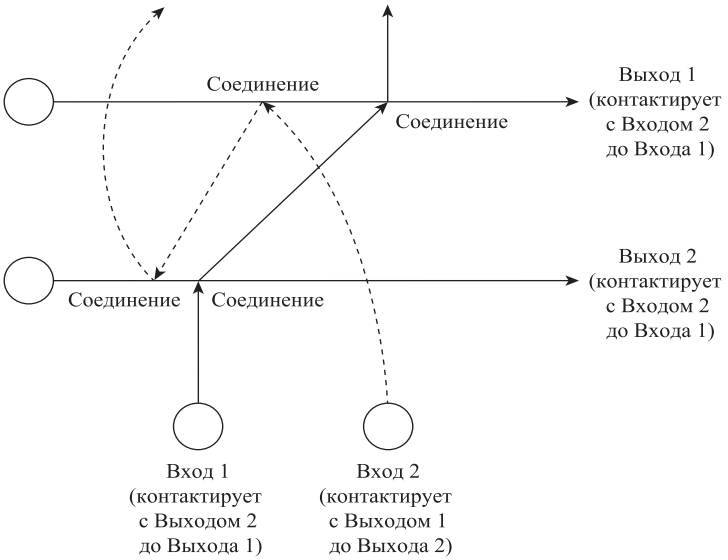


Рис. 1.8. Конфигурация с двумя входными и двумя выходными проводами. Вход 1 контактирует с Выходом 2 выше контакта с Выходом 1; Вход 2 контактирует с Выходом 1 выше контакта с Выходом 2. Допустимым решением будет переключение потока данных Входа 1 на Выход 2, а потока данных Входа 2 — на Выход 1. С другой стороны, если переключить поток Входа 1 на Выход 1, а поток Входа 2 — на Выход 2, то оба потока будут проходить через соединительный блок в точке контакта Входа1 и Выхода 2 — а это недопустимо

А теперь перейдем собственно к коммутации. По каждому входному проводу передается свой поток данных, который должен быть передан на один из выходных проводов. Если поток Входа *i* переключается на Выход *j* в соединительном блоке *B*, то этот поток проходит через все соединительные блоки, расположенные выше *B* на Входе *i*, затем через *B* и через все соединительные блоки, расположенные ниже *B* на Выходе *j*. Неважно, какой входной поток данных переключается на тот или иной выходной провод, но каждый входной поток должен быть подан на *отдельный* выходной провод. Кроме того — в этом-то вся сложность! — никакие два потока данных не могут проходить через один соединительный блок после операции переключения.

Итак, задача: покажите, что для любой заданной схемы контактов между входными и выходными проводами (каждая пара контактирует ровно один раз) всегда можно найти допустимое переключение потоков данных, при котором каждый входной поток переключается на свой выходной поток и никакие два итоговых потока не проходят через один соединительный блок. Также приведите алгоритм для поиска допустимой схемы переключений.

8. В этой задаче рассматривается проблема *правдивости* алгоритма устойчивых паросочетаний, а конкретнее — алгоритма Гейла–Шепли. Основной вопрос заключается в следующем: могут ли мужчина или женщина улучшить свое положение, солгав относительно своих предпочтений? Допустим, что у каж-

дого участника имеется истинный порядок предпочтений. Теперь рассмотрим женщину w. Предположим, w предпочитает мужчину m мужчине m', но и m, и m' находятся на нижних позициях ее списка предпочтений. Может ли случиться так, что переключение порядка m и m' в списке предпочтений (например, ложным утверждением о том, что она предпочитает m' мужчине m) и выполнением алгоритма с ложным списком предпочтений w окажется в паре с мужчиной m'', которого она в действительности предпочитает как m, так и m'? (Тот же вопрос можно задать и для мужчин, но в контексте нашего упражнения мы ограничимся женщинами.)

Чтобы ответить на этот вопрос, сделайте одно из двух:

(а) приведите доказательство того, что для любого множества списков предпочтений изменение порядка следования пары в списке предпочтений не сможет улучшить статус партнера женщины в алгоритме Гейла–Шепли; или

(б) приведите пример множества списков предпочтений, для которых существует изменение порядка следования пары, способное улучшить статус партнера женщины.

Примечания и дополнительная литература

Задача устойчивых паросочетаний была впервые определена и проанализирована Гейлом и Шепли (Gale, Shapley, 1962); по словам Дэвида Гейла, побудительной причиной для анализа задачи стала история, прочитанная ими в еженедельнике «Нью-Йоркер» о сложностях организации вступительных экзаменов в колледжах (Gale, 2001). Поиск устойчивых паросочетаний превратился в самостоятельную область исследований, которая рассматривается в книгах Гасфилда и Ирвинга (Gusfield, Irving, 1989) и Кнута (Knuth, 1997c). Гасфилд и Ирвинг также предоставляют хороший обзор «параллельной» истории задачи устойчивых паросочетаний на примере механизма, который был разработан для подбора сочетаний кандидатов и нанимателей в медицине и других областях.

Как упоминалось в этой главе, наши пять типичных задач будут занимать центральное место в основных темах книги — а именно «жадных» алгоритмах, динамическом программировании, нахождении потока в сети, *NP*-полноте и PSPACE-полноте. Все эти задачи будут рассмотрены в указанных контекстах позднее в книге.

Глава 2
Основы анализа алгоритмов

Основной задачей анализа алгоритмов является выявление закономерности масштабирования требований к ресурсам (затраты времени и памяти) с возрастанием размера входных данных. В начале этой главы речь пойдет о том, как эта концепция применяется в конкретных ситуациях, так как конкретизация помогает понять суть понятия вычислительной разрешимости. После этого будет разработан математический механизм, необходимый для описания закономерностей скорости роста различных функций с увеличением размера входных данных; он помогает точно определить, что же понимается под выражением «одна функция растет быстрее другой».

Затем мы разработаем граничные оценки времени выполнения некоторых базовых алгоритмов, начиная с реализации алгоритма Гейла–Шепли из главы 1. Далее в обзор будут включены оценки времени выполнения и некоторые специфические характеристики алгоритмов, обеспечивающие это время выполнения. В некоторых случаях достижение хорошего времени выполнения зависит от использования более сложных структур данных, и в конце главы будет рассмотрен очень полезный пример такой структуры данных: *приоритетные очереди* и их реализация на базе кучи (heap).

2.1. Вычислительная разрешимость

Главной темой этой книги является поиск эффективных алгоритмов для вычислительных задач. На таком уровне общности к этой теме вроде бы можно отнести всю область компьютерных вычислений; чем же наш подход будет отличаться от других?

Сначала мы попробуем выявить общие темы и принципы проектирования при разработке алгоритмов. Нас будут интересовать парадигматические задачи и методы, которые с минимумом второстепенных подробностей будут демонстрировать базовые методы проектирования эффективных алгоритмов. Бессмысленно обсуждать эти принципы проектирования «в вакууме», поэтому рассматриваемые задачи и методы почерпнуты из фундаментальных в компьютерной науке тем, а общее изучение алгоритмов дает неплохое представление о вычислительных концепциях, встречающихся во многих областях.

Другое свойство, встречающееся во многих задачах, — их принципиальная *дискретность*. Иначе говоря, как и задача устойчивых паросочетаний, они подразумевают неявный поиск по большому набору комбинаторных вариантов; целью является эффективный поиск решения, удовлетворяющего некоторым четко определенным условиям.

Пытаясь разобраться в общих концепциях вычислительной эффективности, мы сосредоточимся прежде всего на эффективности по времени выполнения: алгоритмы должны работать быстро. Но важно понимать, что алгоритмы могут быть эффективны в использовании других ресурсов. В частности, объем памяти, используемой алгоритмом, также может оказаться важным аспектом эффективности, который будет неоднократно упоминаться в книге. Вы увидите, какие приемы могут использоваться для сокращения объема памяти, необходимого для выполнения вычислений.

Первые попытки определения эффективности

Первый серьезный вопрос, на который нужно ответить, выглядит так: как преобразовать размытое понятие «эффективный алгоритма» в нечто более конкретное?

На первый взгляд рабочее определение эффективности могло бы выглядеть так.

Предлагаемое определение эффективности (1): *алгоритм называется эффективным, если его реализация быстро выполняется на реальных входных данных*.

Немного поразмыслим над этим определением. На определенном уровне с ним трудно спорить: одной из целей, заложенных в основу нашего изучения алгоритмов, является возможность быстрого решения реальных задач. И в самом деле, целая область исследований посвящена тщательной реализации и профилированию разных алгоритмов для дискретных вычислительных задач.

Однако в этом определении отсутствуют некоторые ключевые подробности, хотя нашей главной целью и является быстрое решение реальных задач на реальных компьютерах. Во-первых, в нем не указано, где и насколько хорошо реализуется алгоритм. Даже плохой алгоритм может быстро отработать с малым размером тестовых данных на исключительно быстром процессоре; даже хорошие алгоритмы могут быстро выполняться при небрежном программировании. Кроме того, какие входные данные можно считать «реальными»? Полный диапазон входных данных, которые могут встречаться на практике, неизвестен, а некоторые наборы данных могут создавать значительно больше сложностей при обработке. Наконец, предлагаемое определение не учитывает, насколько хорошо (или плохо) алгоритм масштабируется с ростом задачи до непредвиденных уровней. Типичная ситуация: два совершенно разных алгоритма примерно одинаково работают на входных данных размера 100; при 10-кратном увеличении размера данных один алгоритм продолжает работать быстро, а выполнение другого резко замедляется.

Итак, нам хотелось бы иметь конкретное определение эффективности — не зависящее от платформы и набора входных данных и имеющее прогностическое

значение в отношении увеличения размера входных данных. Прежде чем переходить к рассмотрению практических выводов из этого утверждения, мы можем по крайней мере проанализировать неявное высокоуровневое предположение, из которого оно вытекает: что ситуацию следует рассматривать с более математической точки зрения.

В качестве ориентира возьмем задачу устойчивых паросочетаний. У входных данных имеется естественный параметр — «размер» N; за него можно принять общий размер представления всех списков предпочтений, поскольку именно они будут передаваться на вход любого алгоритма для решения задачи. Значение N тесно связано с другим естественным параметром этой задачи: n, количеством мужчин и количеством женщин. Всего существуют $2n$ списков предпочтений, каждый из которых имеет длину n, поэтому мы можем считать, что $N = 2n^2$, игнорируя второстепенные подробности реализации представления данных. При рассмотрении задачи мы будем стремиться к описанию алгоритмов на высоком уровне, а затем проводить математический анализ времени выполнения как функции размера входных данных N.

Худшее время выполнения и поиск методом «грубой силы»

Для начала сосредоточимся на анализе времени выполнения в худшем случае: нас будет интересовать граница наибольшего возможного времени выполнения алгоритма для всех входных данных заданного размера N и закономерность ее масштабирования с изменением N. На первый взгляд повышенное внимание, уделяемое быстродействию в худшем случае, кажется излишне пессимистичным: а если алгоритм хорошо работает в большинстве случаев и есть лишь несколько патологических вариантов входных данных, с которыми он работает очень медленно? Конечно, такие ситуации возможны, но, как показали исследования, в общем случае анализ худшей производительности алгоритма неплохо отражает его практическую эффективность. Более того, при выборе пути математического анализа трудно найти практическую альтернативу для анализа худшего случая. Анализ среднего случая — очевидная альтернатива, при которой рассматривается производительность алгоритма, усредненная по набору «случайных» входных данных, — выглядит, конечно, заманчиво и иногда предоставляет полезную информацию, но чаще только создает проблемы. Как упоминалось ранее, очень трудно описать весь диапазон входных данных, которые могут встретиться на практике, поэтому попытки изучения производительности алгоритма для «случайных» входных данных быстро вырождаются в споры о том, как должны генерироваться случайные данные: один и тот же алгоритм может очень хорошо работать для одной категории случайных входных данных и очень плохо — для другой. В конец концов, реальные входные данные для этого алгоритма обычно берутся не из случайного распределения, поэтому риски анализа среднего случая больше скажут о средствах получения случайных данных, нежели о самом алгоритме.

Итак, в общем случае мы будем проводить анализ худшего случая времени выполнения алгоритма. Но какой разумный аналитический показатель сможет нам сообщить, хороша или плоха граница времени выполнения? Первый простой ориентир — сравнение с эффективностью поиска методом «грубой силы» по всему пространству возможных решений.

Вернемся к примеру задачи устойчивых паросочетаний. Даже при относительно небольшом размере входных данных определяемое ими пространство поиска огромно (существует $n!$ возможных идеальных паросочетаний между n мужчинами и n женщинами), а нам нужно найти паросочетание, которое является устойчивым. Естественный алгоритм «грубой силы» для такой задачи будет перебирать все идеальные паросочетания и проверять каждое из них на устойчивость. Удивительный в каком-то смысле вывод для нашего решения задачи устойчивых паросочетаний заключается в том, что для нахождения устойчивого паросочетания в этом колоссальном пространстве возможностей достаточно времени, пропорционального всего лишь N. Это решение было получено *на аналитическом уровне*. Мы не реализовали алгоритм и не опробовали его на тестовых списках предпочтений, а действовали исключительно математическими методами. Тем не менее в то же время анализ показал, как реализовать этот алгоритм на практике, и предоставил достаточно исчерпывающие доказательства того, что он существенно превосходит метод простого перебора.

Эта общая особенность встречается в большинстве задач, которыми мы будем заниматься: компактное представление, неявно определяющих колоссальное пространство поиска. Для большинства таких задач существует очевидное решение методом «грубой силы»: перебор всех возможностей и проверка каждого из них на соответствие критерию. Мало того, что этот метод почти всегда работает слишком медленно, чтобы его можно было признать практичным, — по сути, это обычная интеллектуальная увертка; он не дает абсолютно никакой информации о структуре изучаемой задачи. И если в алгоритмах, рассматриваемых в книге, и существует нечто общее, то это следующее альтернативное определение эффективности:

Предлагаемое определение эффективности (2): *алгоритм считается эффективным, если он обеспечивает (на аналитическом уровне) качественно лучшую производительность в худшем случае, чем поиск методом «грубой силы»*.

Это рабочее определение чрезвычайно полезно для нас. Алгоритмы, обеспечивающие существенно лучшую эффективность по сравнению с «грубым поиском», почти всегда содержат ценную эвристическую идею, благодаря которой достигается это улучшение; кроме того, они сообщают полезную информацию о внутренней структуре и вычислительной разрешимости рассматриваемой задачи.

Если у второго определения и есть какой-то недостаток, то это его расплывчатость. Что следует понимать под «качественно лучшей производительностью»? Похоже, нам нужно более тщательно исследовать фактическое время выполнения алгоритмов и попытаться предоставить количественную оценку времени выполнения.

Полиномиальное время как показатель эффективности

Когда аналитики только начали заниматься анализом дискретных алгоритмов (а эта область исследований стала набирать темп в 1960-е годы), у них начал формироваться консенсус относительно того, какой количественной оценкой можно описать концепцию «разумного» времени выполнения. Время поиска в естественных комбинаторных задачах склонно к экспоненциальному росту относительно размера N входных данных; если размер увеличивается на единицу, то количество возможностей возрастает мультипликативно. Для таких задач хороший алгоритм должен обладать более эффективной закономерностью масштабирования; при возрастании размера входных данных с постоянным множителем (скажем, вдвое) время выполнения алгоритма должно увеличиваться с постоянным множителем C.

На математическом уровне эта закономерность масштабироваться может быть сформулирована так. Предположим, алгоритм обладает следующим свойством: существуют такие абсолютные константы $c > 0$ и $d > 0$, что для любого набора входных данных N время выполнения ограничивается cN^d примитивными вычислительными шагами. (Иначе говоря, время выполнения не более чем пропорционально N^d.) Пока мы намеренно будем сохранять неопределенность по поводу того, что считать «примитивным вычислительным шагом», — но это понятие легко формализуется в модели, в которой каждый шаг соответствует одной ассемблерной команде на стандартном процессоре или одной строке стандартного языка программирования (такого, как C или Java). В любом случае при выполнении этой границы времени выполнения для некоторых c и d можно сказать, что алгоритм обеспечивает *полиномиальное время выполнения* или что он относится к категории алгоритмов с полиномиальным временем. Обратите внимание: любая граница с полиномиальным временем обладает искомым свойством масштабирования. Если размер входных данных возрастает с N до $2N$, то граница времени выполнения возрастает с cN^d до $c(2N)^d = c \cdot 2^d N^d$, что соответствует замедлению с коэффициентом 2^d. Так как d является константой, коэффициент 2^d тоже является константой; как нетрудно догадаться, полиномы с меньшими степенями масштабируются лучше, чем полиномы с высокими степенями.

Из новых терминов и интуитивного замечания, изложенного выше, вытекает наша третья попытка сформулировать рабочее определение эффективности.

Предлагаемое определение эффективности (3): *алгоритм считается эффективным, если он имеет полиномиальное время выполнения.*

Если предыдущее определение казалось слишком расплывчатым, это может показаться слишком жестко регламентированным. Ведь алгоритм с временем выполнения, пропорциональным n^{100} (а следовательно, полиномиальным), будет безнадежно неэффективным, не так ли? И неполиномиальное время выполнения $n^{1+0{,}02(\log n)}$ покажется нам относительно приемлемым? Конечно, в обоих случаях ответ будет положительным. В самом деле, как бы мы ни старались абстрактно оправдать определение эффективности в контексте полиномиального времени, главное оправдание будет таким: *это определение действительно работает.* У задач,

для которых существуют алгоритмы с полиномиальным временем, эти алгоритмы почти всегда работают с временем, пропорциональным полиномам с умеренной скоростью роста, такой как n, $n \log n$, n^2 или n^3. И наоборот, задачи, для которых алгоритм с полиномиальной скоростью неизвестен, обычно оказываются очень сложными на практике. Конечно, у этого принципа есть исключения с обеих сторон: например, известны случаи, в которых алгоритм с экспоненциальным поведением в худшем случае обычно хорошо работает на данных, встречающихся на практике; а есть случаи, в которых лучший алгоритм с полиномиальным временем полностью непрактичен из-за больших констант или высокого показателя степени в полиномиальной границе. Все это лишь доказывает тот факт, что наше внимание к полиномиальным границам худшего случая — всего лишь абстрактное представление практических ситуаций. Но, как оказалось, в подавляющем большинстве случаев конкретное математическое определение полиномиального времени на удивление хорошо соответствует нашим наблюдениям по поводу эффективности алгоритмов и разрешимости задач в реальной жизни.

Еще одна причина, по которой математические формальные методы и эмпирические данные хорошо сочетаются в случае полиномиальной разрешимости, связана с огромными расхождениями между скоростью роста полиномиальных и экспоненциальных функций. Допустим, имеется процессор, выполняющий миллион высокоуровневых команд в секунду, и алгоритмы с границами времени выполнения n, $n \log_2 n$, n^2, n^3, $1{,}5^n$, 2^n и $n!$. В табл. 2.1 приведено время выполнения этих алгоритмов (в секундах, минутах, днях или годах) для входных данных с размером $n = 10, 30, 50, 100, 1000, 10\,000, 100\,000$ и $1\,000\,000$.

Таблица 2.1. Время выполнения (с округлением вверх) разных алгоритмов для входных данных разного размера (для процессора, выполняющего миллион высокоуровневых команд в секунду). Если время выполнения алгоритма превышает 10^{25} лет, мы просто используем пометку «очень долго»

	n	$n \log_2 n$	n^2	n^3	$1{,}5^n$	2^n	$n!$
$n = 10$	< 1 с	< 1 с	< 1 с	< 1 с	< 1 с	< 1 с	4 с
$n = 30$	< 1 с	< 1 с	< 1 с	< 1 с	< 1 с	18 мин	10^{25} лет
$n = 50$	< 1 с	< 1 с	< 1 с	< 1 с	11 минут	36 лет	очень долго
$n = 100$	< 1 с	< 1 с	< 1 с	1 с	12 892 лет	10^{17} лет	очень долго
$n = 1000$	< 1 с	< 1 с	1 с	18 минут	очень долго	очень долго	очень долго
$n = 10\,000$	< 1 с	< 1 с	2 минуты	12 дней	очень долго	очень долго	очень долго
$n = 100\,000$	< 1 с	2 с	3 часа	32 года	очень долго	очень долго	очень долго
$n = 1\,000\,000$	1 с	20 с	12 дней	31 710 лет	очень долго	очень долго	очень долго

Наконец, у такого конкретного определения эффективности есть еще одно фундаментальное преимущество: оно позволяет выразить концепцию отсутствия эффективного алгоритма для некоторой задачи. Эта возможность является обязательным предусловием для перевода нашего изучения алгоритмов в научную плоскость, потому что вопрос существования или отсутствия эффективных алгоритмов приобретает вполне четкий смысл. Напротив, оба предыдущих определения были полностью субъективными, а следовательно, ограничивали возможность конкретного обсуждения некоторых вопросов.

В частности, с первым определением, привязанным к конкретной реализации алгоритма, эффективность становилась постоянно меняющейся величиной: с ростом скорости процессоров все больше алгоритмов подпадает под это понятие эффективности. Наше определение в терминах полиномиального времени куда ближе к абсолютным понятиям; оно тесно связано с идеей о том, что каждой задаче присущ некоторый уровень вычислительной разрешимости: у одних задач есть эффективное решение, у других его нет.

2.2. Асимптотический порядок роста

Итак, наше обсуждение вычислительной разрешимости по своей сути базируется на способности выражения того, что худшее время выполнения алгоритма для входных данных размера n растет со скоростью, которая не более чем пропорциональна некоторой функции $f(n)$. Функция $f(n)$ становится граничной оценкой времени выполнения алгоритма. Теперь нужно заложить основу для дальнейшего рассмотрения этой концепции.

Алгоритмы будут в основном выражаться на псевдокоде наподобие того, который использовался в алгоритме Гейла–Шепли. Время от времени появляется необходимость в более формальных средствах, но для большинства целей такого стиля определения алгоритмов вполне достаточно. Вычисляя граничную оценку для времени выполнения алгоритма, мы будем подсчитывать количество выполняемых шагов псевдокода; в этом контексте один шаг будет состоять из присваивания значения переменной, поиска элемента в массиве, перехода по указателю или выполнения арифметической операции с целым числом фиксированного размера.

Если нам потребуется что-то сказать о времени выполнения алгоритма с входными данными размера n, можно попытаться сделать предельно конкретное заявление, например: «Для любых входных данных размера n этот алгоритм будет выполнен не более чем за $1{,}62n^2 + 3{,}5n + 8$ шагов». В некоторых контекстах такое утверждение будет представлять интерес, но в общем случае ему присущ ряд недостатков. Во-первых, получение такой точной оценки может потребовать значительных усилий, а настолько подробная оценка все равно не нужна. Во-вторых, так как нашей конечной целью является идентификация широких классов алгоритмов, обладающих сходным поведением, нам хотелось бы различать время

выполнения с меньшим уровнем детализации, чтобы сходство между разными алгоритмами (и разными задачами) проявлялось более наглядно. И наконец, излишне точные утверждения о количестве шагов алгоритма часто оказываются (в некоторой степени) бессмысленными. Как говорилось ранее, мы обычно подсчитываем шаги в описании алгоритма на псевдокоде, напоминающем язык программирования высокого уровня. Каждый из этих шагов обычно преобразуется в некоторое фиксированное число примитивных шагов при компиляции программы в промежуточное представление, которые затем преобразуются в еще большее количество шагов в зависимости от конкретной архитектуры, используемой для проведения вычислений. Итак, максимум, что можно сказать с уверенностью, — что при рассмотрении задачи на разных уровнях вычислительной абстракции понятие «шаг» может увеличиваться или уменьшаться с постоянным коэффициентом. Например, если для выполнения одной операции языка высокого уровня требуется 25 низкоуровневых машинных команд, то наш алгоритм, выполняемый за максимум $1{,}62n^2 + 3{,}5n + 8$ шагов, может рассматриваться как выполняемый за $40{,}5n^2 + 87{,}5n + 200$ шагов при анализе на уровне, приближенном к уровню физического оборудования.

O, Ω и Θ

По всем указанным причинам мы хотим выразить скорость роста времени выполнения и других функций способом, из которого исключены постоянные факторы и составляющие низших порядков. Иначе говоря, нам хотелось бы взять время выполнения типа приведенного выше, $1{,}62n^2 + 3{,}5n + 8$, и сказать, что оно растет со скоростью n^2. А теперь посмотрим, как же решается эта задача.

Асимптотические верхние границы

Пусть $T(n)$ — функция (допустим, время выполнения в худшем случае для некоторого алгоритма) с входными данными размера n. (Будем считать, что все рассматриваемые функции получают неотрицательные значения.) Для другой функции $f(n)$ говорится, что $T(n)$ имеет порядок $f(n)$ (в условной записи $O(f(n))$), если для достаточно больших n функция $T(n)$ ограничивается сверху произведением $f(n)$ на константу. Также иногда используется запись $T(n) = O(f(n))$. А если выражаться точнее, функция $T(n)$ называется имеющей порядок $O(f(n))$, если существуют такие константы $c > 0$ и $n_0 \geq 0$, что для всех $n \geq n_0$ выполняется условие $T(n) \leq c \cdot f(n)$. В таком случае говорят, что T имеет асимптотическую верхнюю границу f. Важно подчеркнуть, что это определение требует существования константы c, работающей для *всех* n; в частности, c не может зависеть от n.

В качестве примера того, как это определение используется для выражения верхней границы времени выполнения, рассмотрим алгоритм, время выполнения которого (как в предшествующем обсуждении) задается в форме $T(n) = pn^2 + qn + r$ для положительных констант p, q и r. Мы утверждаем, что любая такая функция имеет $O(n^2)$. Чтобы понять, почему это утверждение справедливо, следует заме-

тить, что для всех $n \geq 1$ истинны условия $qn \leq qn^2$, и $r \leq rn^2$. Следовательно, можно записать

$$T(n) = pn^2 + qn + r \leq P\,n^2 + qn^2 + rn^2 = (p + q + r)n^2$$

для всех $n \geq 1$. Это неравенство в точности соответствует требованию определения $O(\cdot)$: $T(n) \leq cn^2$, где $c = p + q + r$.

Учтите, что запись $O(\cdot)$ выражает только верхнюю границу, а не точную скорость роста функции. Например, по аналогии с утверждением, что функция $T(n) = pn^2 + qn + r$ имеет $O(n^2)$, также будет правильно утверждать, что она имеет $O(n^3)$. В самом деле, мы только что выяснили, что $T(n) \leq (p + q + r)n^2$, а поскольку также $n^2 \leq n^3$, можно заключить, что $T(n) \leq (p + q + r)n^3$, как того требует определение $O(n^3)$. Тот факт, что функция может иметь много верхних границ, — не просто странность записи; он также проявляется при анализе времени выполнения. Встречались ситуации, в которых алгоритм имел доказанное время выполнения $O(n^3)$; проходили годы, и в результате более тщательного анализа того же алгоритма выяснялось, что в действительности время выполнения было равно $O(n^2)$. В первом результате не было ничего ошибочного; он определял правильную верхнюю границу. Просто эта оценка времени выполнения была недостаточно точной.

Асимптотические нижние границы

Для нижних границ тоже существует парное обозначение. Часто при анализе алгоритма (допустим, было доказано, что время выполнения для худшего случая $T(n)$ имеет порядок $O(n^2)$) бывает нужно показать, что эта верхняя граница является лучшей из возможных. Для этого следует выразить условие, что для входных данных произвольно большого размера n функция $T(n)$ *не меньше* произведения некой конкретной функции $f(n)$ на константу. (В данном примере в роли $f(n)$ оказывается n^2). Соответственно говорят, что $T(n)$ имеет нижнюю границу $\Omega(f(n))$ (также используется запись $T(n) = \Omega(f(n))$), если существуют такие константы $\epsilon > 0$ и $n_0 \geq 0$, что для всех $n \geq n_0$ выполняется условие $T(n) \geq \epsilon \cdot f(n)$. По аналогии с записью $O(\cdot)$ мы будем говорить, что T в таком случае имеет асимптотическую нижнюю границу f. И снова следует подчеркнуть, что константа ϵ должна быть фиксированной и не зависящей от n.

Это определение работает точно так же, как $O(\cdot)$, не считая того, что функция $T(n)$ ограничивается снизу, а не сверху. Например, возвращаясь к функции $T(n) = pn^2 + qn + r$, где p, q и r — положительные константы, будем утверждать, что $T(n) = \Omega(n^2)$. Если при установлении верхней границы приходилось «раздувать» функцию $T(n)$ до тех пор, пока она не начинала выглядеть как произведение n^2 на константу, на этот раз нужно сделать обратное: «подрезать» $T(n)$, пока она не начнет выглядеть как произведение n^2 на константу. Сделать это несложно; для всех $n \geq 0$

$$T(n) = pn^2 + qn + r \geq pn^2,$$

что соответствует требованию определения $\Omega(\cdot)$ с $\epsilon = p > 0$.

Для нижних границ существует то же понятие «точности» и «слабости», что и для верхних. Например, будет правильно утверждать, что функция $T(n) = pn^2 + qn + r$ имеет $\Omega(n)$, так как $T(n) \geq pn^2 \geq pn$.

Асимптотические точные границы

Если можно показать, что время выполнения $T(n)$ одновременно характеризуется $O(f(n))$ и $\Omega(f(n))$, то возникает естественное ощущение, что найдена «правильная» граница: $T(n)$ растет в точности как $f(n)$ в пределах постоянного коэффициента. Например, к такому выводу можно прийти из того факта, что $T(n) = pn^2 + qn + r$ одновременно имеет $O(n^2)$ и $\Omega(n^2)$.

Для выражения этого понятия тоже существует специальная запись: если функция $T(n)$ одновременно имеет $O(f(n))$ и $\Omega(f(n))$, говорят, что $T(n)$ имеет $\Theta(f(n))$, а $f(n)$ называется *асимптотически точной границей* для $T(n)$. Итак, например, приведенный выше анализ показывает, что $T(n) = pn^2 + qn + r$ имеет $\Theta(n^2)$.

Асимптотически точные границы для худшего времени выполнения полезны, поскольку они характеризуют быстродействие алгоритма в худшем случае с точностью до постоянного множителя. И, как следует из определения $\Theta(\cdot)$, такие границы могут быть получены сокращением промежутка между верхней и нижней границами. Например, иногда приходится читать (отчасти неформально выраженные) утверждения вида «Показано, что в худшем случае время выполнения алгоритма имеет верхнюю границу $O(n^3)$, однако не существует известных примеров, для которых алгоритм выполняется за более чем $\Omega(n^2)$ шагов». Такое утверждение можно считать неявным предложением поискать асимптотически точную границу для худшего времени выполнения алгоритма.

Иногда асимптотически точная граница может быть вычислена напрямую, как предел при стремлении n к бесконечности. Фактически если отношение функций $f(n)$ и $g(n)$ сходится к положительной константе, когда n уходит в бесконечность, то $f(n) = \Theta(g(n))$.

(2.1) Имеются две функции f и g, для которых

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

существует и равен некоторому числу $c > 0$. Тогда $f(n) = \Theta(g(n))$.

Доказательство. Мы воспользуемся тем фактом, что предел существует и он положителен, чтобы показать, что $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$, как того требует определение $\Omega(\cdot)$.

Поскольку

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0,$$

из определения предела следует, что существует некоторое значение n_0, начиная с которого отношение всегда лежит в диапазоне между $\frac{1}{2}c$ и $2c$. Таким образом, $f(n) \le 2cg(n)$ для всех $n \ge n_0$, из чего следует, что $f(n)=O(g(n))$; а $f(n) \ge \frac{1}{2}cg(n)$ для всех $n \ge n_0$, из чего следует, что $f(n) = \Omega(g(n))$. ■

Свойства асимптотических скоростей роста

После знакомства с определениями O, Ω и Θ будет полезно рассмотреть некоторые из их основных свойств.

Транзитивность

Первое свойство — транзитивность: если функция f асимптотически ограничена сверху функцией g, а функция g, в свою очередь, асимптотически ограничена сверху функцией h, то функция f асимптотически ограничена сверху функцией h. Аналогичное свойство действует и для нижних границ. Более точно это можно записать в следующем виде.

(2.2)
(a) Если $f = O(g)$ и $g = O(h)$, то $f = O(h)$.
(b) Если $f = \Omega(g)$ и $g = \Omega(h)$, то $f = \Omega(h)$.

Доказательство. Мы ограничимся частью (a) этого утверждения; часть (б) доказывается практически аналогично.

Для (a) известно, что для некоторых констант c и n_0 выполняется условие $f(n) \leq cg(n)$ для всех $n \geq n_0$. Кроме того, для некоторых (возможно, других) констант c' и n_0' выполняется условие $g(n) \leq c'h(n)$ для всех $n \geq n_0'$. Возьмем любое число n, не меньшее как n_0, так и n_0'. Известно, что $f(n) \leq cg(n) \leq cc'h(n)$, поэтому $f(n) \leq cc'h(n)$ для всех $n \geq \max(n_0, n_0')$. Из последнего неравенства следует, что $f = O(h)$. ■

Объединяя части (a) и (b) в (2.2), можно прийти к аналогичному результату для асимптотически точных границ. Допустим, известно, что $f = \Theta(g)$, а $g = \Theta(h)$. Так как $f = O(g)$ и $g = O(h)$, из части (a) известно, что $f = O(h)$; а поскольку $f = \Omega(g)$ и $g = \Omega(h)$, из части (б) следует, что $f = \Omega(h)$. Таким образом, $f = \Theta(h)$. Следовательно, мы показали, что

(2.3) Если $f = \Theta(g)$ и $g = \Theta(h)$, то $f = \Theta(h)$.

Суммы функций

Также полезно иметь количественную оценку эффекта суммирования двух функций. Прежде всего, если имеется асимптотическая верхняя оценка, применимая к каждой из двух функций f и g, то она применима и к сумме этих функций.

(2.4) Предположим, f и g — такие две функции, что для некоторой функции h выполняются условия $f = O(h)$ и $g = O(h)$. В этом случае $f + g = O(h)$.

Доказательство. Из постановки задачи следует, что для некоторых констант c и n_0 выполняется условие $f(n) \leq ch(n)$ для всех $n \geq n_0$. Кроме того, для некоторых (возможно, других) констант c' и n_0' выполняется условие $g(n) \leq c'h(n)$ для всех $n \geq n_0'$. Возьмем любое число n, не меньшее как n_0, так и n_0'. Известно, что $f(n) + g(n) \leq ch(n) + c'h(n)$. Из этого следует, что $f(n) + g(n) \leq (c + c')h(n)$ для всех $n \geq \max(n_0, n_0')$. Из последнего неравенства следует, что $f + g = O(h)$. ■

Это свойство обобщается для суммы фиксированного числа функций k, где k может быть больше двух. Точная формулировка результата приведена ниже;

доказательство не приводится, так как оно, по сути, повторяет доказательство (2.4) для сумм, состоящих из k слагаемых вместо 2.

(2.5) Пусть k — фиксированная константа, а $f_1, f_2, ..., f_k$ и h — функции, такие что $f_i = O(h)$ для всех i. В этом случае $f_1 + f_2 + ... + f_k = O(h)$.

У (2.4) имеется следствие, которое проявляется в типичной ситуации: во время анализа алгоритма, состоящего из двух высокоуровневых частей, часто удается легко показать, что одна из двух частей медленнее другой. В этом случае время выполнения всего алгоритма асимптотически сравнимо со временем выполнения медленной части. Поскольку общее время выполнения представляет собой сумму двух функций (время выполнения двух частей), полученные выше результаты для асимптотических границ сумм функций напрямую применимы в данном случае.

(2.6) Предположим, f и g — две функции (получающие неотрицательные значения) и $g = O(f)$. В этом случае $f + g = \Theta(f)$. Другими словами, f является асимптотически точной границей для объединенной функции $f + g$.

Доказательство. Очевидно, $f + g = \Omega(f)$, так как для всех $n \geq 0$ выполняется условие $f(n) + g(n) \geq f(n)$. Чтобы завершить доказательство, необходимо показать, что $f + g = O(f)$.

Но это утверждение является прямым следствием (2.4): дано, что $g = O(f)$, а условие $f = O(f)$ выполняется для любой функции, поэтому из (2.4) следует $f + g = O(f)$. ■

Данный результат также распространяется на сумму любого фиксированного количества функций: самая быстрорастущая из функций является асимптотически точной границей для суммы.

Асимптотические границы для некоторых распространенных функций

Некоторые функции постоянно встречаются при анализе алгоритмов, поэтому будет полезно изучить асимптотические свойства основных разновидностей: полиномиальных, логарифмических и экспоненциальных.

Полиномиальные функции

Напомним, что полиномиальной называется функция, которая может быть записана в форме $f(n) = a_0 + a_1 n + a_2 n^2 + ... + a_d n^d$ для некоторой целочисленной константы $d > 0$, где последний коэффициент a_d отличен от нуля. Значение d называется *степенью* (или порядком) полинома. Например, упоминавшиеся ранее функции вида $pn^2 + qn + r$ (где $p \neq 0$) представляют собой полиномиальные функции со степенью 2.

Основная особенность полиномиальных функций заключается в том, что их асимптотическая скорость роста детерминируется «членом высшего порядка» — то есть слагаемым, определяющим степень. Это утверждение более формально определяется в следующем утверждении. Так как нас интересуют только функции,

получающие неотрицательные значения, наше внимание будет ограничено полиномиальными функциями, у которых член высшего порядка имеет положительный коэффициент $a_d > 0$.

(2.7) Предположим, f является полиномиальной функцией степени d с положительным коэффициентом a_d. В этом случае $f = O(n^d)$.

Доказательство. Условие записывается в виде $f = a_0 + a_1 n + a_2 n^2 + ... + a_d n^d$, где $a_d > 0$. Верхняя граница напрямую следует из (2.5). Во-первых, следует заметить, что коэффициенты a_j для $j < d$ могут быть отрицательными, но в любом случае $a_j n^j \leq |a_j| n^d$ для всех $n \geq 1$. Следовательно, каждый член полинома имеет $O(n^d)$. Так как f представляет собой сумму фиксированного количества функций, каждая из которых имеет $O(n^d)$, из (2.5) следует, что $f = O(n^d)$. ■

Также можно показать, что по условиям (2.7) $f = \Omega(n^d)$, из чего следует, что фактически $f = \Theta(n^d)$.

Это свойство играет важную роль для рассмотрения отношения между асимптотическими границами такого рода и понятием полиномиального времени, к которому мы пришли в предыдущем разделе, для формализации более расплывчатого понятия эффективности. В записи $O(\cdot)$ можно легко дать определение полиномиального времени: *алгоритмом с полиномиальным временем* называется алгоритм, время выполнения которого $T(n) = O(n^d)$ для некоторой константы d, не зависящей от размера входных данных.

Таким образом, алгоритмы с границами времени выполнения вида $O(n^2)$ и $O(n^3)$ являются алгоритмами с полиномиальным временем. Однако важно понимать, что алгоритм может выполняться с полиномиальным временем даже в том случае, если время выполнения не записывается в форме n в целочисленной степени. Прежде всего, многие алгоритмы имеют время выполнения в форме $O(n^x)$ для некоторого числа x, которое не является целым. Например, в главе 5 представлен алгоритм с временем выполнения $O(n^{1,59})$; также встречаются экспоненты меньшие 1, как в границах типа $O(\sqrt{n}) = O(n^{1/2})$.

Другой распространенный пример: часто встречаются алгоритмы, время выполнения которых записывается в форме $O(n \log n)$. Такие алгоритмы также имеют полиномиальное время выполнения: как будет показано далее, $\log n \leq n$ для всех $n \geq 1$, а следовательно, $n \log n \leq n^2$ для всех $n \geq 1$. Другими словами, если алгоритм имеет время выполнения $O(n \log n)$, он также имеет время выполнения $O(n^2)$, а следовательно, является алгоритмом с полиномиальным временем.

Логарифмические функции

Напомним, что логарифм n по основанию b — $\log_b n$ — равен такому числу x, что $b^x = n$. Чтобы получить примерное представление о скорости роста $\log_b n$, можно заметить, что при округлении вниз до ближайшего целого числа логарифм на 1 меньше количества цифр в представлении n по основанию b (таким образом, например, значение $1 + \log_2 n$ с округлением вниз определяет количество битов, необходимых для представления n).

Итак, логарифмические функции растут очень медленно. В частности, для каждого основания b функция $\log_b n$ асимптотически ограничивается всеми функциями вида n^x, даже для (дробных) значений x, сколь угодно близких к 0.

(2.8) Для всех $b > 1$ и всех $x > 0$ выполняется условие $\log_b n = O(n^x)$.

Логарифмы могут напрямую преобразовываться к другому основанию по следующей фундаментальной формуле:

$$\log_a n = \frac{\log_b n}{\log_b a}.$$

Эта формула объясняет, почему границы часто записываются в виде $O(\log n)$ без указания основания логарифма. Не стоит рассматривать это как неточность записи: из приведенной формулы следует, что $\log_a n = \frac{1}{\log_b a} \cdot \log_b n$, а это означает, что $\log_a n = \Theta(\log_b n)$. Следовательно, при использовании асимптотической записи основание алгоритма роли не играет.

Экспоненциальные функции

Экспоненциальные функции записываются в форме $f(n) = r^n$ для некоторого постоянного основания r. Здесь нас интересует случай $r > 1$, что приводит к исключительно быстрому росту функции.

Если полиномиальная функция возводит n в фиксированную степень, экспоненциальная функция возводит фиксированное число в степень n; скорость роста при этом сильно увеличивается. Один из способов описания отношений между полиномиальными и экспоненциальными функциями представлен ниже.

(2.9) Для всех $r > 1$ и всех $d > 0$ выполняется условие $n^d = O(r^n)$.

В частности, любая экспоненциальная функция растет быстрее любой полиномиальной. И как было показано в табл. 2.1, при подстановке реальных значений n различия в скорости роста становятся весьма впечатляющими.

По аналогии с записью $O(\log n)$ без указания основания часто встречаются формулировки вида «Алгоритм имеет экспоненциальное время выполнения» без указания конкретной экспоненциальной функции. В отличие от свободного использования $\log n$, оправданного игнорированием постоянных множителей, такое широкое использование термина «экспоненциальный» несколько неточно. В частности, для разных оснований $r > s > 1$ никогда не выполняется условие $r^n = \Theta(s^n)$. В самом деле, для этого бы потребовалось, чтобы для некоторой константы $c > 0$ выполнялось условие $r^n \le cs^n$ для всех достаточно больших n. Но преобразование этого неравенства дает $(r/s)^n \le c$ для всех достаточно больших n. Так как $r > s$, выражение $(r/s)^n$ стремится к бесконечности с ростом n, поэтому оно не может ограничиваться константой c.

Итак, с асимптотической точки зрения все экспоненциальные функции различны. Тем не менее обычно смысл неточной формулировки «Алгоритм имеет экспоненциальное время выполнения» понятен — имеется в виду, что время выполнения

растет по крайней мере со скоростью некоторой экспоненциальной функции, а все экспоненциальные функции растут так быстро, что алгоритм можно попросту отбросить, не вдаваясь в подробности относительно точного времени выполнения. Это не всегда оправданно — в некоторых случаях за экспоненциальными алгоритмами скрывается больше, чем видно на первый взгляд (как мы увидим, например, в главе 10); но как указано в первой части этой главы, это разумное эмпирическое правило.

Итак, логарифмические, полиномиальные и экспоненциальные функции служат полезными ориентирами в диапазоне функций, встречающихся при анализе времени выполнения. Логарифмические функции растут медленнее полиномиальных, а полиномиальные растут медленнее экспоненциальных.

2.3. Реализация алгоритма устойчивых паросочетаний со списками и массивами

Мы рассмотрели общий подход к выражению границ времени выполнения алгоритмов. Чтобы проанализировать асимптотическое время выполнения алгоритмов, выраженных на высоком уровне (например, алгоритма поиска устойчивых паросочетаний Гейла–Шепли в главе 1), следует не писать код, компилировать и выполнять его, а думать над представлением и способом обработки данных в алгоритме, чтобы получить оценку количества выполняемых вычислительных действий.

Вероятно, у вас уже имеется некоторый опыт реализации базовых алгоритмов со структурами данных. В этой книге структуры данных будут рассматриваться в контексте реализации конкретных алгоритмов, поэтому в ней вы встретите разные структуры данных в зависимости от потребностей разрабатываемого алгоритма. Для начала мы рассмотрим реализацию алгоритма устойчивых паросочетаний Гейла–Шепли; ранее было показано, что алгоритм завершается максимум за n^2 итераций, а наша реализация обеспечивает соответствующее худшее время выполнения $O(n^2)$ с подсчетом фактических вычислительных шагов вместо простого количества итераций. Чтобы получить такую границу для алгоритма устойчивых паросочетаний, достаточно использовать две простейшие структуры данных: *списки* и *массивы*. Таким образом, наша реализация также предоставит хорошую возможность познакомиться с использованием этих базовых структур данных.

В задаче устойчивых паросочетаний у каждого мужчины и каждой женщины должны существовать оценки всех членов противоположного пола. Самый первый вопрос, который необходимо обсудить, — как будут представляться эти оценки? Кроме того, алгоритм ведет учет паросочетаний, поэтому на каждом шаге он должен знать, какие мужчины и женщины свободны и кто с кем находится в паре. Для реализации алгоритма необходимо решить, какие структуры данных будут использоваться для всех этих задач.

Следует учесть, что структура данных выбирается проектировщиком алгоритма; для каждого алгоритма мы будем выбирать структуры данных, обеспечи-

вающие эффективную и простую реализацию. В некоторых случаях это может потребовать *предварительной обработки* входных данных и их преобразования из заданного входного представления в структуру, лучше подходящую для решаемой задачи.

Массивы и списки

Для начала сосредоточимся на одном списке — например, списке женщин в порядке предпочтений некоторого мужчины. Возможно, для хранения списка из n элементов проще всего воспользоваться массивом A длины n, в котором $A[i]$ содержит i-й элемент списка. Такой массив легко реализуется практически на любом стандартном языке программирования и обладает следующими свойствами.

- Ответ на запрос вида «Какое значение хранится в i-м элементе списка?» можно получить за время $O(1)$, напрямую обратившись к значению $A[i]$.
- Если мы хотим определить, входит ли в список некоторый элемент e (то есть равен ли он $A[i]$ для некоторого i), необходимо последовательно проверить все элементы за время $O(n)$ — при условии, что нам ничего не известно о порядке следования элементов в A.
- Если элементы массива отсортированы в четко определенном порядке (например, по числовым значениям или по алфавиту), тогда принадлежность элемента e списку проверяется за время $O(\log n)$ методом *бинарного поиска*; в алгоритме устойчивых паросочетаний бинарный поиск не используется, но мы вернемся к нему в следующем разделе.

Массив не так хорошо подходит для динамического ведения списка элементов, изменяющихся со временем (например, списка свободных мужчин в алгоритме устойчивых паросочетаний); поскольку мужчины переходят из свободного состояния в состояние помолвки (а возможно, и обратно), список свободных мужчин должен увеличиваться и уменьшаться в процессе выполнения алгоритма. Обычно частое добавление и удаление элементов из списка, реализованного на базе массива, реализуется громоздко и неудобно.

Для ведения таких динамических наборов элементов можно (а часто и желательно) использовать связанные списки. В связанном списке для формирования последовательности элементов каждый элемент указывает на следующий элемент списка. Таким образом, в каждом элемента v должен храниться указатель на следующий элемент; если это последний элемент, указателю присваивается *null*. Также существует указатель `First`, ссылающийся на первый элемент. Начиная с `First` и последовательно переходя по указателям на следующий элемент, пока не будет обнаружен указатель *null*, можно перебрать все содержимое списка за время, пропорциональное его длине.

Обобщенный способ реализации связанного списка, в котором набор возможных элементов не может быть зафиксирован изначально, основан на создании записи e для каждого элемента, включаемого в список. Такая запись содержит поле e.`val`, в котором хранится значение элемента, и поле e.`Next` с указателем на сле-

дующий элемент списка. Также можно создать *двусвязный список*, поддерживающий переходы в обоих направлениях; для этого добавляется поле *e*.Prev со ссылкой на предыдущий элемент списка. (Для первого элемента в списке *e*.Prev = *null*.) Также добавляется указатель Last — аналог First, указывающий на последний элемент в списке. Схематическое изображение части такого списка представлено в верхней части рис. 2.1.

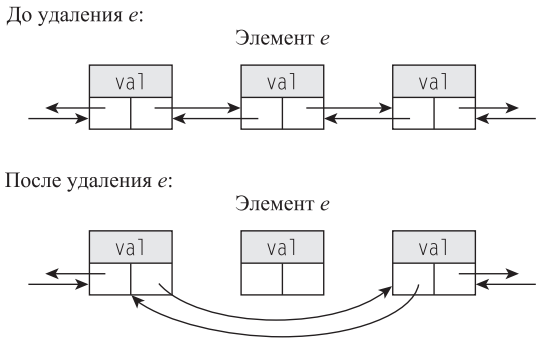


Рис. 2.1. Схематическое представление двусвязного списка с удалением элемента *e*

С двусвязным списком могут выполняться следующие операции.

- *Удаление*. Чтобы удалить элемент *e* из двусвязного списка, достаточно «обойти» его так, чтобы предыдущий элемент (на который указывает *e*.Prev) и следующий элемент (на который указывает *e*.Next) указывали друг на друга. Операция удаления изображена на рис. 2.1.
- *Вставка*. Чтобы вставить элемент *e* между элементами *d* и *f* в списке, мы присваиваем *d*.Next и *f*.Prev указатель на *e*, а полям Next и Prev в *e* — указатели на *d* и *f* соответственно. Эта операция, по сути, является обратной по отношению к удалению; чтобы понять происходящее, достаточно просмотреть рис. 2.1 снизу вверх.

При вставке или удалении *e* в начале списка обновляется указатель First — вместо обновления записи элемента, предшествующего *e*.

Списки хорошо подходят для ведения динамически изменяемых множеств, но у них имеются свои недостатки. В отличие от массивов, *i*-й элемент списка невозможно найти за время $O(1)$: переход к *i*-му элементу потребует перехода по указателям Next, начиная от начала списка, что займет в общей сложности время $O(i)$.

С учетом относительных достоинств и недостатков массивов и списков может оказаться так, что мы получим входные данные задачи в одном из двух форматов и захотим преобразовать их к другому формату. Как упоминалось ранее, такая предварительная обработка часто приносит пользу; и в таком случае преобразование между массивами и списками легко выполняется за время $O(n)$. Это позволит нам свободно выбрать структуру данных, которая лучше подходит для конкретного алгоритма, не привязываясь к той структуре, в которой были получены входные данные.

Реализация алгоритма устойчивых паросочетаний

Воспользуемся массивами и связанными списками для реализации алгоритма устойчивых паросочетаний из главы 1. Ранее уже было показано, что алгоритм завершается максимум за n^2 итераций, и это значение дает своего рода верхнюю оценку для времени выполнения. Но чтобы реализовать алгоритм Гейла–Шепли так, чтобы он действительно выполнялся за время, пропорциональное n^2, каждая итерация должна выполняться за постоянное время. Сейчас мы поговорим о том, как это сделать.

Для простоты будем считать, что элементы множеств мужчин и женщин пронумерованы $\{1, ..., n\}$. Для этого можно упорядочить мужчин и женщин (скажем, по алфавиту) и связать число i с i-м мужчиной m_i и i-й женщиной w_i в этом порядке. Такое предположение (или система обозначений) позволяет определить массив, индексированный по всем мужчинам или всем женщинам. Нам нужно создать список предпочтений для каждого мужчины и каждой женщины. Для этого понадобятся два массива: в одном будут храниться предпочтения женщин, а в другом предпочтения мужчин. Мы будем использовать запись `ManPref`$[m, i]$ для обозначения i-й женщины в списке предпочтений мужчины m и аналогичную запись `WomanPref`$[w, i]$ для обозначения i-го мужчины в списке предпочтений женщины w. Обратите внимание: для хранения предпочтений всех $2n$ людей понадобится пространство $O(n^2)$, так как для каждого человека создается список длины n.

Мы должны проанализировать каждый шаг алгоритма и понять, какая структура данных позволит эффективно реализовать его. Фактически необходимо, чтобы каждая из следующих четырех операций выполнялась за постоянное время.

1. Найти свободного мужчину.
2. Для мужчины m — найти женщину с наивысшей оценкой, которой он еще не делал предложение.
3. Для женщины w — проверить, находится ли w в состоянии помолвки, и если находится, найти ее текущего партнера.
4. Для женщины w и двух мужчин m и m' — определить (также за постоянное время), кто из m и m' является предпочтительным для w.

Начнем с выбора свободного мужчины. Для этого мы организуем множество свободных мужчин в связанный список. Когда потребуется выбрать свободного мужчину, достаточно взять первого мужчину m в этом списке. Если m переходит в состояние помолвки, он удаляется из списка, возможно, со вставкой другого мужчины m', если тот переходит в свободное состояние. В таком случае m' вставляется в начало списка — также за постоянное время.

Возьмем мужчину m. Требуется выявить женщину с наивысшей оценкой, которой он еще не делал предложение. Для этого понадобится дополнительный массив `Next`, в котором для каждого мужчины m хранится позиция следующей женщины, которой он сделает предложение, в его списке. Массив инициализируется `Next`$[m] = 1$ для всех мужчин m. Если мужчина m должен сделать предложение,

он делает его женщине w = ManPref[m,Next[m]], а после предложения w значение Next[m] увеличивается на 1 независимо от того, приняла w его предложение или нет.

Теперь предположим, что мужчина m делает предложение женщине w; необходимо иметь возможность определить мужчину m', с которым помолвлена w (если он есть). Для этого будет создан массив Current длины n, где Current[w] — текущий партнер m' женщины w. Если мы хотим обозначить, что женщина w в настоящее время не помолвлена, Current[w] присваивается специальное обозначение *null*; в начале алгоритма Current[w] инициализируется *null* для всех w.

Итак, структуры данных, которые мы определили до настоящего момента, способны реализовать каждую из операций (1)–(3) за время $O(1)$.

Пожалуй, сложнее всего будет выбрать способ хранения предпочтений женщин, обеспечивающий эффективную реализацию шага (4). Рассмотрим шаг алгоритма, на котором мужчина m делает предложение женщине w. Предположим, w уже помолвлена и ее текущим партнером является m' = Current[w]. Нам хотелось бы за время $O(1)$ решить, кто, с точки зрения w, является более предпочтительным — m или m'. Хранение предпочтений женщин в массиве WomanPref (по аналогии с тем, как это делалось для мужчин) не работает, так как нам придется последовательно перебирать элементы списка w; поиск m и m' в списке займет время $O(n)$. И хотя время $O(n)$ является полиномиальным, можно получить намного лучший результат, построив вспомогательную структуру данных.

В начале алгоритма мы создадим массив Ranking с размерами $n \times n$, в котором Ranking[w, m] содержит оценку мужчины m в отсортированном порядке предпочтений женщины w. Для каждой женщины w этот массив создается за линейное время при одном проходе по ее списку предпочтений; таким образом, общие затраты времени пропорциональны n^2. После этого, чтобы решить, кто из мужчин — m или m' — является предпочтительным для w, достаточно сравнить значения Ranking[w, m] и Ranking[w, m'].

Это позволяет выполнить шаг (4) за постоянное время, а следовательно, у нас появляется все необходимое для обеспечения желаемого времени выполнения.

(2.10) Структуры данных, описанные выше, позволяют реализовать алгоритм Гейла–Шепли за время $O(n^2)$.

2.4. Обзор типичных вариантов времени выполнения

Когда вы пытаетесь проанализировать новый алгоритм, полезно иметь хотя бы приблизительное представление о «спектре» разных вариантов времени выполнения. В самом деле, некоторые типы анализа встречаются часто, и когда вы сталкиваетесь с распространенными границами времени выполнения вида $O(n)$, $O(n \log n)$ или $O(n^2)$, часто это объясняется одной из немногочисленных, но хорошо узнаваемых причин. Для начала мы приведем сводку типичных границ времени выполнения, а также типичные аналитические методы, приводящие к ним.

Ранее мы уже говорили о том, что у многих задач существует естественное «пространство поиска» — совокупность всех возможных решений, а также упоминали, что объединяющей темой в проектировании алгоритмов является поиск алгоритмов, превосходящих по эффективности перебор пространства поиска методом «грубой силы». Соответственно, когда вы беретесь за новую задачу, часто бывает полезно рассматривать два вида границ: для времени выполнения, которое вы рассчитываете достичь, и для размера естественного пространства поиска задачи (а следовательно, для времени выполнения алгоритма методом «грубой силы»). Обсуждение вариантов времени выполнения в этом разделе часто будет начинаться с анализа алгоритма «грубой силы», который поможет разобраться в особенностях задачи. Большая часть книги будет посвящена тому, как добиться повышения быстродействия по сравнению с такими алгоритмами.

Линейное время

Алгоритм с временем $O(n)$, или *линейным* временем, обладает одним очень естественным свойством: его время выполнения не превышает размера входных данных умноженного на константу. Чтобы обеспечить линейное время выполнения, проще всего обработать входные данные за один проход, с постоянными затратами времени на обработку каждого элемента. Другие алгоритмы достигают линейной границы времени выполнения по менее очевидным причинам. Чтобы дать представление о некоторых принципах анализа, мы рассмотрим два простых алгоритма с линейным временем выполнения.

Вычисление максимума

Вычисление максимума из n чисел может быть выполнено классическим «однопроходным» методом. Допустим, числа поступают на вход в виде списка или массива. Числа a_1, a_2,... обрабатываются последовательно, с сохранением текущего максимума во время выполнения. Каждое очередное число a_i сравнивается с текущим максимумом, и если оно больше, текущий максимум заменяется значением a_i.

```
Max = a1
For i = 2 to n
   Если ai > max
      Присвоить max = ai
   Конец Если
Конец For
```

При таком решении для каждого элемента выполняется постоянный объем работы, а общее время выполнения составляет $O(n)$.

Иногда необходимость применения однопроходных алгоритмов объясняется ограничениями приложения: например, алгоритм, работающий на высокоскоростном коммутаторе в Интернете, «видит» проходящий мимо него поток пакетов и может выполнять любые необходимые вычисления с проходящими пакетами. При этом для каждого пакета выполняется ограниченный объем вычислительной

работы, и поток не может сохраняться для перебора в будущем. Для изучения этой модели вычислений были разработаны две специализированные подобласти: *онлайновые алгоритмы* и *алгоритмы потоков данных*.

Слияние двух отсортированных списков

Часто алгоритмы обладают временем выполнения $O(n)$ по более сложным причинам. Сейчас будет рассмотрен алгоритм слияния двух отсортированных списков, который немного выходит за рамки «однопроходной» схемы, но при этом все равно обладает линейным временем.

Допустим, имеются два списка, каждый из которых состоит из n чисел: a_1, a_2, ..., a_n и b_1, b_2, ..., b_n. Оба списка уже отсортированы по возрастанию. Требуется объединить их в один список c_1, c_2, ..., c_{2n}, который также упорядочен по возрастанию. Например, при слиянии списков 2, 3, 11, 19 и 4, 9, 16, 25 должен быть получен список 2, 3, 4, 9, 11, 16, 19, 25.

Для этого можно было бы просто свалить оба списка в одну кучу, игнорируя тот факт, что они уже упорядочены по возрастанию, и запустить алгоритм сортировки. Но такое решение явно неэффективно: нужно использовать существующий порядок элементов во входных данных. Чтобы спроектировать более совершенный алгоритм, подумайте, как бы вы выполняли слияние двух списков вручную. Допустим, имеются две стопки карточек с числами, уже разложенные по возрастанию, и вы хотите получить одну стопку со всеми карточками. Взглянув на верхнюю карточку каждой стопки, вы знаете, что карточка с меньшим числом должна предшествовать второй в выходной стопке; вы снимаете карточку, кладете ее в выходную стопку и повторяете процедуру для полученных стопок.

Иначе говоря, имеется следующий алгоритм:

```
Объединение отсортированных списков A = a_1, ..., a_n и B = b_1, ..., b_n:
   Создать для каждого списка текущий указатель Current, который инициализируется
указателем на начальные элементы
   Пока оба списка не пусты:
      Присвоить a_i и b_j элементы, на которые указывают текущие указатели
      Присоединить меньший из этих двух элементов к выходному списку
      Переместить текущий указатель в списке, из которого был выбран меньший
элемент
   Конец Пока
   Когда один из списков окажется пустым, присоединить остаток другого списка
к выходному списку
```

Схема работы этого алгоритма изображена на рис. 2.2.

Утверждение о линейной границе времени выполнения хотелось бы подтвердить тем же аргументом, который использовался для алгоритма поиска максимума: «Для каждого элемента выполняется постоянный объем работы, поэтому общее время выполнения составляет $O(n)$». Но на самом деле нельзя утверждать, что для каждого элемента выполняется постоянная работа. Допустим, n — четное число; рассмотрим списки $A = 1, 3, 5, ..., 2n - 1$ и $B = n, n + 2, n + 4, ..., 3n - 2$. Число b_1 в начале списка B будет находиться в начале списка для $n/2$ итераций при повторном

выборе элементов из A, а следовательно, будет участвовать в $\Omega(n)$ сравнениях. Каждый элемент может быть задействован максимум в $O(n)$ сравнениях (в худшем случае он сравнивается с каждым элементом другого списка), и при суммировании по всем элементам будет получена граница времени выполнения $O(n^2)$. Эта граница верна, но можно существенно усилить ее точность.

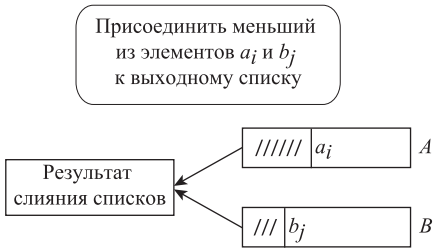


Рис. 2.2. Чтобы выполнить слияние двух списков, мы в цикле извлекаем меньший из начальных элементов двух списков и присоединяем его к выходному списку

Самый наглядный аргумент — ограничение количества итераций цикла «Пока» по «учетной» схеме. Предположим, для каждого элемента при выборе и добавлении его в выходной список взимается оплата. За каждый элемент платить придется только один раз, так как оплаченный элемент добавляется в выходной список и уже никогда не рассматривается алгоритмом. Всего существуют $2n$ элементов, и каждая итерация оплачивается некоторым элементом, поэтому количество итераций не может быть больше $2n$. В каждой итерации используется постоянный объем работы, поэтому общее время выполнения имеет границу $O(n)$, как и требовалось.

Хотя алгоритм слияния перебирает свои входные списки по порядку, «чередование» последовательности обработки списков требует применения чуть более сложного анализа времени выполнения. В главе 3 будут представлены алгоритмы с линейным временем для графов, которые требуют еще более сложного процесса управления: для каждого узла и ребра графа тратится постоянное время, но порядок обработки узлов и ребер зависит от структуры графа.

Время $O(n \log n)$

Время $O(n \log n)$ тоже встречается очень часто, и в главе 5 будет представлена одна из главных причин его распространенности: это время выполнения любого алгоритма, который разбивает входные данные на две части одинакового размера, рекурсивно обрабатывает каждую часть, а затем объединяет два решения за линейное время.

Пожалуй, классическим примером задачи, которая может решаться подобным образом, является сортировка. Так, алгоритм *сортировки слиянием* разбивает входной набор чисел на две части одинакового размера, рекурсивно сортирует каждую часть, а затем объединяет две отсортированные половины в один отсортированный выходной список. Только что было показано, что слияние может быть выполнено

за линейное время; в главе 5 мы обсудим анализ рекурсии для получения границы $O(n \log n)$ в общем времени выполнения.

Время $O(n \log n)$ также часто встречается просто из-за того, что во многих алгоритмах самым затратным шагом является сортировка входных данных. Допустим, имеется набор из n временных меток $x_1, x_2, ..., x_n$ поступления копий файлов на сервер, и нам хотелось бы определить наибольший интервал между первой и последней из этих временных меток, в течение которого не поступило ни одной копии файла. Простейшее решение этой задачи — отсортировать временные метки $x_1, x_2, ..., x_n$, а затем обработать их в порядке сортировки и вычислить интервалы между каждыми двумя соседними числами. Наибольший из этих интервалов дает желаемый результат. Этот алгоритм требует времени $O(n \log n)$ на сортировку чисел, с последующим постоянным объемом работы для каждого числа в порядке перебора. Другими словами, после сортировки вся оставшаяся работа следует основному сценарию линейного времени, который рассматривался ранее.

Квадратичное время

Типичная задача: заданы n точек на плоскости, определяемых координатами (x, y); требуется найти пару точек, расположенных ближе всего друг к другу. Естественный алгоритм «грубой силы» для такой задачи перебирает все пары точек, вычисляет расстояния между каждой парой, а затем выбирает пару, для которой это расстояние окажется наименьшим.

Какое время выполнения обеспечит такой алгоритм? Количество пар точек равно $\binom{n}{2} = \frac{n(n-1)}{2}$, а поскольку эта величина ограничивается $\frac{1}{2}n^2$, она имеет границу $O(n^2)$. Проще говоря, количество пар ограничивается $O(n^2)$ потому, что количество способов выбора первого элемента пары (не больше n) умножается на количество способов выбора второго элемента пары (тоже не больше n). Расстояние между точками (x_i, y_i) и (x_j, y_j) вычисляется по формуле $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ за постоянное время, так что общее время выполнения ограничивается $O(n^2)$.

В этом примере представлена очень распространенная ситуация, в которой встречается время выполнения $O(n^2)$: перебор всех пар входных элементов с постоянными затратами времени на каждую пару.

Квадратичное время так же естественно возникает в парах вложенных циклов: алгоритм состоит из цикла с $O(n)$ итерациями, и каждая итерация цикла запускает внутренний цикл с временем $O(n)$. Произведение двух множителей n дает искомое время выполнения.

Эквивалентная запись алгоритм поиска ближайшей пары точек методом «грубой силы» с использованием двух вложенных циклов выглядит так:

```
Для каждой входной точки (x_i, y_i)
   Для каждой из оставшихся входных точек (x_j, y_j)
      Вычислить расстояние d = sqrt((x_i - x_j)^2 + (y_i - y_j)^2)
```

```
      Если d меньше текущего минимума, заменить минимум значением d
   Конец цикла
Конец цикла
```

Обратите внимание: «внутренний» цикл по (x_j, y_j) состоит из $O(n)$ итераций, каждая из которых выполняется за постоянное время, и «внешний» цикл по (x_i, y_i) состоит из $O(n)$ итераций, каждая из которых один раз запускает внутренний цикл.

Важно заметить, что алгоритм, рассматриваемый нами для задачи ближайшей пары точек, в действительности относится к методам «грубой силы»: естественное пространство поиска этой задачи имеет размер $O(n^2)$, и мы всего лишь перебираем его. На первый взгляд может показаться, что в квадратичном времени есть некая неизбежность — ведь нам все равно придется перебрать все расстояния, не так ли? — но в действительности это всего лишь иллюзия. В главе 5 будет описан очень остроумный алгоритм для поиска ближайшей пары точек на плоскости за время всего лишь $O(n \log n)$, а в главе 13 мы покажем, как применение рандомизации сокращает время выполнения до $O(n)$.

Кубическое время

Более сложные группы вложенных циклов часто приводят к появлению алгоритмов, выполняемых за время $O(n^3)$. Для примера рассмотрим следующую задачу. Имеются множества $S_1, S_2, ..., S_n$, каждое из которых является подмножеством $\{1, 2, ..., n\}$; требуется узнать, можно ли найти среди этих множеств непересекающуюся пару (то есть два множества, не имеющих общих элементов).

Какое время потребуется для решения этой задачи? Предположим, каждое множество S_i представлено таким образом, что элементы S_i могут быть обработаны с постоянными затратами времени на элемент, а проверка того, входит ли заданное число p в S_i, также выполняется за постоянное время. Ниже описано наиболее прямолинейное решение задачи.

```
Для всех пар множеств Si и Sj
   Определить, имеют ли Si и Sj общий элемент
Конец цикла
```

Алгоритм выглядит конкретно, но чтобы оценить его время выполнения, полезно преобразовать его (по крайней мере концептуально) в три вложенных цикла.

```
Для каждого множества Si
   Для каждого из оставшихся множеств Sj
      Для каждого элемента p множества Si
         Определить, принадлежит ли p множеству Sj
      Конец цикла
      Если ни один элемент Si не принадлежит Sj
         Сообщить, что множества Si и Sj являются непересекающимися
      Конец Если
   Конец цикла
Конец цикла
```

Каждое из множеств имеет максимальный размер n, поэтому внутренний цикл выполняется за время $O(n)$. Перебор множеств S_j требует $O(n)$ итераций внутреннего цикла, а перебор множеств S_i требует $O(n)$ итераций внешнего цикла. Перемножая эти три множителя, мы получаем время выполнения $O(n^3)$.

Для этой задачи существуют алгоритмы, улучшающие время выполнения $O(n^3)$, но они достаточно сложны. Более того, не очевидно, дают ли эти улучшенные алгоритмы практический выигрыш при входных данных разумного размера.

Время $O(n^k)$

По тем же соображениям, по которым мы пришли к времени выполнения $O(n^2)$, применяя алгоритм «грубой силы» ко всем парам множества из n элементов, можно получить время выполнения $O(n^k)$ для любой константы k при переборе всех подмножеств размера k.

Для примера возьмем задачу поиска независимого множества в графе, упоминавшуюся в главе 1. Напомним, что множество узлов называется *независимым*, если никакие два узла не соединяются ребром. А теперь допустим, что для некоторой константы k мы хотим узнать, содержит ли входной граф G с n узлами независимое множество размера k. Естественный алгоритм «грубой силы» для этой задачи перебирает все подмножества из k узлов, и для каждого подмножества S проверяет существование ребра, соединяющего любые два узла из S. На псевдокоде это выглядит так:

```
Для каждого подмножества S из k узлов
   Проверить, образует ли S независимое множество
   Если S является независимым множеством
      Остановиться и объявить об успешном поиске
   Конец Если
Конец цикла
Если не найдено ни одно независимое множество из k узлов
   Объявить о неудачном поиске
Конец Если
```

Чтобы понять время выполнения этого алгоритма, необходимо учесть два фактора. Во-первых, общее количество k-элементных подмножеств в множестве из n элементов равно

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \le \frac{n^k}{k!}.$$

Так как k считается постоянной величиной, эта величина имеет порядок $O(n^k)$. Следовательно, внешний цикл алгоритма, перебирающий все k-элементные подмножества n узлов графа, выполняется за $O(n^k)$ итераций.

Внутри цикла необходимо проверить, образует ли заданное множество S из k узлов независимое множество. Согласно определению независимого множества, необходимо для каждой пары узлов проверить, соединяются ли они ребром. Это типичная задача поиска по парам, которая, как было показано ранее при обсужде-

нии квадратичного времени выполнения, требует проверки $\binom{k}{2}$, то есть $O(k^2)$ пар при постоянных затратах времени на каждую пару.

Итак, общее время выполнения равно $O(k^2n^k)$. Так как k считается константой, а константы в записи $O(\cdot)$ могут опускаться, это время выполнения можно записать в виде $O(n^k)$.

Поиск независимых множеств — типичный пример задачи, которая считается вычислительно сложной. В частности, считается, что никакой алгоритм поиска независимого множества из k узлов в произвольных графах не может избежать зависимости от k в показателе степени. Но, как будет показано в главе 10 в контексте похожей задачи, даже если согласиться с тем, что поиск методом «грубой силы» по k-элементным подмножествам неизбежен, существуют разные варианты реализации, приводящие к существенным различиям в эффективности вычислений.

За границами полиномиального времени

Задача поиска независимого множества подводит нас к границе области времен выполнения, растущих быстрее любого полиномиального времени. В частности, на практике очень часто встречаются границы 2^n и $n!$; сейчас мы обсудим, почему это происходит.

Предположим, для некоторого графа нужно найти независимое множество *максимального размера* (вместо того, чтобы проверять сам факт существования при заданном количестве узлов). До настоящего времени не известны никакие алгоритмы, обеспечивающие заметный выигрыш по сравнению с поиском методом «грубой силы», который в данном случае выглядит так:

```
Для каждого подмножества узлов S
   Проверить, образует ли S независимое множество
   Если S является независимым множеством, размер которого превышает
                  размер наибольшего из найденных ранее
      Сохранить размер S как текущий максимум
   Конец Если
Конец цикла
```

Этот алгоритм очень похож на алгоритм «грубой силы» для независимых множеств из k узлов, однако на этот раз перебор ведется по всем подмножествам графа. Общее количество подмножеств n-элементного множества равно 2^n, поэтому внешний цикл алгоритма будет выполнен в 2^n итерациях при переборе всех этих подмножеств. Внутри цикла перебираются все пары из множества S, которое может содержать до n узлов, так что каждая итерация цикла выполняется за максимальное время $O(n^2)$. Перемножая эти два значения, мы получаем время выполнения $O(n^2 2^n)$.

Как видите, граница 2^n естественным образом возникает в алгоритмах поиска, которые должны рассматривать все возможные подмножества. Похоже, в задаче поиска независимого множества такая (или по крайней мере близкая) неэффективность неизбежна; но важно помнить, что пространство поиска с размером 2^n встре-

чается во многих задачах, и достаточно часто удается найти высокоэффективный алгоритм с полиномиальным временем выполнения. Например, алгоритм поиска методом «грубой силы» для задачи интервального планирования, упоминавшейся в главе 1, очень похож на представленный выше: мы опробуем все подмножества интервалов и находим наибольшее подмножество без перекрытий. Однако в задаче интервального планирования, в отличие от задачи поиска независимого множества, оптимальное решение может быть найдено за время $O(n \log n)$ (см. главу 4). Такого рода дихотомия часто встречается при изучении алгоритмов: два алгоритма имеют внешне похожие пространства поиска, но в одном случае алгоритм поиска методом «грубой силы» удается обойти, а в другом нет.

Функция $n!$ растет еще быстрее, чем 2^n, поэтому в качестве границы производительности алгоритма она выглядит еще более угрожающе. Пространства поиска размера $n!$ обычно встречаются по одной из двух причин. Во-первых, $n!$ — количество комбинаций из n элементов с n другими элементами, например количество возможных идеальных паросочетаний n мужчин с n женщинами в задаче поиска устойчивых паросочетаний. У первого мужчины может быть n вариантов пары; после исключения выбранного варианта для второго мужчины остается $n - 1$ вариантов; после исключения этих двух вариантов для третьего остается $n - 2$ вариантов, и т. д. Перемножая все эти количества вариантов, получаем $n(n - 1)\ (n - 2)\ ...\ (2)(1) = n!$

Несмотря на огромное количество возможных решений, нам удалось решить задачу устойчивых паросочетаний за $O(n^2)$ итераций алгоритма с предложениями. В главе 7 аналогичное явление встретится нам в контексте задачи двудольных паросочетаний, о которой говорилось ранее; если на каждой стороне двудольного графа находятся n узлов, существуют $n!$ возможных комбинаций. Однако достаточно хитроумный алгоритм поиска позволит найти наибольшее двудольное паросочетание за время $O(n^3)$.

Функция $n!$ также встречается в задачах, в которых пространство поиска состоит из всех способов упорядочения n элементов. Типичным представителем этого направления является *задача коммивояжера*: имеется множество из n городов и расстояния между всеми парами, как выглядит самый короткий маршрут с посещением всех городов? Предполагается, что коммивояжер начинает и заканчивает поездку в первом городе, так что суть задачи сводится к неявному поиску по всем вариантам упорядочения остальных $n - 1$ городов, что ведет к пространству поиска размером $(n - 1)!$. В главе 8 будет показано, что задача коммивояжера, как и задача независимых множеств, относится к классу *NP*-полных задач; как предполагается, она не имеет эффективного решения.

Сублинейное время

Наконец, в некоторых ситуациях встречается время выполнения, асимптотически меньшее линейного. Так как для простого чтения входных данных потребуется линейное время, такие ситуации обычно встречаются в модели вычислений с косвенными «проверками» входных данных вместо их полного чтения, а целью является минимизация количества необходимых проверок.

Пожалуй, самым известным примером такого рода является алгоритм бинарного поиска. Имеется отсортированный массив A из n чисел; требуется определить, принадлежит ли массиву заданное число p. Конечно, задачу можно решить чтением всего массива, но нам хотелось бы действовать намного более эффективно — проверять отдельные элементы, используя факт сортировки массива. Алгоритм проверяет средний элемент A и получает его значение (допустим, q), после чего q сравнивается с p. Если $q = p$, поиск завершается. Если $q > p$, то значение p может находиться только в нижней половине A; с этого момента верхняя половина A игнорируется, а процедура поиска рекурсивно применяется к нижней половине. Если же $q < p$, то применяются аналогичные рассуждения, а поиск рекурсивно продолжается в верхней половине A.

Суть в том, что на каждом шаге существует некая область A, в которой может находиться p; при каждой проверке размер этой области уменьшается вдвое. Каким же будет размер «активной» области A после k проверок? Начальный размер равен n, поэтому после k проверок он будет равен максимум $\left(\frac{1}{2}\right)^k n$.

Учитывая это обстоятельство, сколько шагов понадобится для сокращения активной области до постоянного размера? Значение k должно быть достаточно большим, чтобы $\left(\frac{1}{2}\right)^k n = O(1/n)$, а для этого мы можем выбрать $k = \log_2 n$. Таким образом, при $k = \log_2 n$ размер активной области сокращается до константы; в этой точке рекурсия прекращается, и поиск в оставшейся части массива может быть проведен за постоянное время.

Следовательно, бинарный поиск из-за последовательного сокращения области поиска имеет время выполнения $O(\log n)$. В общем случае временна́я граница $O(\log n)$ часто встречается в алгоритмах, выполняющих постоянный объем работы для отсечения постоянной доли входных данных. Принципиально здесь то, что $O(\log n)$ таких итераций сократят входные данные до постоянного размера, при котором задача обычно может быть решена напрямую.

2.5. Более сложная структура данных: приоритетная очередь

Наша основная цель в этой книге была представлена в начале главы: нас интересуют алгоритмы, обеспечивающие качественный выигрыш по сравнению с поиском методом «грубой силы»; в общем случае этот критерий имеет конкретную формулировку разрешимости с полиномиальным временем. Как правило, нахождение решения нетривиальной задачи с полиномиальным временем не зависит от мелких подробностей реализации, скорее различия между экспоненциальной и полиномиальной сложностью зависят от преодоления препятствий более высокого уровня. Но после разработки эффективного алгоритма решения задачи часто

бывает возможно добиться дополнительного улучшения времени выполнения за счет внимания к подробностям реализации, а иногда — за счет использования более сложных структур данных.

Некоторые сложные структуры данных фактически адаптированы для алгоритмов одного определенного типа, другие находят более общее применение. В этом разделе рассматривается одна из самых популярных нетривиальных структур данных — *приоритетная очередь*. Приоритетные очереди пригодятся для описания реализации некоторых алгоритмов графов, которые будут разработаны позднее в книге. А пока она послужит полезной иллюстрацией к анализу структуры данных, которая, в отличие от списков и массивов, должна выполнять некоторую нетривиальную работу при каждом обращении.

Задача

В реализации алгоритма устойчивых паросочетаний (см. раздел 2.3) упоминалась необходимость поддержания динамически изменяемого множества S (множества всех свободных мужчин в данном примере). В такой ситуации мы хотим иметь возможность добавлять и удалять элементы из S, а также выбрать элемент из S, когда этого потребует алгоритм. Приоритетная очередь предназначена для приложений, в которых элементам назначается *приоритет* (ключ), и каждый раз, когда потребуется выбрать элемент из S, должен выбираться элемент с наивысшим приоритетом.

Приоритетная очередь представляет собой структуру данных, которая поддерживает множество элементов S. С каждым элементом $v \in S$ связывается ключ $\texttt{key}(v)$, определяющий приоритет элемента v; меньшие ключи представляют более высокие приоритеты. Приоритетные очереди поддерживают добавление и удаление элементов из очереди, а также выбор элемента с наименьшим ключом. В нашей реализации приоритетных очередей также будут поддерживаться дополнительные операции, краткая сводка которых приведена в конце раздела.

Основная область применения приоритетных очередей, которая обязательно должна учитываться при обсуждении их общих возможностей, — задача управления событиями реального времени (например, планирование выполнения процессов на компьютере). Каждому процессу присваивается некоторый приоритет, но порядок поступления процессов не совпадает с порядком их приоритетов. Вместо этого имеется текущее множество активных процессов; мы хотим иметь возможность извлечь процесс с наивысшим приоритетом и активизировать его. Множество процессов можно представить в виде приоритетной очереди, в которой ключ процесса представляет величину приоритета. Планирование процесса с наивысшим приоритетом соответствует выбору из приоритетной очереди элемента с наименьшим ключом; параллельно новые процессы должны вставляться в очередь по мере поступления в соответствии со значениями приоритетов.

На какую эффективность выполнения операций с приоритетной очередью можно рассчитывать? Мы покажем, как реализовать приоритетную очередь, которая в любой момент времени содержит не более n элементов, с возможностью добавления и удаления элементов и выбора элемента с наименьшим ключом за время $O(\log n)$.

Прежде чем переходить к обсуждению реализации, рассмотрим очень простое применение приоритетных очередей. Оно демонстрирует, почему время $O(\log n)$ на операцию является «правильной» границей, к которой следует стремиться.

(2.11) Последовательность из $O(n)$ операций приоритетной очереди может использоваться для сортировки множества из n чисел.

Доказательство. Создадим приоритетную очередь H и вставим каждое число в H, используя его значение в качестве ключа. Затем будем извлекать наименьшие числа одно за другим, пока не будут извлечены все числа; в результате полученная последовательность чисел будет отсортирована по порядку. ■

Итак, с приоритетной очередью, способной выполнять операции вставки и извлечения минимума с временем $O(\log n)$ за операцию, можно отсортировать n чисел за время $O(n \log n)$. Известно, что в модели вычислений на базе сравнений (при которой каждая операция обращается к входным данным только посредством сравнивания пары чисел) время, необходимое для сортировки, должно быть пропорционально как минимум $n \log n$, так что (2.11) поясняет, что в некотором смысле время $O(\log n)$ на операцию — лучшее, на что можно рассчитывать. Следует отметить, что реальная ситуация немного сложнее: реализации приоритетных очередей более сложные, чем представленная здесь, могут улучшать время выполнения некоторых операций и предоставлять дополнительную функциональность. Но из (2.11) следует, что любая последовательность операций приоритетной очереди, приводящая к сортировке n чисел, должна занимать время, пропорциональное по меньшей мере $n \log n$.

Структура данных для реализации приоритетной очереди

Для реализации приоритетной очереди мы воспользуемся структурой данных, называемой *кучей* (heap). Прежде чем обсуждать организацию кучи, следует рассмотреть, что происходит при более простых, более естественных подходах к реализации функций приоритетной очереди. Например, можно объединить элементы в список и завести отдельный указатель `Min` на элемент с наименьшим ключом. Это упрощает добавление новых элементов, но с извлечением минимума возникают сложности. Найти минимум несложно (просто перейти по указателю `Min`), но после удаления минимального элемента нужно обновить `Min` для следующей операции, а для этого потребуется просканировать все элементы за время $O(n)$ для нахождения нового минимума.

Это усложнение наводит на мысль, что элементы стоило бы хранить в порядке сортировки ключей. Такая структура упростит извлечение элемента с наименьшим ключом, но как организовать добавление новых элементов? Может, стоит хранить элементы в массиве или в связанном списке? Допустим, добавляется элемент s с ключом $\texttt{key}(s)$. Если множество S хранится в отсортированном массиве, можно воспользоваться бинарным поиском для нахождения в массиве позиции вставки s за время $O(\log n)$, но для вставки s в массив все последующие элементы

необходимо сдвинуть на одну позицию вправо. Сдвиг займет время $O(n)$. С другой стороны, если множество хранится в двусвязном отсортированном списке, вставка в любой позиции выполняется за время $O(1)$, но двусвязный список не поддерживает бинарный поиск, поэтому для поиска позиции вставки s может потребоваться время до $O(n)$.

Определение кучи

Итак, во всех простых решениях по крайней мере одна из операций выполняется за время $O(n)$ — это намного больше времени $O(\log n)$, на которое мы надеялись. На помощь приходит *куча* — структура данных, в нашем контексте сочетающая преимущества сортированного массива и списка. На концептуальном уровне куча может рассматриваться как сбалансированное бинарное дерево (слева на рис. 2.3). У дерева имеется корень, а каждый узел может иметь до двух дочерних узлов: левый и правый. Говорят, что ключи такого бинарного дерева следуют в *порядке кучи*, если ключ каждого элемента по крайней мере не меньше ключа элемента его родительского узла в дереве. Другими словами,

Порядок кучи: *для каждого элемента v в узле i элемент w родителя i удовлетворяет условию* $\texttt{key}(w) \leq \texttt{key}(v)$.

На рис. 2.3 числа в узлах определяют ключи соответствующих элементов.

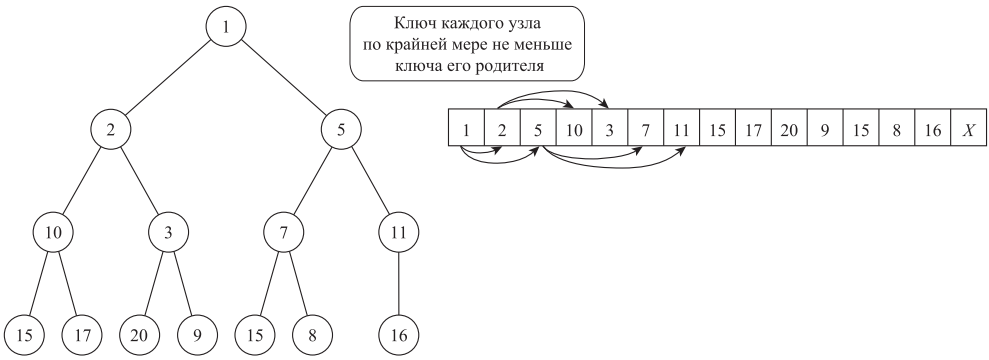


Рис. 2.3. Куча в виде бинарного дерева (слева) и в виде массива (справа). Стрелками обозначены дочерние узлы трех верхних узлов дерева

Прежде чем обсуждать работу с кучей, следует определиться со структурой данных, которая будет использоваться для ее представления. Можно использовать указатели: каждый узел содержит элемент, его ключ и три указателя (два дочерних узла и родительский узел). Однако если заранее известна граница N общего количества элементов, находящихся в куче в любой момент времени, можно обойтись и без указателей. Такие кучи могут храниться в массиве H с индексами $i = 1, ..., N$. Узлы кучи соответствуют позициям в массиве. $H[1]$ представляет собой корневой узел, а для любого узла в позиции i его дочерние узлы находятся в позициях $\texttt{leftChild}(i) = 2i$ и $\texttt{rightChild}(i) = 2i + 1$. Таким образом, два дочерних узла корня находятся в позициях 2 и 3, а родитель узла в позиции i находится в позиции

parent(i) = $\lfloor i/2 \rfloor$. Если куча в какой-то момент содержит $n < N$ элементов, то первые n позиций массива используются для хранения n элементов кучи, а количество элементов в H обозначается length(H). С таким представлением куча в любой момент времени является сбалансированной. В правой части рис. 2.3 показано представление в виде массива для кучи, изображенной в левой части.

Реализация операций с кучей

Элементом кучи с наименьшим ключом является ее корень, так что поиск минимального элемента выполняется за время $O(1)$. Как происходит добавление или удаление элементов кучи? Допустим, в кучу добавляется новый элемент v, а куча H на данный момент содержит $n < N$ элементов. После добавления она будет содержать $n + 1$ элементов. Начать можно с добавления нового элемента v в последнюю позицию $i = n + 1$, для чего выполняется простое присваивание $H[i] = v$. К сожалению, при этом нарушается основное свойство кучи, так как ключ элемента v может быть меньше ключа его родителя. Полученная структура данных почти является кучей, если не считать небольшой «поврежденной» части, где в конце был вставлен элемент v.

Для восстановления кучи можно воспользоваться несложной процедурой. Пусть j = parent(i) = $\lfloor i/2 \rfloor$ является родителем узла i, а $H[j] = w$. Если key[v] < key[w], позиции v и w просто меняются местами. Перестановка восстанавливает свойство кучи в позиции i, но может оказаться, что полученная структура нарушает свойство кучи в позиции j, — иначе говоря, место «повреждения» переместилось вверх от i к j. Рекурсивное повторение этого процесса от позиции j = parent(i) продолжает восстановление кучи со смещением поврежденного участка вверх.

На рис. 2.4 показаны первые два шага процесса после вставки.

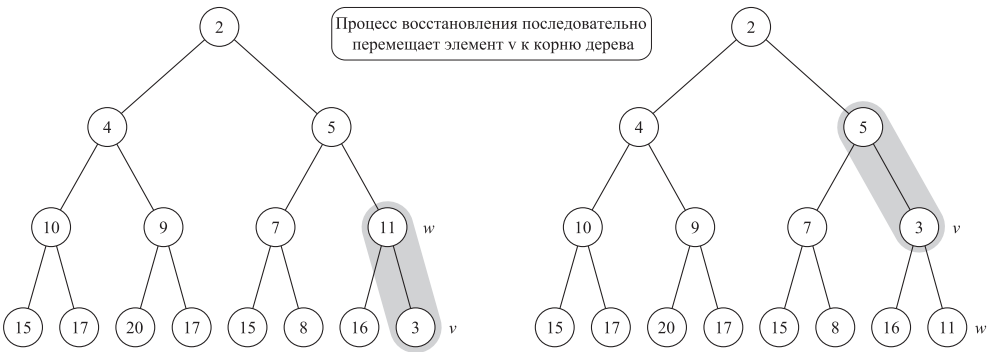


Рис. 2.4. Восходящий процесс восстановления кучи. Ключ 3 в позиции 16 слишком мал (слева). После перестановки ключей 3 и 11 нарушение структуры кучи сдвигается на один шаг к корню дерева (справа)

```
Heapify-up(H,i):
   Если i > 1
      Присвоить j = parent(i) = ⌊i/2⌋
```

```
    Если key[H[i]]<key[H[j]]
       Поменять местами элементы H[i] и H[j]
       Heapify-up(H,j)
    Конец Если
  Конец Если
```

Чтобы понять, почему эта процедура работает и в конечном итоге приводит к восстановлению порядка, полезно вникнуть структуру «слегка поврежденной» кучи в середине процесса. Допустим, H — массив, а v — элемент в позиции i. Будем называть *H псевдокучей со слишком малым ключом $H[i]$*, если существует такое значение $\alpha \geq \text{key}(v)$, что повышение $\text{key}(v)$ до α приведет к выполнению основного свойства кучи для полученного массива. (Другими словами, элемент v в $H[i]$ слишком мал, но повышение его до α решит проблему.) Важно заметить, что если H является псевдокучей со слишком малым корневым ключом (то есть $H[1]$), то в действительности H является кучей. Чтобы понять, почему это утверждение истинно, заметим, что если повышение $H[1]$ до α превратит H в кучу, то значение $H[1]$ также должно быть меньше значений обоих его дочерних узлов, — а следовательно, для H уже выполняется свойство порядка кучи.

(2.12) Процедура Heapify-up(H, i) исправляет структуру кучи за время $O(\log i)$ при условии, что массив H является псевдокучей со слишком малым ключом $H[i]$. Использование Heapify-up позволяет вставить новый элемент в кучу из n элементов за время $O(\log n)$.

Доказательство. Утверждение будет доказано методом индукции по i. Если $i = 1$, истинность очевидна, поскольку, как было замечено ранее, в этом случае H уже является кучей. Теперь рассмотрим ситуацию с $i > 1$: пусть $v = H[i]$, $j = \text{parent}(i)$, $w = H[j]$, и $\beta = \text{key}(w)$. Перестановка элементов v и w выполняется за время $O(1)$. После перестановки массив H представляет собой либо кучу, либо псевдокучу со слишком малым ключом $H[j]$ (теперь содержащим v). Это утверждение истинно, поскольку присваивание в узле j ключа β превращает H в кучу.

Итак, по принципу индукции рекурсивное применение Heapify-up(j) приведет к формированию кучи, как и требовалось. Процесс проходит по ветви от позиции i до корня, а значит, занимает время $O(\log i)$.

Чтобы вставить новый элемент в кучу, мы сначала добавляем его как последний элемент. Если новый элемент имеет очень большой ключ, то массив представляет собой кучу. В противном случае перед нами псевдокуча со слишком малым значением ключа нового элемента. Использование процедуры Heapify-up приводит к восстановлению свойства кучи. ■

Теперь рассмотрим операцию удаления элемента. Во многих ситуациях с применением приоритетных очередей не нужно удалять произвольные элементы — только извлекать минимум. В куче это соответствует нахождению ключа в корневом узле (который содержит минимум) с его последующим удалением; обозначим эту операцию ExtractMin(H). Теперь можно реализовать более общую операцию Delete(H, i), которая удаляет элемент в позиции i. Допустим, куча в данный момент содержит n элементов. После удаления элемента $H[i]$ в куче останется $n - 1$ элементов и помимо нарушения свойства порядка кучи в позиции i возникает «дыра», по-

тому что элемент $H[i]$ пуст. Начнем с заполнения «дыры» в H и переместим элемент w из позиции n в позицию i. После этого массив H по крайней мере удовлетворяет требованию о том, что его первые $n - 1$ элементов находятся в первых $n - 1$ позициях, хотя свойство порядка кучи при этом может не выполняться.

Тем не менее единственным местом кучи, в котором порядок может быть нарушен, является позиция i, так как ключ элемента w может быть слишком мал или слишком велик для позиции i. Если ключ слишком мал (то есть свойство кучи нарушается между узлом i и его родителем), мы можем использовать `Heapify-up`(i) для восстановления порядка. С другой стороны, если ключ `key`[w] слишком велик, свойство кучи может быть нарушено между i и одним или обоими его дочерними узлами. В этом случае будет использоваться процедура `Heapify-down`, очень похожая на `Heapify-up`, которая меняет элемент в позиции i с одним из его дочерних узлов и продолжает рекурсивно опускаться по дереву. На рис. 2.5 изображены первые шаги этого процесса.

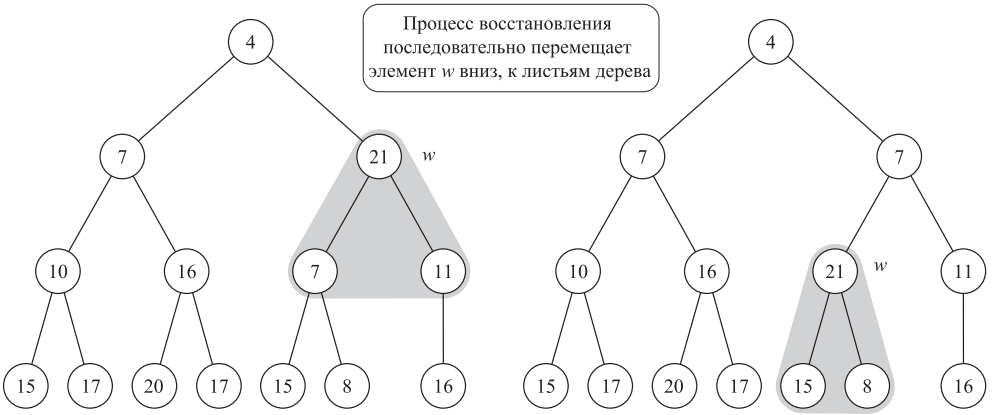


Рис. 2.5. Нисходящий процесс восстановления кучи. Ключ 21 в позиции 3 слишком велик (слева). После перестановки ключей 21 и 7 нарушение структуры кучи сдвигается на один шаг к низу дерева (справа)

```
Heapify-down(H,i):
   Присвоить n = length(H)
   Если 2i > n
      Завершить выполнение без изменения H
   Иначе Если 2i < n
      Присвоить left = 2i и right = 2i + 1
      Присвоить j индекс для минимизации key[H[left]] и key[H[right]]
   Иначе Если 2i = n
      Присвоить j = 2i
   Конец Если
   Если key[H[j]] < key[H[i]]
      Поменять местами элементы H[i] и H[j]
      Heapify-down(H, j)
   Конец Если
```

Допустим, H — массив, а w — элемент в позиции i. Будем называть H *псевдокучей со слишком большим ключом* $H[i]$, если существует такое $\alpha \le \texttt{key}(w)$, что понижение $\texttt{key}(w)$ до α приведет к выполнению основного свойства кучи для полученного массива. Если $H[i]$ соответствует узлу кучи (то есть не имеет дочерних узлов) и при этом является псевдокучей со слишком большим ключом $H[i]$, то в действительности H является кучей. В самом деле, если понижение $H[i]$ превратит H в кучу, то значение $H[i]$ уже больше родительского, а следовательно, для H уже выполняется свойство порядка кучи.

(2.13) Процедура `Heapify-down`(H, i) исправляет структуру кучи за время $O(\log n)$ при условии, что массив H является псевдокучей со слишком большим ключом $H[i]$. Использование `Heapify-up` или `Heapify-down`() позволяет удалить новый элемент в куче из n элементов за время $O(\log n)$.

Доказательство. Для доказательства того, что этот процесс исправляет структуру кучи, будет использован процесс обратной индукции по значению i. Пусть n — количество элементов в куче. Если $2i > n$, то, как было указано выше, H уже является кучей и доказывать ничего не нужно. В противном случае пусть j является дочерним узлом i с меньшим значением ключа, а $w = H[j]$. Перестановка элементов массива w и v выполняется за время $O(1)$. После перестановки массив H представляет собой либо кучу, либо псевдокучу со слишком большим ключом $H[j] = v$. Это утверждение истинно, поскольку присваивание $\texttt{key}(v) = \texttt{key}(w)$ превращает H в кучу. Теперь $j \ge 2i$, поэтому по предположению индукции рекурсивный вызов `Heapify-down` восстанавливает свойство кучи.

Алгоритм многократно перемещает элемент, находящийся в позиции i, вниз по ветви, поэтому за $O(\log n)$ итераций будет сформирована куча.

Чтобы использовать этот процесс для удаления элемента $v = H[i]$ из кучи, мы заменяем $H[i]$ последним элементом массива, $H[n] = w$. Если полученный массив не является кучей, то это псевдокуча со слишком малым или слишком большим значением ключа $H[i]$. Использование процедуры `Heapify-up` или `Heapify-down` приводит к восстановлению свойства кучи за время $O(\log n)$. ■

Реализация приоритетной очереди на базе кучи

Структура данных кучи с операциями `Heapify-down` и `Heapify-up` позволяет эффективно реализовать приоритетную очередь, которая в любой момент времени содержит не более N элементов. Ниже приводится сводка операций, которые мы будем использовать.

- `StartHeap`(N) возвращает пустую кучу H, настроенную для хранения не более N элементов. Операция выполняется за время $O(N)$, так как в ней инициализируется массив, используемый для хранения кучи.
- `Insert`(H, v) вставляет элемент v в кучу H. Если куча в данный момент содержит n элементов, операция выполняется за время $O(\log n)$.
- `FindMin`(H) находит наименьший элемент кучи H, но не удаляет его. Операция выполняется за время $O(1)$.

- `Delete`(H, i) удаляет элемент в позиции i. Для кучи, содержащей n элементов, операция выполняется за время $O(\log n)$.
- `ExtractMin`(H) находит и удаляет из кучи элемент с наименьшим значением ключа. Она объединяет две предыдущие операции, а следовательно, выполняется за время $O(\log n)$.

Также существует второй класс операций, которые должны выполняться с элементами по имени, а не по их позиции в куче. Например, во многих алгоритмах графов, использующих структуру данных кучи, элементы кучи представляют узлы графа, ключи которых вычисляются в процессе выполнения алгоритма. В разных точках этого алгоритма операция должна выполняться с некоторым узлом независимо от того, где он находится в куче.

Чтобы организовать эффективное обращение к нужным элементам приоритетной очереди, достаточно создать дополнительный массив `Position`, в котором хранится текущая позиция каждого элемента (каждого узла графа) в куче. С ним можно реализовать следующие операции.

- Для удаления элемента v применяется операция `Delete`(H,`Position`[v]). Создание массива не увеличивает общее время выполнения, поэтому удаление элемента v из кучи с n узлами выполняется за время $O(\log n)$.
- В некоторых алгоритмах также используется операция `ChangeKey`(H, v, α), которая изменяет ключ элемента v новым значением `key`(v) = α. Чтобы реализовать эту позицию за время $O(\log n)$, необходимо сначала определить позицию v в массиве, что делается при помощи массива `Position`. После определения позиции элемента v мы изменяем его ключ, а затем применяем процедуру `Heapify-up` или `Heapify-down` в зависимости от ситуации.

Упражнения с решениями

Упражнение с решением 1

Расположите функции из следующего списка по возрастанию скорости роста. Другими словами, если функция $g(n)$ в вашем списке следует непосредственно после $f(n)$, из этого следует, что $f(n) = O(g(n))$.

$$f_1(n) = 10^n$$

$$f_2(n) = n^{1/3}$$

$$f_3(n) = n^n$$

$$f_4(n) = \log_2 n$$

$$f_5(n) = 2^{\sqrt{\log_2 n}}$$

Решение

С функциями f_1, f_2 и f_4 все просто — они принадлежат базовым семействам экспоненциальных, полиномиальных и логарифмических функций. В частности, согласно (2.8), $f_4(n)=O(f_2(n))$, а согласно (2.9), $f_2(n)=O(f_1(n))$.

Разобраться с функцией f_3 тоже не так уж сложно. Сначала она растет медленнее 10^n, но после $n \geq 10$ очевидно $10^n \leq n^n$. Это именно то, что нужно для определения $O(\cdot)$: для всех $n \geq 10$ выполняется условие $10^n \leq cn^n$, где в данном случае $c = 1$, а значит, $10^n = O(n^n)$.

Остается функция f_5, которая выглядит, откровенно говоря, несколько странно. В таких случаях бывает полезно вычислить логарифм и посмотреть, не станет ли выражение более понятным. В данном случае $\log_2 f_5(n) = \sqrt{\log_2 n} = (\log_2 n)^{1/2}$. Как выглядят логарифмы других функций? $\log f_4(n) = \log_2 \log_2 n$, а $\log f_2(n) = \frac{1}{3}\log_2 n$. Все эти выражения могут рассматриваться как функции $\log_2 n$; используя обозначение $z = \log_2 n$, получаем

$$\log f_2(n) = \frac{1}{3}z$$

$$\log f_4(n) = \log_2 z$$

$$\log f_5(n) = z^{1/2}.$$

Теперь происходящее становится более понятным. Прежде всего, для $z \geq 16$ имеем $\log_2 z \leq z^{1/2}$. Но условие $z \geq 16$ эквивалентно $n \geq 2^{16} = 65\,536$; следовательно, после $n \geq 2^{16}$ выполняется условие $\log f_4(n) \leq \log f_5(n)$, а значит, $f_4(n) \leq f_5(n)$. Из этого следует, что $f_4(n) = O(f_5(n))$. Аналогично получаем $z^{1/2} \leq \frac{1}{3}z$ при $z \geq 9$ — иначе говоря, при $n \geq 2^9 = 512$. Для n, больших этой границы, выполняется условие $\log f_5(n) \leq \log f^2(n)$; следовательно, $f_5(n) \leq f_2(n)$, и $f_5(n) = O(f_2(n))$. Фактически мы определили, что скорость роста функции $2^{\sqrt{\log_2 n}}$ лежит где-то между логарифмическими и полиномиальными функциями.

Функция f_5 занимает место в списке между f_4 и f_2, а упорядочение функций на этом завершается.

Упражнение с решением 2

Пусть f и g — две функции, получающие неотрицательные значения, и $f = O(g)$. Покажите, что $g = \Omega(f)$.

Решение

Это упражнение помогает формализовать интуитивное ощущение, что $O(\cdot)$ и $\Omega(\cdot)$ в некотором смысле противоположны. Собственно, доказать его несложно, для этого достаточно развернуть определения.

Известно, что для некоторых констант c и n_0 выполняется условие $f(n) \le cg(n)$ для всех $n \ge n_0$. Разделив обе стороны на c, можно сделать вывод, что $g(n) \ge \frac{1}{c} f(n)$ для всех $n \ge n_0$. Но именно это и необходимо, чтобы показать, что $g = \Omega(f)$: мы установили, что $g(n)$ не меньше произведения $f(n)$ на константу (в данном случае $\frac{1}{c}$) для всех достаточно больших n (начиная с n_0).

Упражнения

1. В приведенном ниже списке указано время выполнения пяти алгоритмов (предполагается, что указано *точное* время выполнения). Насколько медленнее будет работать каждый из алгоритмов, если: (a) увеличить размер входных данных вдвое; (b) увеличить размер входных данных на 1?
   (a) n^2
   (b) n^3
   (c) $100n^2$
   (d) $n \log n$
   (e) 2^n
2. Есть шесть алгоритмов, время выполнения которых приведено ниже. (Предполагается, что указано точное количество выполняемых операций как функция размера данных n.) Предположим, имеется компьютер, способный выполнять 10^{10} операций в секунду, и результат должен быть вычислен не более чем за час. При каком наибольшем размере входных данных n результат работы каждого алгоритма может быть вычислен за час?
   (a) n^2
   (b) n^3
   (c) $100n^2$
   (d) $n \log n$
   (e) 2^n
   (f) 2^{2^n}
3. Расположите функции из следующего списка по возрастанию скорости роста. Другими словами, если функция $g(n)$ в вашем списке следует непосредственно после $f(n)$, из этого следует, что $f(n)=O(g(n))$.

$$f_1(n) = n^{2,5}$$

$$f_2(n) = \sqrt{2n}$$

$$f_3(n) = n + 10$$

$$f_4(n) = 10^n$$

$$f_5(n) = 100^n$$

$$f_6(n) = n^2 \log n$$

4. Расположите функции из следующего списка по возрастанию скорости роста. Другими словами, если функция $g(n)$ в вашем списке следует непосредственно после $f(n)$, из этого следует, что $f(n)=O(g(n))$.

$$g_1(n) = 2^{\sqrt{\log n}}$$

$$g_2(n) = 2^n$$

$$g_4(n) = n^{4/3}$$

$$g_3(n) = n(\log n)^3$$

$$g_5(n) = n^{\log n}$$

$$g_6(n) = 2^{2^n}$$

$$g_7(n) = 2^{n^2}$$

5. Допустим, имеются функции f и g, такие что $f(n) = O(g(n))$. Для каждого из следующих утверждений укажите, является оно истинным или ложным, и приведите доказательство или контрпример.
   (a) $\log_2 f(n) = O(\log_2 g(n))$.
   (b) $2^{f(n)} = O(2^{g(n)})$.
   (c) $f(n)^2 = O(g(n)^2)$.

6. Рассмотрим следующую задачу: имеется массив A, состоящий из n целых чисел $A[1]$, $A[2]$, ..., $A[n]$. Требуется вывести двумерный массив B размером $n \times n$, в котором $B[i,j]$ (для $i < j$) содержит сумму элементов массива от $A[i]$ до $A[j]$, то есть сумму $A[i] + A[i+1] + ... + A[j]$. (Для $i \geq j$ значение элемента массива $B[i,j]$ остается неопределенным, поэтому выводиться для таких элементов может что угодно.)

   Ниже приведен простой алгоритм решения этой задачи.

```
For i = 1, 2, ..., n
   For j = i + 1, i + 2, ..., n
      Просуммировать элементы массива от A[i] до A[j]
      Сохранить результат в B[i, j]
   Конец For
Конец For
```

(a) Для некоторой выбранной вами функции f приведите границу времени выполнения этого алгоритма в форме $O(f(n))$ для входных данных размера n (то есть границу количества операций, выполняемых алгоритмом).

(b) Для той же функции f покажите, что время выполнения алгоритма для входных данных размера n также равно $\Omega(f(n))$. (Тем самым определяется асимптотически точная граница $\Theta(f(n))$ для времени выполнения.)

(c) Хотя алгоритм, проанализированный в частях (a) и (b), является наиболее естественным подходом к решению задачи (в конце концов, он просто перебирает нужные элементы массива B и заполняет их значениями), в нем присутствуют избыточные источники неэффективности. Предложите другой алгоритм для решения этой задачи с асимптотически лучшим временем выполнения. Другими словами, вы должны разработать алгоритм со временем выполнения $O(g(n))$, где $\lim_{n\to\infty} g(n)/f(n) = 0$.

7. Существует целый класс народных и праздничных песен, в которых каждый куплет представляет собой предыдущий куплет с добавлением одной строки. Например, песня «Двенадцать дней Рождества» обладает этим свойством; добравшись до пятого куплета, вы поете о пяти золотых кольцах, затем упоминаются четыре дрозда из четвертого куплета, три курицы из третьего, две горлицы из второго и, конечно, куропатка из первого. Песня на арамейском языке «Хад гадья», распеваемая на Пасху, тоже обладает этим свойством, как и многие другие песни.

   Обычно такие песни исполняются долго, несмотря на относительно короткий текст. Чтобы определить слова и инструкции для одной из таких песен, достаточно указать новую строку, добавляемую в каждом куплете, не записывая заново все предшествующие строки. (Таким образом, строка «пять золотых колец» будет записана только один раз, хотя она присутствует в пятом и во всех последующих куплетах.)

   В этом примере присутствует асимптотическое поведение, которое стоит проанализировать. Для конкретности допустим, что каждая строка имеет длину, ограниченную константой c, и при исполнении песня состоит из n слов. Покажите, как записать слова такой песни в виде текста длиной $f(n)$ для функции $f(n)$ с минимально возможной скоростью роста.

8. Вы занимаетесь экстремальными испытаниями различных моделей стеклянных банок для определения высоты, при падении с которой они не разобьются. Тестовый стенд для этого эксперимента выглядит так: имеется лестница с n ступенями, и вы хотите найти самую высокую ступень, при падении с которой банка останется целой. Назовем ее *наивысшей безопасной ступенью*.

   На первый взгляд естественно применить бинарный поиск: сбросить банку со средней ступени, посмотреть, разобьется ли она, а затем рекурсивно повторить попытку от ступени $n/4$ или $3n/4$ в зависимости от результата. Но у такого решения есть недостаток: вероятно, при поиске ответа будет разбито слишком много банок.

С другой стороны, если вы стремитесь к экономии, можно опробовать следующую стратегию. Сначала банка сбрасывается с первой ступени, потом со второй и т. д. Высота каждый раз увеличивается на одну ступень, пока банка не разобьется. При таком подходе достаточно всего одной банки (как только она разобьется, вы получаете правильный ответ), но для получения ответа может потребоваться до n попыток (вместо $\log n$, как при бинарном поиске).

Похоже, намечается компромисс: можно выполнить меньше попыток, если вы готовы разбить больше банок. Чтобы лучше понять, как этот компромисс работает на количественном уровне, рассмотрим проведение эксперимента с ограниченным «бюджетом» из $k \geq 1$ банок. Другими словами, нужно найти правильный ответ (наивысшую безопасную ступень), использовав не более k банок.

(a) Допустим, вам выделен бюджет $k = 2$ банки. Опишите стратегию поиска наивысшей безопасной ступени, требующую не более $f(n)$ бросков для некоторой функции $f(n)$ с менее чем линейной скоростью роста. (Иначе говоря, в этом случае $lim_{n\to\infty} f(n)/n = 0$.)

(b) Теперь допустим, что доступен бюджет $k > 2$ банок с некоторым заданным k. Опишите стратегию поиска наивысшей безопасной ступени с использованием не более k банок. Если обозначить количество бросков по вашей стратегии $f_k(n)$, то функции f_1, f_2, f_3, ... должны обладать тем свойством, что каждая из них растет асимптотически медленнее предыдущей: $\lim_{n\to\infty} f_k(n)/f_{k-1}(n) = 0$ для всех k.

Примечания и дополнительная литература

Принятие разрешимости с полиномиальным временем как формального понятия эффективности было постепенным процессом, на который повлияли работы многих исследователей, среди которых были Кобхэм, Рабин, Эдмондс, Хартманис и Стирнс. В отчете Сипсера (Sipser, 1992) содержится как историческая, так и техническая информация по процессу разработки. Подобным образом использование асимптотической записи порядка скорости роста как границы времени выполнения алгоритмов (в отличие от точных формул с коэффициентами при старших членах и младшими членами) было решением, не столь очевидным на момент его введения; в Тьюринговской лекции Тарьяна (Tarjan, 1987) содержится интересная точка зрения на ранние представления исследователей, работавших в этой области, включая Хопкрофта, Тарьяна и других. Дополнительную информацию об асимптотической записи и скорости роста базовых функций можно найти у Кнута (Knuth, 1997a).

Реализация приоритетных очередей с использованием кучи, а также ее применение для сортировки обычно приписываются Уильямсу (Williams, 1964) и Флойду (Floyd, 1964). Приоритетная очередь представляет собой нетривиальную структуру данных с множеством разных применений; в следующих главах мы обсудим другие структуры данных, когда они будут полезны при реализации

конкретных алгоритмов. Структура данных `Union-Find` рассматривается в главе 4 для реализации алгоритма поиска минимального остовного дерева, а рандомизированное хеширование — в главе 13. Другие структуры данных рассматриваются в книге Тарьяна (Tarjan, 1983). Библиотека LEDA (Library of Efficient Datatypes and Algorithms) Мельхорна и Нэера (Mellhorn, Näher, 1999) предоставляет обширную подборку структур данных, используемых в комбинаторных и геометрических задачах.

Примечания к упражнениям

Упражнение 8 основано на задаче, о которой мы узнали от Сэма Туэга.

Глава 3
Графы

В этой книге мы будем в основном заниматься дискретными задачами. Если непрерывная математика занимается такими базовыми структурами, как вещественные числа, векторы и матрицы, в области дискретной математики были разработаны базовые комбинаторные структуры, заложенные в ее основу. Одной из самых фундаментальных и выразительных из таких структур является *граф*.

Чем больше работаешь с графами, тем скорее начинаешь замечать их повсюду. Мы начнем с базовых определений, относящихся к графам, и знакомства с целым спектром различных алгоритмических областей, в которых представление в виде графа появляется естественным образом. Затем будут рассмотрены некоторые базовые алгоритмические примитивы графов, начиная с задачи связности и разработки фундаментальных методов поиска по графу.

3.1. Основные определения и применения

Как говорилось в главе 1, граф G представляет собой обычный способ кодирования парных отношений в множестве объектов: он состоит из набора узлов V и набора ребер E, каждое из которых «соединяет» два узла. Таким образом, ребро $e \in E$ представляется двухэлементным подмножеством V: $e = \{u, v\}$ для некоторых $u, v \in V$, при этом u и v называются *концами* ребра e.

Ребра в графе обозначают симметричные отношения между их концами. Часто требуется представить асимметричные отношения; в таких случаях используется понятие *направленного графа*. Направленный граф G' состоит из набора узлов V и набора направленных ребер E', также называемых *дугами*. Каждое ребро $e' \in E$ представляется *упорядоченной парой* (u, v); иначе говоря, роли u и v не взаимозаменяемы, узел u называется *начальным*, а узел v — *конечным*. Также говорят, что ребро e' *выходит* из узла u и *входит* в узел v.

Если требуется подчеркнуть, что рассматриваемый граф не является направленным, его называют *ненаправленным графом*; тем не менее по умолчанию под термином «граф» понимается именно ненаправленный граф. Также стоит сделать пару замечаний по поводу нашего использования терминологии. Во-первых, хотя ребро e в ненаправленном графе должно обозначаться как множество узлов $\{u, v\}$, чаще (и даже в книгах) используется запись, принятая для упорядоченных пар:

$e = (u, v)$. Во-вторых, узлы графа часто называются *вершинами*; в данном контексте эти два термина имеют абсолютно одинаковый смысл.

Примеры графов

Определить граф проще простого: возьмите набор объектов и соедините их ребрами. Но на таком уровне абстракции трудно представить типичные ситуации, в которых встречается работа с графами. По этой причине мы предлагаем список распространенных контекстов, в которых графы играют важную роль как модель. Список получился достаточно обширным, не пытайтесь запомнить все его пункты; скорее это перечень полезных примеров для проверки базовых определений и алгоритмических задач, которые встретятся нам позднее в этой главе. Кроме того, при знакомстве с примерами будет полезно поразмыслить над смыслом узлов и ребер в контексте приложения. В одних случаях и узлы и ребра соответствуют физическим объектам реального мира, в других узлы являются реальными объектами, а ребра виртуальны; наконец, и ребра и узлы могут представлять чисто абстрактные понятия.

1. **Транспортные сети**. Карта маршрутов компании-авиаперевозчика естественным образом образует граф: узлы соответствуют аэропортам, а ребро из узла u в v существует при наличии беспосадочного перелета с вылетом из u и посадкой в v. При таком описании граф является направленным; тем не менее на практике при существовании ребра (u, v) почти всегда существует ребро (v, u), поэтому мы практически ничего не потеряем, если будем рассматривать карту маршрутов как ненаправленный граф с ребрами, которые соединяют пары аэропортов, связанные беспосадочными перелетами. При рассмотрении такого графа (который обычно изображается на задней обложке журналов для пассажиров) можно быстро заметить несколько подробностей: часто в графе присутствует небольшое количество «центров» с очень большим количеством инцидентных ребер; и от одного узла графа можно перейти к любому другому узлу за очень небольшое количество промежуточных пересадок.

   Другие транспортные сети тоже могут моделироваться подобным образом. Например, в железнодорожной сети каждый узел может представлять станцию, а ребро, соединяющее узлы u и v, — железнодорожный путь, соединяющий эти станции без промежуточных остановок. Стандартная схема метро в крупном городе представляет собой изображение такого графа.

2. **Коммуникационные сети**. Граф может рассматриваться как естественная модель компьютеров, объединенных в сеть передачи данных. Существуют разные способы такого моделирования. Во-первых, каждый узел может представлять компьютер, а ребро, соединяющее узлы u и v, — прямой физический канал, связывающий эти компьютеры. Кроме того, для описания крупномасштабных структур в Интернете узел часто определяется как группа машин, обслуживаемых одним интернет-провайдером; узлы u и v соединяются ребром, если между ними существует прямая одноранговая связь — проще говоря, соглашение о передаче данных по стандартному протоколу BGP, управляющему глобальной

маршрутизацией в Интернете. Следует заметить, что вторая модель «виртуальнее» первой, потому что связи представляют собой формальное соглашение наряду с физической линией связи.

При анализе беспроводных сетей обычно определяется граф, узлы которого представляют собой устройства, расположенные в разных местах физического пространства, а ребро из u в v существует в том случае, если узел v находится достаточно близко к u для приема сигнала. Такие графы часто удобно рассматривать как направленные, потому что может оказаться, что v может воспринимать сигнал u, а u не может воспринимать сигнал v (например, потому что на u установлен более мощный передатчик). Эти графы также интересны с геометрической точки зрения, потому что они приблизительно соответствуют размещению точек на плоскости с последующим соединением близко расположенных пар.

3. **Информационные сети**. Всемирная паутина может естественным образом рассматриваться как направленный граф, в котором узлы соответствуют вебстраницам, а ребро из u в v существует в том случае, если в u присутствует гиперссылка на v. Направленность графа здесь принципиальна; например, многие страницы содержат ссылки на популярные сайты новостей, тогда как на этих сайтах, естественно, обратных ссылок нет. Структура, сформированная всеми гиперссылками, может использоваться алгоритмами для определения самых важных страниц — эта методика применяется многими современными поисковыми системами.

   Гипертекстовой структуре Всемирной паутины предшествовали информационные сети, появившиеся за много десятков лет до Интернета, такие как сети перекрестных ссылок между статьями в энциклопедиях или других справочниках или сети библиографических ссылок в научных статьях.

4. **Социальные сети**. Для любой группы людей, общающихся друг с другом (работники одной компании, студенты, жители небольшого городка), можно определить сеть, узлы которой представляют людей, а ребро между u и v существует в том случае, если они являются друзьями. Ребра также могут представлять другие отношения, помимо дружеских: ненаправленное ребро (u, v) может обозначать романтические или финансовые отношения; направленное ребро (u, v) может указывать на то, что u обращается к v за советом или u включает v в свою адресную книгу электронной почты. Также легко представить двудольную социальную сеть, основанную на концепции *принадлежности*: для множества X людей и множества Y организаций ребро между $u \in X$ и $v \in Y$ определяется в том случае, если u принадлежит организации v.

   Такие сети широко применяются социологами для изучения динамики взаимодействий между людьми. В частности, они позволяют выявить наиболее «влиятельных» личностей в компании или организации, смоделировать отношения доверия в финансовой или политической среде или отследить процесс распространения слухов, анекдотов, болезней или вирусов, распространяемых по электронной почте.

5. **Сети зависимостей**. Модель направленного графа естественно подходит для отражения взаимозависимостей в группах объектов. Например, для учебного плана в колледже или университете каждый узел может представлять учебный курс, а ребро из *u* в *v* существует в том случае, если *u* является предпосылкой для *v*. Для списка функций или модулей крупной программной системы узел может представлять функцию, а ребро из *u* в *v* существует в том случае, если *u* вызывает *v*. Или для множества видов в экосистеме можно определить граф (пищевую сеть), в которой узлы представляют разные виды, а ребро из *u* в *v* означает, что вид *u* питается представителями вида *v*.

   Этот список далеко не полон — даже для простой классификации его задач. Он просто дает примеры, о которых полезно помнить, когда мы займемся рассмотрением графов в алгоритмическом контексте.

Пути и связность

Одной из основополагающих операций с графами является обход последовательности узлов, соединенных ребрами. В приведенных выше примерах такой обход может соответствовать сеансу просмотра веб-страниц с переходами по гиперссылкам; слухам, передаваемым между людьми на другой конец света; перелету авиапассажира из Сан-Франциско в Рим с несколькими пересадками.

С учетом сказанного *путь* в ненаправленном графе $G = (V, E)$ определяется как последовательность P узлов $v_1, v_2, ..., v_{k-1}, v_k$, обладающая тем свойством, что каждая очередная пара v_i, v_{i+1} соединяется ребром из G. P часто называется путем из v_1 в v_k, или путем $v_1 - v_k$. Например, узлы 4, 2, 1, 7, 8 образуют путь на рис. 3.1. Путь называется *простым*, если все его узлы отличны друг от друга. *Цикл* представляет собой путь $v_1, v_2, ..., v_{k-1}, v_k$, в котором $k > 2$, первые $k - 1$ узлов различны, а $v_1 = v_k$, — другими словами, последовательность узлов, которая «возвращается» к своему началу. Все эти определения естественным образом переносятся на направленные графы со следующим изменением: в направленном пути или цикле каждая пара последовательных узлов обладает тем свойством, что (v_i, v_{i+1}) является ребром. Другими словами, последовательность узлов в пути или цикле должна учитывать направленность ребер.

Ненаправленный граф называется *связным*, если для каждой пары узлов *u* и *v* существует путь из *u* в *v*. Для направленных графов способ определения связности не столь очевиден: может оказаться, что в графе существует путь из *u* в *v*, тогда как пути из *v* в *u* не существует. Направленный граф называется *сильно связным*, если для каждых двух узлов *u* и *v* существует путь из *u* в *v*, и путь из *v* в *u*.

Кроме самого факта существования пути между парой узлов *u* и *v* нам, возможно, понадобится узнать, существует ли короткий путь. *Расстояние* между двумя узлами нами *u* и *v* определяется как минимальное количество ребер в пути *u*–*v*. (Для обозначения расстояния между узлами, между которыми не существует соединяющего пути, можно использовать условный знак — например, ∞). Чтобы понять, откуда происходит термин «расстояние», вообразите G как представление

коммуникационной или транспортной сети; для перехода из *u* в *v* хотелось бы выбрать маршрут с минимальным количеством «пересадок».

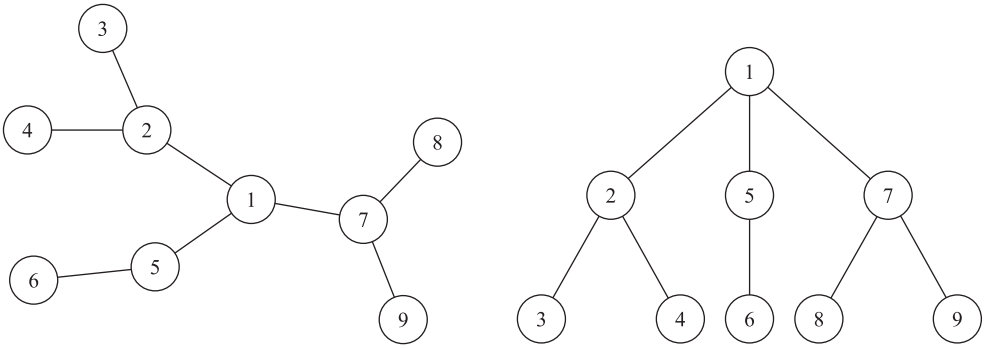


Рис. 3.1. Два представления одного дерева. В правой части хорошо видно, что узел 1 является корнем дерева

Деревья

Ненаправленный граф называется *деревом*, если он является связным и не содержит циклов. Например, два графа на рис. 3.1 являются деревьями. Строго говоря, деревья составляют простейшую разновидность связных графов: удаление любого узла из дерева приведет к нарушению его связности.

Для рассмотрения структуры дерева *T* удобно представить некоторый узел *r* как корень дерева. По сути, дерево «цепляется» за узел *r*, а остальные его ветви свисают вниз под действием силы тяжести. А если выражаться точнее, каждое ребро *T* «ориентируется» в направлении от *r*; для каждого из остальных узлов *v* *родительским* называется узел *u*, который непосредственно предшествует *v* на пути из *r*; узел *w* называется *дочерним* по отношению к *v*, если *v* является родительским узлом для *w*. В более общей формулировке узел *w* называется *потомком* *v* (а *v* — *предком* *w*), если *v* лежит на пути от корня к *w*; узел *x* называется *листовым* (или просто *листом*), если у него нет потомков. Таким образом, две диаграммы на рис. 3.1 изображают одно и то же дерево *T* — те же пары узлов, соединенные ребрами, но правая диаграмма представляет дерево, полученное в результате назначения узла 1 корнем *T*.

Корневые деревья относятся к числу фундаментальных объектов в теории программирования, потому что они представляют концепцию иерархии. Например, корневое дерево на рис. 3.1 может служить представлением организационной структуры небольшой компании с 9 работниками: работники 3 и 4 подчиняются работнику 2; работники 2, 5 и 7 подчиняются работнику 1 и т. д. Многие сайты имеют иерархическую структуру для упрощения навигации. Например, на типичном сайте кафедры информатики главная страница является корневой; страница Люди (и Учебные курсы) является дочерней по отношению к корневой; страницы Преподаватели и Студенты являются дочерними по отношению к странице Люди; домашние страницы профессоров являются дочерними по отношению к странице Преподаватели и т. д.

Для наших целей определение корня дерева T может концептуально упростить ответы на некоторые вопросы по поводу T. Например, если имеется дерево T с n узлами, сколько ребер оно имеет? У каждого узла, отличного от корня, имеется одно ребро, ведущее «наверх» по направлению к родителю; и наоборот, каждое ребро ведет вверх ровно от одного некорневого узла. Это позволяет очень легко доказать следующий факт.

(3.1) Каждое дерево из n узлов содержит ровно $n - 1$ ребро.

Более того, истинно и следующее — более сильное — утверждение, хотя здесь его доказательство не приводится.

(3.2) Допустим, G — ненаправленный граф с n узлами. Если истинны любые два из следующих утверждений, то автоматически выполняется и третье.

(i) Граф G является связным.

(ii) Граф G не содержит циклов.

(iii) Граф G содержит $n - 1$ ребро.

А теперь обратимся к роли деревьев в фундаментальной алгоритмической идее *обхода графа*.

3.2. Связность графа и обход графа

Итак, мы сформулировали некоторые фундаментальные понятия, относящиеся к графам. Пора перейти к одному из важнейших алгоритмических вопросов: связности узлов. Допустим, имеется граф $G = (V, E)$ и два конкретных узла s и t. Нужно найти эффективный алгоритм для получения ответа на следующий вопрос: существует ли в G путь из s в t? Будем называть его задачей проверки *связности s–t*.

Для очень малых графов бывает достаточно взглянуть на граф. Но для больших графов поиск пути может потребовать определенной работы. Более того, задача связности *s–t* также может рассматриваться как задача поиска пути в лабиринте. Если представить G как лабиринт, комнаты которого соответствуют узлам, а коридоры — ребрам, соединяющим узлы (комнаты), то задача заключается в том, чтобы начать с комнаты s и добраться до другой заданной комнаты t. Несколько эффективный алгоритм можно разработать для этой задачи?

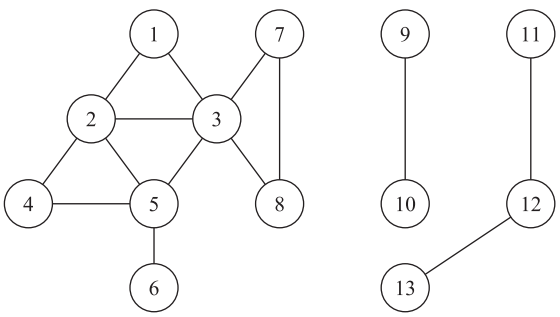


Рис. 3.2. На этом графе существуют пути от узла 1 к узлам 2–8, но не к узлам 9–13

В этом разделе описаны два естественных алгоритма для решения этой задачи на высоком уровне: поиск в ширину (BFS, Breadth-First Search) и *поиск в глубину* (DFS, Depth-First Search). В следующем разделе мы обсудим эффективную реализацию обоих алгоритмов на основании структуры данных для представления графа, описывающего входные данные алгоритма.

Поиск в ширину

Вероятно, простейшим алгоритмом для проверки связности s–t является алгоритм *поиска в ширину* (BFS), при котором просмотр ведется от s во все возможных направлениях с добавлением одного «уровня» за раз. Таким образом, алгоритм начинает с s и включает в поиск все узлы, соединенные ребром с s, — так формируется первый уровень поиска. Затем включаются все узлы, соединенные ребром с любым узлом из первого уровня, — второй уровень. Просмотр продолжается до того момента, когда очередная попытка не обнаружит ни одного нового узла.

Если в примере на рис. 3.2 начать с узла 1, первый уровень будет состоять из узлов 2 и 3, второй — из узлов 4, 5, 7 и 8, а третий только из узла 6. На этой стадии поиск останавливается, потому что новых узлов уже не осталось (обратите внимание на то, что узлы 9–13 остаются недостижимыми для поиска).

Как наглядно показывает этот пример, у алгоритма имеется естественная физическая интерпретация. По сути, мы начинаем с узла s и последовательно «затапливаем» граф расширяющейся волной, которая стремится охватить все узлы, которых может достичь. Уровень узла представляет момент времени, в который данный узел будет достигнут при поиске.

Уровни L_1, L_2, L_3, ..., создаваемые алгоритмом BFS, более точно определяются следующим образом:

- Уровень L_1 состоит из всех узлов, являющихся соседями s. (Для удобства записи мы иногда будем использовать уровень L_0 для обозначения множества, состоящего только из s.)
- Если предположить, что мы определили уровни L_1, ..., L_j, то уровень L_{j+1} состоит из всех узлов, не принадлежащих ни одному из предшествующих уровней и соединенных ребром с узлом уровня L_j.

Если вспомнить, что расстояние между двумя узлами определяется как минимальное количество ребер в пути, соединяющем эти узлы, становится понятно, что уровень L_1 представляет собой множество всех узлов, находящихся на расстоянии 1 от s, или в более общей формулировке — уровень L_j представляет собой множество всех узлов, находящихся от s на расстоянии ровно j. Узел не присутствует ни на одном уровне в том, и только в том случае, если пути к нему не существует. Таким образом, алгоритм поиска в ширину определяет не только узлы, достижимые из s, но и вычисляет кратчайшие пути до них. Это свойство алгоритма обобщается в виде следующего факта.

(3.3) Для каждого $j \geq 1$ уровень L_j, создаваемый алгоритмом BFS, состоит из всех узлов, находящихся от s на расстоянии ровно j. Путь от s к t существует в том, и только в том случае, если узел t встречается на каком-либо уровне.

Другое свойство поиска в ширину заключается в том, что этот алгоритм естественным образом генерирует дерево T с корнем s, которое состоит из узлов, достижимых из s. А конкретнее, для каждого узла v (отличного от s) рассмотрим момент, когда v впервые «обнаруживается» алгоритмом BFS; это происходит в тот момент, когда проверяется некоторый узел u из уровня L_j и обнаруживается, что из этого узла ребро ведет к узлу v, который не встречался ранее. В этот момент ребро (u, v) добавляется в дерево T — u становится родителем v, представляя тот факт, что u «отвечает» за завершение пути к v. Дерево, построенное таким образом, называется *деревом поиска в ширину*.

На рис. 3.3 изображен процесс построения дерева BFS с корнем в узле 1 для графа, изображенного на рис. 3.2. Сплошными линиями обозначены ребра T, а пунктирными — ребра G, не принадлежащие T. Ход выполнения алгоритма BFS, приводящего к построению этого дерева, можно описать следующим образом.

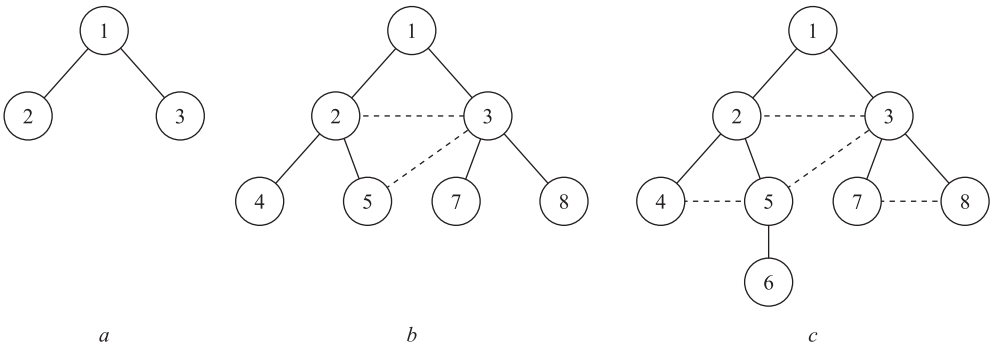


Рис. 3.3. Построение дерева поиска в ширину T для графа на рис. 3.2; фазы (a), (b) и (c) обозначают добавляемые уровни. Сплошными линиями обозначаются ребра T; пунктирные ребра входят в компоненту связности G, содержащую узел 1, но не принадлежащую T

(a) Начиная от узла 1 уровень L_1 состоит из узлов $\{2, 3\}$.

(b) Затем для построения уровня L_2 узлы L_1 последовательно проверяются в некотором порядке (допустим, сначала 2, а потом 3). При проверке узла 2 обнаруживаются узлы 4 и 5, поэтому узел 2 становится их родителем. При проверке узла 2 также обнаруживается ребро к узлу 3, но оно не добавляется в дерево BFS, потому что узел 3 нам уже известен.

Узлы 7 и 8 впервые обнаруживаются при проверке узла 3. С другой стороны, ребро от 3-го к 5-му становится еще одним ребром G, которое не включается в T, потому что к моменту обнаружения этого ребра, выходящего из узла 3, узел 5 уже известен.

(c) Затем по порядку проверяются узлы уровня L_2. Единственным новым узлом, обнаруженным при проверке L_2, оказывается узел 6, который добавляется в уровень L_3. Обратите внимание: ребра (4, 5) и (7, 8) не добавляются в дерево BFS, потому что они не приводят к обнаружению новых узлов.

(d) При проверке узла 6 новые узлы не обнаруживаются, поэтому в уровень L_4 ничего не добавляется, и работа алгоритма на этом завершается. Полное дерево BFS изображено на рис. 3.3(c).

Стоит заметить, что при выполнении алгоритма BFS для этого графа ребра, не входящие в дерево, соединяют либо узлы одного уровня, либо узлы смежных уровней. Сейчас мы докажем, что это свойство присуще деревьям BFS вообще.

(3.4) Пусть T — дерево поиска в ширину, x и y — узлы T, принадлежащие уровням L_i и L_j соответственно, а (x, y) — ребро G. В таком случае i и j отличаются не более чем на 1.

Доказательство. Действуя от противного, предположим, что i и j отличаются более чем на 1 — например, что $i < j - 1$. Рассмотрим точку алгоритма BFS, в которой проверяются ребра, инцидентные x. Так как x принадлежит уровню L_i, то все узлы, обнаруженные от x, принадлежат уровням L_{i+1} и менее; следовательно, если y является соседом x, то этот узел должен быть обнаружен к текущему моменту, а это означает, что он должен принадлежать уровню L_{i+1} и менее. ■

Связная компонента

Множество узлов, обнаруживаемых алгоритмом BFS, в точности соответствует множеству узлов, достижимых из начального узла s. Это множество R называется *компонентой связности* G, содержащей s; зная компоненту связности, содержащую s, для ответа на вопрос о связности s–t достаточно проверить, принадлежит ли t компоненте связности.

Если задуматься, становится ясно, что BFS — всего лишь один из возможных способов построения этой компоненты. На более общем уровне компоненту R можно построить «проверкой» G в любом порядке, начиная с s. Сперва определяется $R = \{s\}$. Затем в любой момент времени, если удается найти ребро (u, v), для которого $u \in R$ и $v \notin R$, узел v добавляется в R. В самом деле, если существует путь P из s в u, то существует путь из s в v, который состоит из P с последующим переходом по ребру (u, v). На рис. 3.4 изображен этот базовый шаг расширения компоненты R.

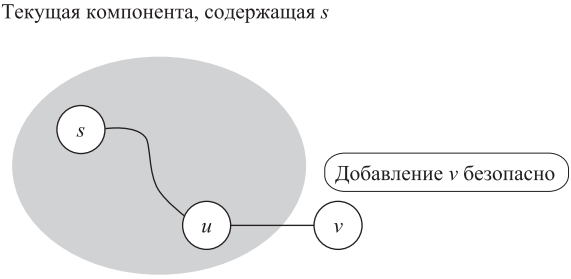


Рис. 3.4. При расширении компоненты связности, содержащей s, мы ищем узлы, которые еще не посещались, такие как v

Предположим, множество R продолжает расти до того момента, пока не останется ни одного ребра, ведущего из R; иначе говоря, выполняется следующий алгоритм.

```
R состоит из узлов, к которым существует путь из s
Перед началом выполнения R = {s}
Пока существует ребро (u, v), для которого u ∈ R и v ∉ R
   Добавить v в R
Конец Пока
```

Ниже сформулировано ключевое свойство этого алгоритма.

(3.5) Множество R, построенное в конце выполнения этого алгоритма, в точности совпадает с компонентой связности G, содержащей s.

Доказательство. Ранее уже было показано, что для любого узла $v \in R$ существует путь из s в v.

Теперь рассмотрим узел $w \in R$; действуя от обратного, предположим, что в G существует путь s-w, который будет обозначаться P. Так как $s \in R$, но $w \notin R$, в пути P должен существовать первый узел v, который не принадлежит R, и этот узел v отличен от s. Следовательно, должен существовать узел u, непосредственно предшествующий v в P, такой что (u, v) является ребром. Более того, поскольку v является первым узлом P, не принадлежащим R, должно выполняться условие $u \in R$. Отсюда следует, что (u, v) — ребро, для которого $u \in R$ и $v \notin R$; однако это противоречит правилу остановки алгоритма. ■

Заметьте, что для любого узла t в компоненте R можно легко восстановить фактический путь от s к t по описанному выше принципу: для каждого узла v просто фиксируется ребро (u, v), которое рассматривалось на итерации, в которой узел v был добавлен в R. Перемещаясь по этим ребрам в обратном направлении от t, мы обрабатываем серию узлов, добавлявшихся на все более и более ранних итерациях, постепенно достигая s; таким образом определяется путь s–t.

В завершение следует отметить, что общий алгоритм расширения R недостаточно точно определен: как решить, какое ребро должно рассматриваться следующим? Среди прочего, алгоритм BFS предоставляет способ упорядочения посещаемых узлов — по последовательным уровням, на основании их расстояния от s. Однако существуют и другие естественные способы расширения компоненты, часть из которых ведет к эффективным алгоритмам решения задачи связности с применением поисковых схем, основанных на других структурах. Сейчас мы займемся другим алгоритмом такого рода — *поиском в глубину* — и изучим некоторые из его базовых свойств.

Поиск в глубину

Другой естественный метод поиска узлов, достижимых из s, естественно применяется в ситуации, когда граф G действительно представляет собой лабиринт из взаимосвязанных комнат. Вы начинаете от s и проверяете первое ребро, ведущее из него, — допустим, к узлу v. Далее вы следуете по первому ребру, выходящему из

v, и продолжаете действовать по этой схеме, пока не окажетесь в «тупике» — узле, для которого вы уже исследовали всех соседей. В таком случае вы возвращаетесь к узлу, у которого имеется непроверенный сосед, и продолжаете от него. Этот алгоритм называется *алгоритмом поиска в глубину* (DFS, Depth-First Search), потому что он продвигается по G на максимально возможную глубину и отступает только по мере необходимости.

Алгоритм DFS также является конкретной реализацией общего алгоритма расширения компоненты, представленного выше. Проще всего описать его в рекурсивной форме: DFS можно запустить от любой начальной точки, но с хранением глобальной информации об уже посещенных узлах.

```
DFS(u):
   Пометить узел u как «проверенный» и добавить u в R
   Для каждого ребра (u, v), инцидентного u
      Если узел v не помечен как «проверенный»
         Рекурсивно вызвать DFS(v)
      Конец Если
   Конец цикла
```

Чтобы применить его для решения задачи связности s–t, достаточно изначально объявить все узлы «непроверенными» и вызвать $DFS(s)$.

Между алгоритмами DFS и BFS существуют как фундаментальное сходство, так и фундаментальные отличия. Сходство основано на том факте, что оба алгоритма строят компоненту связности, содержащую s, и, как будет показано в следующем разделе, в обоих случаях достигаются сходные уровни эффективности.

Хотя алгоритм DFS в конечном итоге посещает точно те же узлы, что и BFS, обычно посещение происходит в совершенно ином порядке; поиск в глубину проходит по длинным путям и может заходить очень далеко от s, прежде чем вернуться к более близким непроверенным вариантам. Это различие проявляется в том факте, что алгоритм DFS, как и BFS, строит естественное корневое дерево T из компоненты, содержащей s, но обычно такое дерево имеет совершенно иную структуру. Узел s становится корнем дерева T, а узел u становится родителем v, если u отвечает за обнаружение v. А именно, если $DFS(v)$ вызывается непосредственно во время вызова $DFS(u)$, ребро (u, v) добавляется в T. Полученное дерево называется *деревом поиска в глубину* компоненты R.

На рис. 3.5 изображен процесс построения дерева DFS с корнем в узле 1 для графа на рис. 3.2. Сплошными линиями обозначены ребра T, а пунктирными — ребра G, не принадлежащие T. Выполнение алгоритма DFS начинается с построения пути, содержащего узлы 1, 2, 3, 5, 4. Выполнение заходит в тупик в узле 4, в котором найти новые узлы не удается, поэтому алгоритм возвращается к узлу 5, находит узел 6, снова возвращается к 4 и находит узлы 7 и 8. В этой точке в компоненте связности новых узлов нет, поэтому все незавершенные рекурсивные вызовы DFS завершаются один за другим, а выполнение подходит к концу. Полное дерево DFS изображено на рис. 3.5, *g*.

Этот пример дает представление о внешних отличиях деревьев DFS от деревьев BFS. Такие деревья обычно получаются узкими и глубокими, тогда как для последних характерны минимально короткие пути от корня до листьев. Тем не менее, как и в случае BFS, мы можем выдвинуть достаточно сильное утверждение о расположении ребер G, не входящих в дерево, относительно ребер DFS-дерева T: как и показано на схеме, ребра, не входящие в дерево, могут соединять только предков T с потомками.

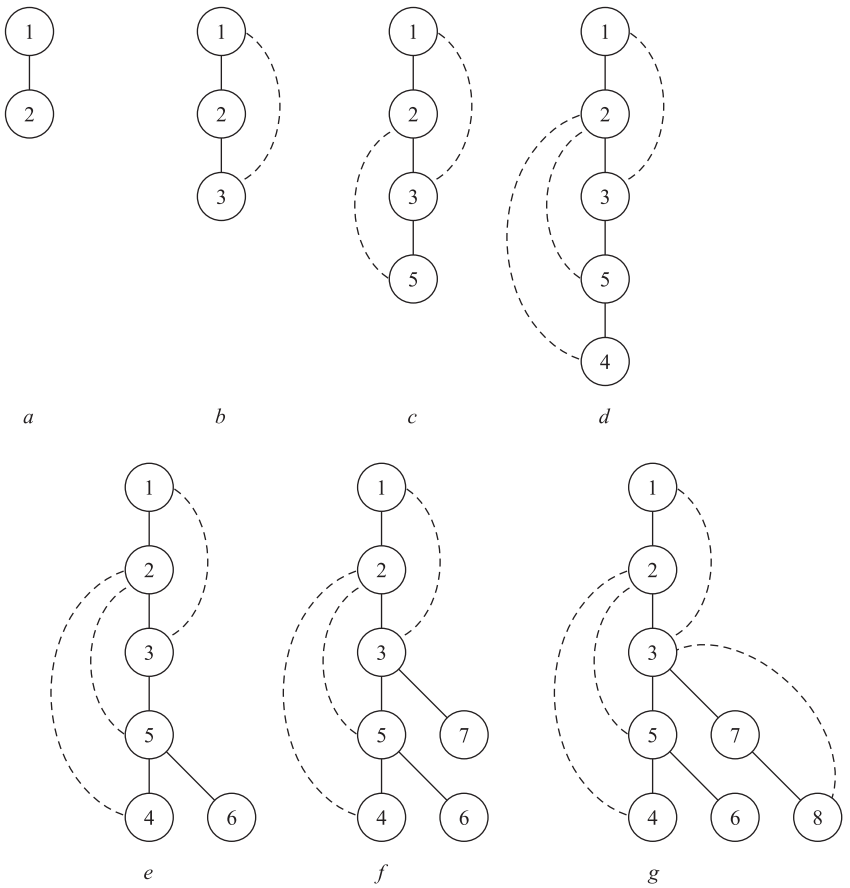


Рис. 3.5. Построение дерева поиска в глубину T для графа на рис. 3.2; фазы (a)–(g) обозначают порядок обнаружения узлов в ходе выполнения. Сплошными линиями обозначены ребра T, а пунктирными — ребра G, не принадлежащие T

Чтобы обосновать это утверждение, следует отметить следующее свойство алгоритма DFS и дерева, которое он строит.

(3.6) Для заданного рекурсивного вызова $DFS(u)$ все узлы, помеченные как «проверенные» между вызовом и концом рекурсивного вызова, являются потомками u в T.

Используя (3.6), мы докажем следующее утверждение:

(3.7) Пусть T — дерево поиска в глубину, x и y — узлы T, а (x, y) — ребро G, которое не является ребром T. В этом случае один из узлов — x или y — является предком другого.

Доказательство. Предположим, (x, y) — ребро G, которое не является ребром T, и без потери общности будем считать, что узел x первым обнаруживается алгоритмом DFS. При проверке ребра (x, y) в ходе выполнения $DFS(x)$ оно не добавляется в T, потому что узел y помечен как «проверенный». Так как узел y не был помечен как «проверенный» при изначальном вызове $DFS(x)$, этот узел был обнаружен между вызовом и концом рекурсивного вызова $DFS(x)$. Из (3.6) следует, что y является потомком x. ■

Набор всех компонент связности

До настоящего момента мы говорили о компоненте связности, содержащей конкретный узел s. Однако для каждого узла в графе существует своя компонента связности. Какими отношениями связаны эти компоненты?

Как выясняется, эти отношения четко структурированы и описываются следующим утверждением.

(3.8) Для любых двух узлов s и t в графе их компоненты связности либо идентичны, либо не пересекаются.

Это утверждение интуитивно понятно, если взглянуть на граф наподобие изображенного на рис. 3.2. Граф разделен на несколько частей, не соединенных ребрами; самая большая часть — компонента связности узлов 1–8, средняя — компонента связности узлов 11, 12 и 13 и меньшая — компонента связности узлов 9 и 10. Чтобы доказать это общее утверждение, достаточно показать, как точно определяются эти «части» для произвольного графа.

Доказательство. Возьмем любые два узла s и t в графе G, между которыми существует путь. Утверждается, что компоненты связности, содержащие s и t, представляют собой одно и то же множество. Действительно, для любого узла v в компоненте s узел v также должен быть достижим из t через путь: мы можем просто перейти из t в s, а затем из s в v. Аналогичные рассуждения работают при смене ролей s и t, так что узел входит в компоненту одного в том, и только том случае, если он также входит в компоненту другого.

С другой стороны, если пути между s и t не существует, не может быть узла v, входящего в компоненту связности каждого из них. Если бы такой узел v существовал, то мы могли бы перейти от s к v, а затем к t, строя путь между s и t. Следовательно, если пути между s и t не существует, то их компоненты связности изолированы. ■

Из этого доказательства вытекает естественный алгоритм получения всех компонент связности графа, основанный на расширении одной компоненты за раз. Алгоритм начинает с произвольного узла s, а затем использует BFS (или DFS) для генерирования его компоненты связности. Далее мы находим узел v (если он существует), который не был посещен при поиске от s, и итеративным методом,

используя BFS от узла v, генерирует компоненту связности — которая, согласно (3.8), изолирована от компоненты s. Это продолжается до тех пор, пока алгоритм не посетит все узлы.

3.3. Реализация перебора графа с использованием очередей и стеков

До настоящего момента мы рассматривали базовые алгоритмические примитивы для работы с графами без упоминания каких-либо подробностей реализации. В этом разделе мы расскажем, как использовать списки и массивы для представления графов, и обсудим достоинства и недостатки разных представлений. Затем мы воспользуемся этими структурами данных для эффективной реализации алгоритмов в ширину (BFS) и в глубину (DFS) с обходом графа. Вы увидите, что алгоритмы BFS и DFS, по сути, различаются только тем, что один использует очередь, а другой — стек, — две простые структуры данных, которые будут описаны позднее в этом разделе.

Представление графов

Существуют два основных способа представления графов: *матрица смежности* и *список смежности*. В этой книге будет использоваться представление списка смежности. Тем не менее начнем мы с обзора этих двух представлений и обсудим их достоинства и недостатки.

Граф $G = (V, E)$ имеет два естественных параметра: количество узлов $|V|$ и количество ребер $|E|$. Для их обозначения мы будем использовать запись $n = |V|$ и $m = |E|$. Время выполнения будет формулироваться в зависимости от этих двух параметров. Как обычно, мы будем ориентироваться на полиномиальное время выполнения, желательно с низкой степенью. Но когда время выполнения зависит от двух параметров, сравнение не всегда однозначно. Какое время лучше — $O(m^2)$ или $O(n^3)$? Это зависит от отношений между n и m. Если любая пара узлов соединяется не более чем одним ребром, количество ребер m не превышает $\binom{n}{2} \leq n^2$. С другой стороны, во многих практических ситуациях рассматриваются связные графы, а согласно (3.1) связный граф должен иметь не менее $m \geq n - 1$ ребер. Однако из этих сравнений не всегда очевидно, какой из двух вариантов времени выполнения (например, m^2 или n^3) лучше, поэтому мы будем анализировать время выполнения с учетом обоих параметров. В этом разделе будет реализован базовый алгоритм поиска в графе с временем $O(m + n)$. Это время будет называться *линейным*, потому что время $O(m + n)$ необходимо просто для чтения ввода. Учтите, что при работе со связными графами время выполнения $O(m + n)$ эквивалентно $O(m)$, потому что $m \geq n - 1$.

Возьмем граф $G = (V, E)$ с n узлами, множество которых обозначается $V = \{1, ..., n\}$. В простейшем способе представления графа используется *матрица смежности*. Это матрица A размером $n \times n$, в которой элемент $A[u, v]$ равен 1, если граф содержит ребро (u, v), или 0 в противном случае. Для ненаправленного графа матрица A симметрична, а $A[u, v] = A[v, u]$ для всех узлов $u, v \in V$. С представлением матрицы смежности наличие в графе заданного ребра (u, v) проверяется за время $O(1)$. Тем не менее такое представление имеет два основных недостатка.

- Затраты памяти составляют $\Theta(n^2)$. Если количество ребер в графе существенно меньше n^2, возможны более компактные представления.
- Многие алгоритмы графов требуют проверки всех ребер, инцидентных заданному узлу v. В представлении матрицы смежности для этого требуется перебрать все остальные узлы w и проверить элемент матрицы $A[v, w]$, чтобы узнать, существует ли ребро (v, w), а эта проверка занимает время $\Theta(n)$. В худшем случае v может иметь $\Theta(n)$ инцидентных ребер, тогда проверка всех ребер будет выполняться за время $\Theta(n)$ независимо от представления. Но на практике во многих графах количество инцидентных ребер для большинства узлов существенно меньше, поэтому было бы полезно иметь возможность более эффективно искать инцидентные ребра.

Для представления графов в книге используется *список смежности*, который лучше подходит для разреженных графов, то есть графов, у которых количество ребер существенно меньше n^2. В представлении списка смежности для каждого узла v создается запись со списком всех узлов, с которыми узел v соединен ребрами. А если говорить точнее, создается массив `Adj`, в котором `Adj`$[v]$ — запись со списком всех узлов, смежных с узлом v. Для ненаправленного графа $G = (V, E)$ каждое ребро $e = (v, w) \in E$ присутствует в двух списках смежности: узел w присутствует в списке узла v, а узел v присутствует в списке узла w.

Сравним представления матрицы смежности и списка смежности. Начнем с пространства, необходимого для представления. Матрица смежности имеет размеры $n \times n$, а следовательно, занимает пространство $O(n^2)$. С другой стороны, мы утверждаем, что для хранения списка смежности достаточно пространства $O(m + n)$. Ниже приводится обоснование.

Для начала нам понадобится массив указателей длины n для организации хранения списков в `Adj`, а также место для самих списков. Длины списков могут отличаться от узла к узлу, но, как указано в предыдущем абзаце, каждое ребро $e = (v, w)$ входит ровно в два списка: для v и для w. Таким образом, общая длина всех списков составляет $2m = O(m)$.

Есть и другой (по сути, эквивалентный) способ обоснования этой границы. *Степенью* n_v узла v называется количество ребер, инцидентных этому узлу. Длина списка в позиции `Adj`$[v]$ равна n_v, так что общая длина по всем узлам равна $O\left(\sum_{v \in V} n_v\right)$. Сумма степеней графа — характеристика, часто встречающаяся при анализе алгоритмов графов, поэтому полезно выяснить, чему она равна.

(3.9) $\sum_{v \in V} n_v = 2m$.

Доказательство. Каждое ребро $e = (v, w)$ вносит вклад в эту сумму ровно два раза: один раз в величине n_v, и один раз в величине n_w. Так как в сумме учитываются вклады всех ребер, она составит $2m$. ■

Сравнение матриц смежности со списками смежности можно резюмировать следующим образом.

(3.10) Представление графа в виде матрицы смежности требует пространства $O(n^2)$, а для представления в виде списка смежности достаточно пространства $O(m + n)$.

Так как мы уже выяснили, что $m \leq n^2$, ограничение $O(m + n)$ никогда не бывает хуже $O(n^2)$, и оно намного лучше для разреженных графов, в которых m намного меньше n^2.

Теперь нужно разобраться с простотой обращения к информации, хранящейся в этих двух представлениях. Вспомните, что в матрице смежности проверка присутствия в графе конкретного ребра (u, v) выполняется за время $O(1)$. В списке смежности проверка может занимать время, пропорциональное степени $O(n_v)$: чтобы узнать, входит ли ребро v в список, нужно пройти по указателям списка смежности u. С другой стороны, если алгоритм в настоящее время проверяет узел u, он может прочитать список соседей с постоянными затратами времени на каждого соседа.

С учетом сказанного список смежности является естественным представлением для изучения графов. Если алгоритм в настоящее время рассматривает узел u, он может прочитать список соседей с постоянным временем на каждого соседа; переместиться к соседу v после его обнаружения в списке за постоянное время; и быть готовым к чтению списка, связанного с узлом v. Таким образом, списковое представление соответствует физическому «изучению» графа, в котором при переходе к узлу u вы узнаете его соседей и можете прочитать их с постоянным временем для каждого соседа.

Очереди и стеки

Во многих алгоритмах имеется внутренняя фаза, в которой они должны обработать множество элементов: например, множество всех ребер, инцидентных узлу графа, множество проверенных узлов в алгоритмах BFS и DFS или множество всех свободных мужчин в алгоритме устойчивых паросочетаний. Для этой цели естественно использовать для множества элементов представление связного списка, как это было сделано для множества свободных мужчин в алгоритме устойчивых паросочетаний.

Одним из важных аспектов такого представления является порядок рассмотрения элементов в таком списке. В алгоритме устойчивых паросочетаний порядок рассмотрения свободных мужчин не влиял на результат, хотя этот факт потребовал достаточно неочевидных доказательств. Во многих других алгоритмах (таких, как DFS и BFS) порядок рассмотрения элементов критичен.

Два простейших и самых естественных варианта организации множеств элементов — очередь и стек. Элементы *очереди* извлекаются в порядке «первым пришел, первым вышел» (FIFO, First-In, First-Out), то есть в том же порядке, в котором они добавлялись. Элементы *стека* извлекаются в порядке «последним пришел, первым вышел» (LIFO, Last-In, First-Out): при каждом извлечении выбирается элемент, который был добавлен последним. И очереди и стеки легко реализуются на базе двусвязного списка. В обоих случаях всегда выбирается первый элемент списка; отличается позиция вставки нового элемента. В очереди новый элемент добавляется в конец списка в последнюю позицию, а в стеке он размещается в первой позиции списка. Помните, что в двусвязном списке хранятся указатели `First` и `Last` на начало и на конец списка соответственно, поэтому в обоих случаях вставка выполняется за постоянное время.

В следующем разделе будет показано, как реализовать алгоритмы поиска из предыдущего раздела за линейное время. Вы увидите, что алгоритм BFS может рассматриваться так, словно следующий узел выбирается из очереди, тогда как алгоритм DFS фактически предполагает стековую реализацию.

Реализация поиска в ширину

Структура данных списка смежности идеально подходит для реализации поиска в ширину. Алгоритм поочередно проверяет ребра, выходящие из заданного узла. Когда мы проверяем ребра, выходящие из u, и добираемся до ребра (u, v), необходимо знать, был ли узел v обнаружен ранее в ходе поиска. Для упрощения этой задачи создается массив `Discovered` длины n, а при первом обнаружении v в процессе поиска присваивается значение `Discovered`$[v]$ = *true*. Алгоритм, описанный в предыдущем разделе, строит из узлов уровни $L_1, L_2, ...$, где L_i — множество узлов, находящихся на расстоянии i от источника s. Для хранения узлов уровня L_i создается список $L[i]$ для всех $i = 0, 1, 2, ...$

```
BFS(s):
   Присвоить Discovered[s] = true и Discovered[v] = false для остальных v
   Инициализировать L[0] одним элементом s
   Присвоить значение счетчика уровней i = 0
   Присвоить текущее дерево BFS T = Ø
   Пока элемент L[i] не пуст
      инициализировать пустой список L[i + 1]
      Для каждого узла u ∈ L[i]
         Рассмотреть каждое ребро (u, v), инцидентное u
         Если Discovered[v]=false
            Присвоить Discovered[v]=true
            Добавить ребро (u, v) в дерево T
            Добавить v в список L[i + 1]
         Конец Если
      Конец цикла
      Увеличить счетчик уровней i на 1
   Конец Пока
```

В этой реализации несущественно, хранится каждый список $L[i]$ в формате очереди или стека, поскольку алгоритму разрешено рассматривать узлы уровня L_i в произвольном порядке.

(3.11) Приведенная выше реализация алгоритма BFS выполняется за время $O(m + n)$ (то есть в линейной зависимости от размера ввода), если граф описывается представлением списка смежности.

Доказательство. Для начала легко продемонстрировать для времени выполнения алгоритма ограничение $O(n^2)$ (более слабая граница, чем заявленная $O(m + n)$). Чтобы убедиться в этом, достаточно заметить, что достаточно создать максимум n списков $L[i]$, а эта операция выполняется за время $O(n)$. Теперь необходимо рассмотреть узлы u этих списков. Каждый узел входит не более чем в один список, поэтому цикл будет выполнен не более n раз по всем итерациям цикла `Пока`. Рассматривая узел u, необходимо просмотреть все ребра (u, v), инцидентные u. Таких ребер не больше n, и на каждое ребро тратится время $O(1)$. Итак, общее время, затраченное на одну итерацию внутреннего цикла, не более $O(n)$. Соответственно мы заключаем, что внутренний цикл состоит не более чем из n итераций, а каждая итерация выполняется за время не более $O(n)$, так что общее время не превышает $O(n^2)$.

Чтобы прийти к усиленной границе $O(m + n)$, следует увидеть, что внутренний цикл, обрабатывающий узел u, может выполняться за время меньше $O(n)$ при небольшом количестве соседей у u. Как и прежде, n_u обозначает степень узла u, то есть количество ребер, инцидентных u. Время, потраченное во внутреннем цикле, в котором проверяются ребра, инцидентные u, составляет $O(n_u)$, так что суммарное время по всем узлам составляет $O\left(\sum_{u \in V} n_u\right)$. Вспомните из (3.9), что $\sum_{u \in V} n_u = 2m$, а общее время, потраченное на проверку ребер для всего алгоритма, составляет $O(m)$. Дополнительное время $O(n)$ понадобится для подготовки списков и управления массивом `Discovered`. Таким образом, общие затраты времени составляют $O(m + n)$, как и утверждалось выше. ■

В нашем описании алгоритма упоминались n разных списков $L[i]$ для каждого уровня L_i. Вместо нескольких разных списков можно реализовать алгоритм с одним списком L, который организован в формате очереди. В этом случае алгоритм обрабатывает узлы в том порядке, в котором они были изначально обнаружены; каждый обнаруженный новый узел добавляется в конец очереди, а алгоритм всегда обрабатывает ребра, выходящие из узла, который в настоящее время является первым в очереди.

Если хранить обнаруженные узлы в этом порядке, то все узлы уровня L_i будут находиться в очереди до всех узлов уровня L_{i+1} для $i = 0, 1, 2...$ Следовательно, все узлы уровня L_i образуют непрерывную последовательность, за которой следуют все узлы уровня L_{i+1}, и т. д. Из этого вытекает, что такая реализация на базе одной очереди приводит к тому же результату, что и описанная ранее реализация BFS.

Реализация поиска в глубину

А теперь рассмотрим алгоритм поиска в глубину. В предыдущем разделе алгоритм DFS был представлен в виде рекурсивной процедуры — естественный способ ее определения. Однако он также может рассматриваться как почти полный аналог алгоритма BFS, с тем различием, что обрабатываемые узлы организуются в стек, а не в очередь. По сути, рекурсивная структура DFS может рассматриваться как механизм занесения узлов в стек для последующей обработки с первоочередной обработкой последних обнаруженных узлов. А сейчас будет показано, как реализовать алгоритм DFS на базе стека узлов, ожидающих обработки.

В обоих алгоритмах, BFS и DFS, существуют различия между *обнаружением* узла v (когда алгоритм впервые видит узел при обнаружении ребра, ведущего к v) и *проверкой* узла v (когда перебираются все ребра, инцидентные v, что может привести к обнаружению дальнейших узлов). Различия между BFS и DFS проявляются в особенностях чередования обнаружений и проверок.

В алгоритме BFS, приступая к проверке узла u в уровне L_i, мы добавляли всех его вновь обнаруженных соседей в следующий уровень L_{i+1}, а непосредственная проверка этих соседей откладывалась до перехода к обработке уровня L_{i+1}. С другой стороны, алгоритм DFS действует более «импульсивно»: при проверке узла u он перебирает соседей u, пока не найдет первый непроверенный узел v (если он есть), после чего немедленно переключается на проверку v.

Чтобы реализовать стратегию проверки DFS, мы сначала добавляем все узлы, смежные с u, в список рассматриваемых узлов, но после этого переходим к проверке v — нового соседа u. В процессе проверки v соседи v последовательно добавляются в хранимый список, но добавление происходит в порядке стека, так что эти соседи будут проверены до возвращения к проверке других соседей u. К другим узлам, смежным с u, алгоритм возвращается только тогда, когда не останется иных узлов.

Также в этой реализации используется массив `Explored`, аналогичный массиву `Discovered` из реализации BFS. Различие заключается в том, что `Explored`[v] присваивается значение `true` только после перебора ребер, инцидентных v (когда поиск DFS находится в точке v), тогда как алгоритм BFS присваивает `Discovered`[v] значение `true` при первом обнаружении v. Полная реализация приведена ниже.

```
DFS(s):
   Инициализировать S как стек с одним элементом s
   Пока стек S не пуст
      Получить узел u из S
      Если Explored[u]=false
         Присвоить Explored[u]=true
         Для каждого ребра (u, v), инцидентного u
            Добавить v в стек S
         Конец цикла
      Конец Если
   Конец Пока
```

Осталось упомянуть об одном последнем недочете: процедура поиска в глубину недостаточно четко определена, потому что список смежности проверяемого узла может обрабатываться в любом порядке. Так как приведенный выше алгоритм заносит все смежные узлы в стек до их рассмотрения, он фактически обрабатывает каждый список смежности в порядке, обратном порядку рекурсивной версии DFS из предыдущего раздела.

(3.12) Приведенный выше алгоритм реализует поиск DFS в том смысле, что он посещает узлы точно в таком же порядке, как и рекурсивная процедура DFS из предыдущего раздела (не считая того, что списки смежности обрабатываются в обратном порядке). Если мы хотим, чтобы алгоритм также строил дерево DFS, для каждого узла u в стеке S также должна храниться информация об узле, который «привел» к включению u в стек. Задача легко решается при помощи массива `parent`: при каждом добавлении узла v в стек из-за ребра (u, v) выполняется присваивание `parent`$[v] = u$. Когда узел $u \neq s$ помечается как «проверенный», ребро (u,`parent`$[u]$) также можно добавить в дерево T. Учтите, что узел v может включаться в стек S многократно, поскольку он может быть смежным с несколькими проверяемыми узлами, и для каждого такого узла в стек S добавляется копия v. Однако для проверки узла v будет использоваться только одна из этих копий — та, которая была добавлена последней. В результате для каждого узла v достаточно хранить только одно значение `parent`$[v]$, просто перезаписывая значение `parent`$[v]$ при каждом добавлении в стек S новой копии v.

Основной фазой алгоритма является добавление и удаление узлов из стека S, которое выполняется за время $O(1)$. Следовательно, чтобы определить границу времени выполнения, необходимо оценить количество таких операций. Чтобы подсчитать количество операций со стеком, достаточно подсчитать количество узлов, добавленных в S, так как для каждого удаления из S узел должен быть сначала добавлен.

Сколько элементов будет добавлено в S? Как и прежде, пусть n_v — степень узла v. Узел v будет добавляться в стек S каждый раз, когда проверяется один из n_v его смежных узлов, так что общее количество узлов, добавленное в S, не превышает $\sum_u n_v = 2m$. Это доказывает искомую оценку $O(m + N)$ для времени выполнения DFS.

(3.13) Для графа, заданного представлением списка смежности, приведенная выше реализация алгоритма DFS выполняется за время $O(m + n)$ (то есть в линейной зависимости от размера входных данных).

Определение всех компонент связности

В предыдущем разделе говорилось о том, как использовать алгоритм BFS (или DFS) для поиска всех компонент связности в графе. Вы начинаете с произвольного узла s и используете BFS (или DFS) для построения его компоненты связности. Затем находится узел v (если он есть), который не был посещен при поиске от s, и алгоритм BFS (или DFS) начинает от узла v и генерирует его компоненту связ-

ности, — которая, согласно (3.8), будет изолирована от компоненты s. Этот процесс продолжается до тех пор, пока не будут посещены все узлы.

Хотя ранее время выполнения BFS и DFS выражалось в виде $O(m + n)$, где m и n — общее количество ребер и узлов в графе, фактически оба вида поиска — и BFS и DFS — работают только с ребрами и узлами компоненты связности, содержащей начальный узел. (Другие ребра и узлы им не видны.) Таким образом, приведенный выше алгоритм, хотя он и может выполнять BFS и DFS несколько раз, выполняет постоянный объем работы для каждого конкретного узла или ребра в итерации для той компоненты связности, которой этот узел или ребро принадлежит. А следовательно, общее время выполнения алгоритма по-прежнему остается равным $O(m + n)$.

3.4. Проверка двудольности: практическое применение поиска в ширину

Вспомним определение двудольного графа: так называется граф, множество узлов V которого может быть разбито на такие подмножества X и Y, что один конец каждого ребра принадлежит X, а другой конец принадлежит Y. Чтобы обсуждение было более наглядным, представьте, что узлы множества X окрашены в красный цвет, а узлы множества Y — в синий. Тогда можно сказать, что граф является двудольным, если его узлы можно раскрасить в красный и синий цвет так, чтобы у каждого ребра один конец был красным, а другой синим.

Задача

В предыдущих главах приводились примеры двудольных графов. Для начала зададимся вопросом: существуют ли естественные примеры недвудольных графов, то есть графов, для которых такое разбиение V невозможно?

Разумеется, треугольник не может быть двудольным: первый узел окрашивается в красный цвет, другой в синий, а с третьим уже ничего не сделать. В более общем смысле возьмем цикл C нечетной длины с пронумерованными узлами $1, 2, 3, \ldots, 2k, 2k + 1$. Если раскрасить узел 1 в красный цвет, то узел 2 должен быть окрашен в синий цвет, после чего узел 3 окрашивается в красный и т. д.; нечетные узлы окрашиваются в красный цвет, а четные в синий. Но тогда узел $2k + 1$ должен быть красным, и от него ведет ребро к узлу 1 — тоже красному. Следовательно, разбить C на красные и синие узлы так, как требуется по условию задачи, невозможно. Обобщая далее, мы видим, что если граф G просто содержит нечетный цикл, можно применить тот же аргумент; следовательно, мы приходим к следующему утверждению.

(3.14) Если граф G является двудольным, он не может содержать нечетные циклы.

Легко проверить, что граф является двудольным, если соответствующие множества X и Y (то есть красные и синие узлы) уже были определены за нас; и во многих ситуациях, в которых встречаются двудольные графы, это естественно. Но предположим, имеется граф G без информации о разбиении, и нам хотелось бы определить, является ли граф двусторонним, то есть существует ли требуемое разбиение на красные и синие узлы. Насколько сложна эта задача? Из (3.14) видно, что одним из простых препятствий к двудольности графа является наличие нечетных циклов. Существуют ли другие, более сложные препятствия?

Проектирование алгоритма

На самом деле существует очень простая процедура проверки двудольности, а из ее анализа следует, что других препятствий, кроме нечетных циклов, нет. Сначала предположим, что граф G является связным, поскольку в противном случае мы могли бы сначала вычислить его компоненты связности и проанализировать каждую из них по отдельности. Затем мы выбираем любой узел $s \in V$ и окрашиваем его в красный цвет; никакой потери общности при этом не происходит, так как s все равно должен быть связан с каким-то цветом. Все соседи s должны быть окрашены в синий цвет, мы так и поступаем. Все соседи этих узлов окрашиваются в красный цвет, их соседи в синий и т. д., пока не будет раскрашен весь граф. В этот момент у нас либо появляется действительная красно-синяя окраска G, в которой каждое ребро имеет разноцветные концы, либо у какого-то ребра концы имеют одинаковый цвет. Похоже, в последнем случае ничего сделать было нельзя: просто граф G не является двудольным. Теперь это обстоятельство нужно обосновать более формально, а также выработать эффективный способ раскраски.

Прежде всего следует заметить, что описанная процедура окрашивания по сути идентична описанию BFS: мы перемещаемся вперед от s, раскрашивая узлы, когда они впервые попадаются в процессе перебора. Более того, алгоритм окрашивания можно описать следующим образом: мы выполняем поиск BFS, узел s окрашивается в красный цвет, все узлы уровня L_1 — в синий, все узлы уровня L_2 в красный и так далее. Нечетные уровни окрашиваются в синий цвет, а четные — в красный.

Чтобы реализовать это описание на базе BFS, достаточно взять реализацию BFS и добавить дополнительный массив `Color`. При каждом шаге BFS, на котором узел v добавляется в список $L[i + 1]$, выполняется присваивание `Color`$[v]$ = `red`, если $i + 1$ является четным числом, или `Color`$[v]$ = `blue`, если $i + 1$ нечетное. В конце этой процедуры остается перебрать все ребра и определить, не был ли присвоен обоим концам одинаковый цвет. Следовательно, общее время выполнения алгоритма окрашивания составляет $O(m + n)$, как и для алгоритма BFS.

Анализ алгоритма

Теперь можно доказать утверждение о том, что этот алгоритм правильно определяет, является ли граф двудольным. Также оно показывает, что если граф G не является двудольным, в нем можно найти нечетный цикл.

(3.15) Пусть G — связный граф, а L_1, L_2, ... — уровни, построенные алгоритмом BFS, начиная с узла s. В этом случае должно выполняться ровно одно из следующих двух условий.

(i) В G не существует ребра, соединяющего два узла одного уровня. В таком случае G является двудольным графом, в котором узлы четных уровней могут быть окрашены в красный цвет, а узлы нечетных уровней — в синий цвет.

(ii) В G существует ребро, соединяющее два узла одного уровня. В этом случае G содержит цикл нечетной длины, а следовательно, не может быть двудольным.

Доказательство. Сначала рассмотрим случай (i): предположим, не существует ребра, соединяющего два узла одного уровня. В соответствии с (3.4) мы знаем, что каждое ребро G соединяет узлы либо одного уровня, либо смежных уровней. Предположение для случая (i) заключается в том, что первая из двух вариантов альтернатив не встречается, то есть каждое ребро соединяет два узла смежных уровней. Но наша процедура окрашивания назначает смежным уровням противоположные цвета, поэтому каждое ребро имеет разноцветные концы. Следовательно, описанная схема окрашивания обеспечивает двудольность G.

Теперь допустим, что выполняется условие (ii); почему граф G обязан содержать нечетный цикл? Известно, что G содержит ребро, соединяющее два узла одного уровня. Допустим, это ребро $e = (x, y)$, при этом $x, y \in L_j$. Кроме того, для удобства записи будем считать, что L_0 («уровень 0») представляет собой множество, состоящее только из s. Теперь возьмем дерево BFS T, построенное нашим алгоритмом, и обозначим z узел наивысшего возможного уровня, для которого z является предком как x, так и y в дереве T; по очевидным причинам z можно назвать *низшим общим предком* x и y. Допустим, $z \in L_i$, где $i < j$. Возникает ситуация, изображенная на рис. 3.6. Рассмотрим цикл C, определяемый переходом по пути z-x в T, затем по ребру e и затем по пути y-z в T. Длина этого цикла, вычисляемая суммированием длин трех частей, равна $(j - i) + 1 + (j - i)$; получается $2(j - i) + 1$, то есть нечетное число. ■

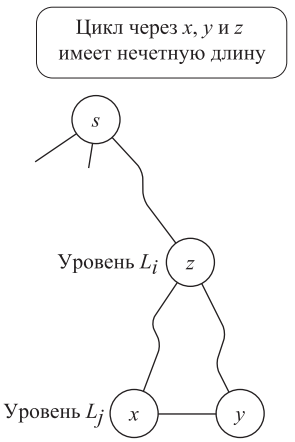


Рис. 3.6. Если два одноуровневых узла x и y соединены ребром, то цикл через x, y и их низшего общего предка z имеет нечетную длину; это показывает, что граф не может быть двудольным

3.5. Связность в направленных графах

До настоящего момента мы рассматривали задачи, относящиеся к ненаправленным графам; теперь посмотрим, в какой степени эти идеи применимы к направленным графам.

Напомним, что в направленном графе ребро (u, v) имеет направление: оно проходит из u в v. В этом случае отношение между u и v является асимметричным, что имеет качественные последствия для структуры полученного графа. Например, в разделе 3.1 Всемирная паутина упоминалась как экземпляр большого, сложного направленного графа, узлами которого являются страницы, а ребрами — гиперссылки. Процесс веб-серфинга может быть представлен последовательностью ребер этого направленного графа; направленность важна, потому что в общем случае переход по гиперссылкам в обратном направлении невозможен.

В то же время многие базовые определения и алгоритмы имеют естественные аналоги в направленных графах. Например, к их числу относится представление списка смежности и алгоритмы поиска в графах, такие как BFS и DFS.

Представление направленных графов

Для представления направленных графов в контексте проектирования алгоритмов используется разновидность представления списка смежности, использовавшегося для ненаправленных графов. Вместо одного списка соседей в каждом узле хранятся два списка: один состоит из узлов, *к которым* ведут ребра от данного узла, а второй — из узлов, *из которых* выходят ребра к данному узлу. Таким образом, алгоритм, просматривающий узел u, может получить информацию как об узлах, достижимых при переходе на один шаг вперед по направленному ребру, так и об узлах, которые были бы достижимы при возврате на один шаг в обратном направлении (по ребру, ведущему к u).

Алгоритмы поиска

Алгоритмы поиска в ширину и в глубину в направленных графах почти не отличаются от аналогичных алгоритмов для ненаправленных графов. В этом разделе мы займемся BFS. Алгоритм начинает с узла s, определяет первый уровень из всех узлов, к которым ведет ребро из s, затем определяет второй уровень из всех узлов, к которым ведут ребра из узлов первого уровня, и т. д. При таком подходе узлы будут обнаруживаться уровень за уровнем в ходе распространения поиска от s, а уровень j будет состоять из узлов, для которых кратчайший путь из s содержит ровно j ребер. Как и в ненаправленном графе, этот алгоритм выполняет не более чем постоянный объем работы для каждого узла и ребра и поэтому работает за время $O(m + n)$.

Важно понимать, что именно вычисляет направленная версия BFS. В направленных графах путь от узла s к t может существовать даже в том случае, если путь от t к s не существует; а направленная версия BFS вычисляет множество всех

узлов t, обладающих тем свойством, что от s существует путь к t. От таких узлов могут существовать пути обратно к s, а могут и не существовать.

Для поиска в глубину тоже существует естественная аналогия, которая выполняется за линейное время и вычисляет то же множество узлов. В этом случае также используется рекурсивная процедура, которая пытается исследовать граф на максимально возможную глубину (на этот раз только по ребрам с соответствующим направлением). Когда DFS находится в узле u, он по порядку запускает рекурсивный поиск в глубину для каждого узла, к которому ведет ребро из u.

Допустим, для заданного узла s требуется получить множество узлов, от которых существуют пути к s (вместо узлов, к которым ведут пути из s). Это можно легко сделать, определив новый направленный граф G^{rev}, который получается из G простым изменением направления каждого ребра. После этого алгоритм BFS или DFS применяется к G^{rev}; путь из s к узлу существует в G^{rev} в том, и только в том случае, если в G существует путь из него в s.

Сильная связность

Вспомните, что направленный граф называется *сильно связным*, если для каждых двух узлов u и v существует путь из u в v и путь из v в u. Также полезно сформулировать некоторые термины для свойства, лежащего в основе этого определения; два узла u и v в направленном графе называются *взаимодостижимыми*, если существует путь из u в v и путь из v в u. (Таким образом, граф является сильно связным, если каждая пара узлов в нем взаимодостижима.)

Взаимодостижимость обладает рядом полезных свойств, многие из которых вытекают из следующего простого факта.

(3.16) Если узлы u и v являются взаимодостижимыми и узлы v и w являются взаимодостижимыми, то узлы u и w также являются взаимодостижимыми.

Доказательство. Чтобы построить путь от u к w, мы сначала перейдем от u к v (по пути, существование которого гарантируется взаимодостижимостью u и v), а затем от v к w (по пути, существование которого гарантируется взаимодостижимостью v и w). Для построения пути от w к u те же рассуждения применяются в обратном направлении: мы сначала перейдем от w к v (по пути, существование которого гарантируется взаимодостижимостью v и w), а затем от v к u (по пути, существование которого гарантируется взаимодостижимостью u и v).

Для проверки сильной связности направленного графа существует простой алгоритм с линейным временем выполнения, неявно базирующийся на (3.16). Мы выбираем любой узел s и проводим поиск BFS в G, начиная с s. Затем BFS выполняется от s в G^{rev}. Если хотя бы один из двух поисков не найдет все узлы, то очевидно, что G не является сильно связным. Но допустим, мы выяснили, что из s существует путь в каждый другой узел, и из каждого другого узла существует путь s. В этом случае s и v взаимодостижимы для каждого v, из чего можно сделать вывод о взаимодостижимости любых двух узлов u и v: s и u взаимодостижимы, s и v взаимодостижимы, и из (3.16) следует, что u и v также взаимодостижимы. ■

По аналогии с компонентами связности в ненаправленных графах мы можем определить сильную компоненту, содержащую узел s, для направленного графа как множество всех узлов v, для которых s и v является взаимодостижимыми. Если задуматься, алгоритм из предыдущего абзаца в действительности вычисляет сильную компоненту, содержащую s: мы выполняем BFS, начиная с s, в G и G^{rev}; множество узлов, достигнутых при обоих поисках, представляет собой множество узлов с путями в s и из s; следовательно, это множество является сильной компонентой, содержащей s.

На этом сходство между понятиями компоненты связности в ненаправленных графах и сильной компоненты в направленных графах не ограничивается. Вспомните, что компоненты связности образуют естественное разбиение графа, поскольку каждые две компоненты либо идентичны, либо изолированы. Сильные компоненты также обладают этим свойством, причем фактически по той же причине, следующей из (3.16).

(3.17) Для каждых двух узлов s и t в направленном графе их сильные компоненты либо идентичны, либо изолированны.

Доказательство. Возьмем два любых взаимодостижимых узла s и t; утверждается, что сильные компоненты, содержащие s и t, идентичны. В самом деле, для каждого узла v, если s и v взаимодостижимы, то, согласно (3.16), t и v также взаимодостижимы. Аналогичным образом, если t и v взаимодостижимы, то, согласно (3.16), s и v также взаимодостижимы.

С другой стороны, если s и t не являются взаимодостижимыми, то не может быть узла v, входящего в сильную компоненту каждого из этих узлов. Если бы такой узел v существовал, то узлы s и v были бы взаимодостижимыми, узлы v и t были бы взаимодостижимыми, поэтому из (3.16) следовало бы, что s и t также являются взаимодостижимыми.

Хотя подробное обоснование здесь не приводится, при некоторой дополнительной работе возможно вычислить сильные компоненты всех узлов за общее время $O(m + n)$. ■

3.6. Направленные ациклические графы и топологическое упорядочение

Если ненаправленный граф не содержит циклов, то его структура чрезвычайно проста: каждая из его компонент связности представляет собой дерево. Однако направленный граф может и не иметь (направленных) циклов, обладая при этом довольно богатой структурой. Например, такие графы могут иметь большое количество ребер: если начать с множества узлов $\{1,2, ..., n\}$ и включать ребро (i, j) для всех $i < j$, то полученный направленный граф состоит из $\binom{n}{2}$ ребер, но не содержит циклов.

Если направленный граф не содержит циклов, он называется (вполне естественно) *направленным ациклическим графом*, или сокращенно *DAG* (Directed Acyclic Graph). Пример направленного ациклического графа изображен на рис. 3.7, *a*, хотя чтобы убедиться в том, что он не содержит направленных циклов, придется немного потрудиться.

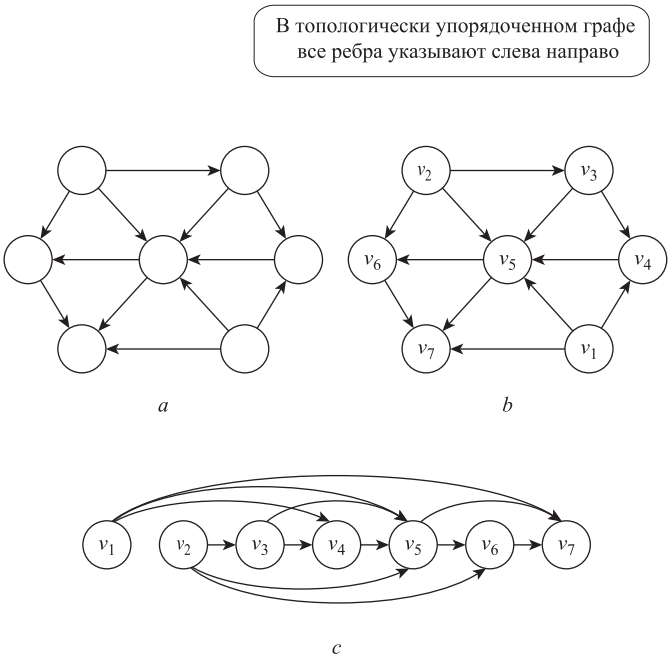


Рис. 3.7. *a* — направленный ациклический граф; *b* — тот же граф в топологическом порядке, обозначенном метками на каждом узле; *c* — другое представление того же графа, структурированное для выделения топологического порядка

Задача

Направленные ациклические графы (DAG) очень часто встречаются в теории обработки информации, потому что многие типы сетей зависимостей (см. раздел 3.1) ацикличны. Следовательно, DAG могут естественным образом использоваться для представления отношений предшествования или зависимостей. Допустим, имеется множество задач {1, 2, ..., *n*}, которые требуется выполнить; между задачами существуют зависимости — допустим, они указывают, что для пары *i* и *j* задача *i* должна быть выполнена до *j*. Скажем, задачи могут представлять учебные курсы, а некий курс можно пройти только после завершения необходимых вводных курсов. Или же задачи могут соответствовать последовательности вычислительных задач; задача *j* получает свои входные данные на выходе задачи *i*, поэтому задача *i* должна быть завершена до задачи *j*.

Для представления каждой задачи в таком взаимозависимом множестве задач создается узел, а требование о завершении i до j представляется направленным ребром (i, j). Чтобы отношения предшествования имели реальный смысл, полученный граф G должен быть направленным ациклическим графом, то есть DAG. В самом деле, если граф содержит цикл C, то никакие задачи из C выполнены быть не могут: так как каждая задача из C не может начаться до завершения другой задачи из C, ни одна из них не может быть выполнена первой.

Давайте немного разовьем это представление DAG как отношений предшествования. Для заданного множества задач с зависимостями возникает естественный вопрос о нахождении действительного порядка выполнения этих задач с соблюдением всех зависимостей, а именно: для направленного графа G *топологическим упорядочением* называется такое упорядочение его узлов $v_1, v_2, ..., v_n$, при котором для каждого ребра (v_i, v_j) выполняется условие $i < j$. Иначе говоря, все ребра в этом упорядочении указывают «вперед». Топологическое упорядочение задач описывает порядок их безопасного выполнения, то есть когда мы приходим к задаче v_j, все задачи, которые ей должны предшествовать, уже были выполнены. На рис. 3.7, *b* узлы DAG из части (а) помечены в соответствии с их топологическим упорядочением; каждое ребро действительно ведет от узла с малым индексом к узлу с более высоким индексом.

Более того, топологическое упорядочение G может рассматриваться как прямое «свидетельство» отсутствия циклов в G.

(3.18) Если в графе G существует топологическое упорядочение, то граф G является DAG.

Доказательство. Действуя от обратного, предположим, что в графе G существует топологическое упорядочение $v_1, v_2, ..., v_n$ и цикл C. Пусть v_i — узел C с наименьшим индексом, а v_j — узел C, непосредственно предшествующий v_i; следовательно, существует ребро (v_j, v_i). Но согласно нашему выбору i выполняется условие $j > i$, а это противоречит предположение о том, что $v_1, v_2, ..., v_n$ является топологическим упорядочением.

Гарантия ацикличности, предоставляемая топологическим упорядочением, может быть очень полезной даже на визуальном уровне. На рис. 3.7, *c* изображен тот же граф, что и на рис. 3.7, *a* и *b*, но с расположением узлов в соответствии с топологическим упорядочением. С первого взгляда видно, что граф (с) представляет собой DAG, потому что все ребра идут слева направо. ■

Вычисление топологического упорядочения

Основной вопрос, который нас интересует, по смыслу противоположен (3.18): существует ли топологическое упорядочение в каждом DAG, и если существует, как эффективно вычислить его? Универсальный метод получения ответа для любых DAG был бы очень полезен: он показал бы, что для произвольных отношений предшествования в множестве задач, не содержащем циклов, существует порядок выполнения этих задач, который может быть эффективно вычислен.

Проектирование и анализ алгоритма

На самом деле условие, обратное (3.18), выполняется, и для доказательства этого факта мы воспользуемся эффективным алгоритмом вычисления топологического упорядочения. Ключевым фактором станет определение отправной точки: с какого узла должно начинаться топологическое упорядочение? Такой узел v_1 не должен иметь входящих ребер, поскольку любое входящее ребро нарушило бы определяющее свойство топологического упорядочения (все ребра должны указывать «вперед»). Следовательно, нужно доказать следующий факт:

(3.19) В каждом направленном ациклическом графе G существует узел v, не имеющий входящих ребер.

Доказательство. Пусть G — направленный граф, в котором каждый узел имеет по крайней мере одно входящее ребро. Мы покажем, как найти цикл в таком графе G; тем самым утверждение будет доказано. Выбираем любой узел v и начинаем переходить по ребрам в обратном направлении от v; так как v содержит как минимум одно входящее ребро (u, v), можно перейти к u; так как ребро u содержит как минимум одно входящее ребро (x, u), можно вернуться обратно к x; и т. д. Этот процесс может продолжаться бесконечно, так как каждый обнаруженный узел имеет входящее ребро. Но после $n + 1$ шагов какой-то узел w неизбежно будет посещен дважды. Если обозначить C серию узлов, встреченных между последовательными посещениями w, то C очевидным образом образует цикл. ■

Существование такого узла v — все, что необходимо для построения топологического упорядочения G. Докажем посредством индукции, что каждый DAG имеет топологическое упорядочение. Это утверждение очевидно истинно для DAG с одним или двумя узлами. Теперь допустим, что оно истинно для DAG с количеством узлов до n. В DAG G с $n + 1$ узлом будет присутствовать узел v, не имеющий входящих ребер; его существование гарантировано по (3.19). Узел v ставится на первое место в топологическом упорядочении; такое размещение безопасно, поскольку все ребра из v указывают вперед. Теперь заметим, что граф $G - \{v\}$ является DAG, поскольку удаление v не может привести к образованию циклов, не существовавших ранее. Кроме того, граф $G - \{v\}$ содержит n узлов, что позволяет нам применить предположение индукции для получения топологического упорядочения $G - \{v\}$. Присоединим узлы $G - \{v\}$ в этом порядке после v; в полученном упорядочении G все ребра будут указывать вперед, а следовательно, оно является топологическим упорядочением.

Итак, положение, обратное (3.18), успешно доказано.

(3.20) Если граф G является DAG, то в нем существует топологическое упорядочение.

Доказательство методом индукции содержит следующий алгоритм для вычисления топологического упорядочения G.

```
Чтобы вычислить топологическое упорядочение G:
Найти узел v, не имеющий входящих ребер, и поставить его в начало.
Удалить v из G.
Рекурсивно вычислить топологическое упорядочение G – {v}
   и присоединить его после v
```

На рис. 3.8 изображена последовательность удалений узлов, происходящих при применении к графу на рис. 3.7. Закрашенные узлы на каждой итерации не имеют входящих ребер; здесь важно то, что, как гарантирует (3.19), при применении алгоритма к DAG всегда найдется хотя бы один такой узел, который может быть удален.

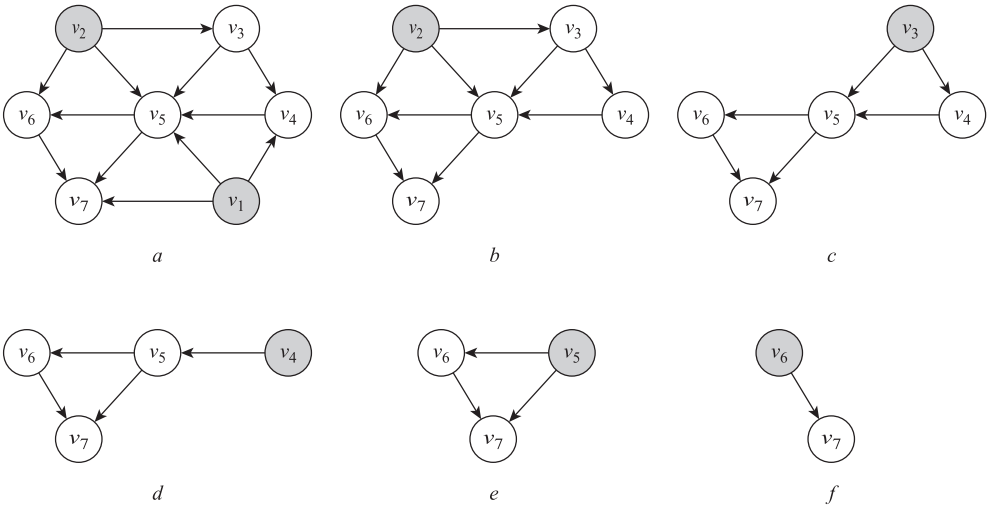


Рис. 3.8. Начиная с графа на рис. 3.7 узлы удаляются один за другим для добавления в топологическое упорядочение. Закрашенные узлы не имеют входящих ребер; следует заметить, что на каждой стадии выполнения алгоритма всегда найдется хотя бы один такой узел

Чтобы сформулировать оценку времени выполнения алгоритма, заметим, что нахождение узла v, не имеющего входящих ребер, и удаление его из G может выполняться за время $O(n)$. Так как алгоритм выполняется за n итераций, общее время выполнения составит $O(n^2)$.

Результат не так уж плох; а для сильно разреженного графа G, содержащего $\Theta(n^2)$ ребер, обеспечивается линейная зависимость от размера входных данных. Но если количество ребер m намного меньше n^2, хотелось бы чего-то большего. В таком случае время выполнения $O(m + n)$ было бы значительным улучшением по сравнению с $\Theta(n^2)$.

Оказывается, время выполнения $O(m + n)$ может быть обеспечено тем же высокоуровневым алгоритмом — итеративным удалением узлов, не имеющих входящих ребер. Просто нужно повысить эффективность поиска таких узлов, а это можно сделать так.

Узел объявляется «активным», если он еще не был удален алгоритмом, и мы будем явно хранить два вида данных:

(a) для каждого узла w — количество входящих ребер, ведущих к w от активных узлов; и

(b) множество S всех активных узлов G, не имеющих входящих ребер из других активных узлов.

В начале выполнения все узлы активны, поэтому данные (a) и (b) могут инициализироваться за один проход по узлам и ребрам. Тогда каждая итерация состоит из выбора узла v из множества S и его удаления. После удаления v перебираются все узлы w, к которым существовали ребра из v, и количество активных входящих ребер, хранимое для каждого узла w, уменьшается на 1. Если в результате количество активных входящих ребер для w уменьшается до 0, то w добавляется в множество S. Продолжая таким образом, мы постоянно следим за узлами, пригодными для удаления, осуществляя постоянный объем работы на каждое ребро в ходе выполнения всего алгоритма.

Упражнения с решениями

Упражнение с решением 1

Возьмем направленный ациклический граф G на рис. 3.9. Сколько топологических упорядочений он имеет?

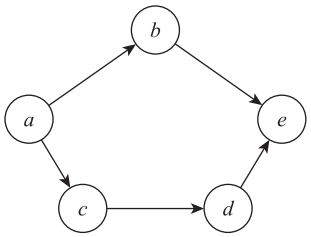


Рис. 3.9. Сколько топологических упорядочений существует у этого графа?

Решение

Вспомните, что топологическим упорядочением G называется такое упорядочение узлов v_1, v_2, ..., v_n, при котором все ребра указывают «вперед»: для каждого ребра (v_i, v_j) выполняется условие $i < j$.

Конечно, чтобы ответить на вопрос, можно перебрать все $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ возможных упорядочений и проверить, является ли каждое из них топологическим упорядочением. Впрочем, такой перебор займет много времени.

Лучше действовать иначе. Как было показано в тексте (и как следует непосредственно из определения), первым узлом топологического упорядочения должен быть узел, не имеющих входящих ребер. Аналогичным образом последним узлом должен быть узел, не имеющий выходящих ребер. Следовательно, в каждом топологическом упорядочении G узел a должен стоять на первом месте, а узел e — на последнем.

Теперь нужно понять, как должны располагаться узлы b, c и d в середине упорядочения. Ребро (c, d) требует, чтобы узел c предшествовал d, однако b может

находиться в любой позиции относительно этих двух узлов: до обоих, между c и d, или после обоих. Других вариантов нет, и мы можем заключить, что существуют три возможных топологических упорядочения:

a, b, c, d, e
a, c, b, d, e
a, c, d, b, e

Упражнение с решением 2

Ваши друзья работают над методами координации групп мобильных роботов. Каждый робот оснащен радиопередатчиком, который используется для связи с базовой станцией. Выяснилось, что если роботы оказываются слишком близко друг к другу, между передатчиками возникают помехи. Встает естественная задача: как спланировать такое перемещение роботов, чтобы каждый робот оказался у предполагаемой цели, но роботы не сближались на опасное расстояние?

Задачу можно абстрактно смоделировать следующим образом. Допустим, имеется ненаправленный граф $G = (V, E)$, предоставляющий план здания; два робота изначально расположены в узлах a и b графа. Робот в узле a должен переместиться к узлу c по пути в графе G, а робот в узле b должен переместиться к узлу d. Для этого используется механизм *планирования*: на каждом временном отрезке один из роботов перемещается по одному ребру, от ребра к соседнему узлу; в конце периода планирования робот из узла a должен находиться в узле c, а робот из узла b — в узле d.

Планирование свободно от помех, если не существует точки, в которой два робота занимают узлы на расстоянии $\leq r$ друг от друга в графе (для заданного параметра r). Предполагается, что два начальных узла a и b находятся на расстоянии, большем r (как и два конечных узла c и d).

Предложите алгоритм с полиномиальным временем, который решает, существует ли вариант планирования без помех, при котором каждый робот доберется до своей цели.

Решение

Рассматриваемая задача относится к более общей категории: имеется набор возможных конфигураций для роботов (под конфигурацией понимается выбор местонахождения каждого робота). Мы пытаемся перейти из заданной начальной конфигурации (a, b) в конечную конфигурацию (c, d), принимая во внимание ограничения на возможность перемещения между конфигурациями (робот может перейти только в соседний узел) и на то, какие конфигурации считаются «допустимыми».

Задача получается довольно сложной, если рассматривать происходящее на уровне нижележащего графа G: для заданной конфигурации роботов (то есть текущего местонахождения каждого робота) неясно, какое правило следует использовать для принятия решения о перемещении одного из роботов. Вместо этого

применяется идея, которая часто бывает очень полезной в ситуациях с подобным поиском. Можно заметить, что задача очень похожа на задачу поиска пути — но не в исходном графе G, а в пространстве всех возможных конфигураций.

Определим следующий граф H (большего размера). Множество узлов H соответствует множеству всех возможных конфигураций роботов; иначе говоря, H состоит из всех возможных пар узлов в G. Два узла H соединяются ребром, если они представляют конфигурации, которые могут занимать последовательные позиции в процессе планирования; иначе говоря, (u, v) и (u', v') соединяются ребром в H, если одна из пар u, u' или v, v' состоит из одинаковых значений, а другая пара соответствует ребру в G.

Мы уже видим, что пути из (a, b) в (c, d) в H соответствуют правилам планирования для роботов: такой путь состоит из последовательности конфигурации, в которой на каждом шаге один робот пересекает ровно одно ребро в G. Тем не менее в расширенной формулировке еще не отражено правило, согласно которому планирование должно быть свободно от помех.

Для этого из H просто удаляются все узлы, соответствующие конфигурациям с помехами. Таким образом определяется H' — граф, полученный удалением всех узлов (u, v), для которых расстояние между u и v в G не превышает r.

В этом случае полный алгоритм выглядит так. Мы строим граф H', а затем выполняем алгоритм из текста для проверки существования пути из (a, b) в (c, d). Правильность работы алгоритма следует из того факта, что пути в H' соответствуют процессам планирования, а узлы H' точно соответствуют конфигурациям без помех.

Остается оценить время выполнения. Пусть n — количество узлов в G, а m — количество ребер в G. Чтобы проанализировать время выполнения, мы сформулируем оценки для: (1) размера графа H' (который в общем случае больше G), (2) времени построения H' и (3) времени поиска пути от (a, b) к (c, d) в H.

1. Начнем с размера H'. Граф H' содержит не более n^2 узлов, так как его узлы соответствуют парам узлов в G. Сколько же ребер содержит H' ? С узлом (u, v) связаны ребра (u', v) для каждого соседа u' узла u в G и ребра (u, v') для каждого соседа v' узла v в G. Чтобы сформулировать простую верхнюю границу, заметим, что имеется не более n вариантов для (u', v) и не более n вариантов для (u, v'), поэтому каждый узел H' имеет максимум $2n$ инцидентных ребер. Суммируя по (максимум) n^2 узлам H', получаем $O(n^3)$ ребер. (Вообще говоря, для количества ребер H' можно дать улучшенную границу $O(mn)$ на основании границы (3.9), доказанной в разделе 3.3, для суммы степеней графа. Мы вернемся к этой теме позднее.)
2. Перейдем ко времени, необходимому для построения H'. Чтобы построить H, мы переберем все пары узлов в G за время $O(n^2)$ и построим ребра по приведенному выше определению за время $O(n)$ на каждый узел, за общее время $O(n^3)$. Теперь нужно определить, какие узлы следует удалить из H для получения H'. Это можно сделать так: для каждого узла u в G запускается поиск в ширину от u, который находит все узлы v, находящиеся на расстоянии не более r от u. Все такие пары (u, v) удаляются из H. Каждый поиск в ширину в G выполняется за время $O(m + n)$, и мы проводим по одному такому поиску от каждого узла, поэтому общее время для этой части составит $O(mn + n^2)$.

3. Итак, граф H' построен, и теперь достаточно определить, существует ли путь из (a, b) в (c, d). Для этого можно воспользоваться алгоритмом связности (см. выше) за время, линейное по отношению к количеству ребер и узлов H'. Так как H' состоит из $O(n^2)$ узлов и $O(n^3)$ ребер, последний шаг также выполняется за полиномиальное время.

Упражнения

1. Рассмотрим направленный ациклический граф G на рис. 3.10. Сколько топологических упорядочений он имеет?

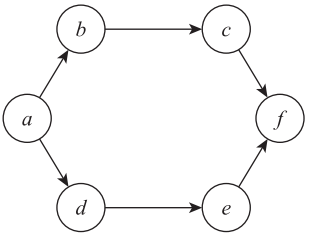


Рис. 3.10. Сколько топологических упорядочений существует у этого графа?

2. Предложите алгоритм для проверки наличия циклов в заданном ненаправленном графе. Если граф содержит цикл, то ваш алгоритм должен вывести его. (Выводить все циклы в графе не нужно, достаточно одного.) Алгоритм должен выполняться за время $O(m + n)$ для графа с n узлами и m ребрами.
3. Алгоритм, описанный в разделе 3.6 для вычисления топологического упорядочения DAG, многократно находит узел, не имеющий входящих ребер, и удаляет его. В конечном итоге это приводит к получению топологического упорядочения (при условии, что входной граф действительно является DAG).

   Но допустим, что вы получили произвольный граф, который может быть или не быть DAG. Расширьте алгоритм топологического упорядочения так, чтобы для входного направленного графа G он выводил одно из двух: (a) топологическое упорядочение, доказывающее, что входной граф является DAG; или (b) цикл в G, доказывающий, что входной граф не является DAG. Алгоритм должен выполняться за время $O(m + n)$ для направленного графа с n узлами и m ребрами.
4. Вдохновившись примером великого Владимира Набокова, ваши друзья занялись лепидоптерологией (проще говоря, изучением бабочек). Часто при возвращении из поездки с новыми образцами им бывает очень трудно определить, сколько же разных видов было поймано, ведь многие виды очень похожи друг на друга.

   Однажды они возвращаются с n бабочками, каждая из которых принадлежит одному из двух видов, — назовем их A и B. Вашим друзьям хотелось бы раз-

делить n образцов на две группы (относящиеся к видам A и B), но напрямую классифицировать каждый отдельно взятый образец очень трудно. Вместо этого они решают действовать немного иначе.

Каждая пара образцов i и j располагается рядом и тщательно изучается. Если исследователи достаточно уверены в своей оценке, они помечают образцы из пары (i, j) либо как «одинаковые» (то есть как относящиеся к одному виду), либо как «разные» (то есть как относящиеся к разным видам). Также они могут отказаться от назначения оценки конкретной паре; в таком случае пара называется неопределенной.

Итак, имеется набор из n образцов, а также набор из m оценок («одинаковые» или «разные») для пар, которые не были отнесены к неопределенным. Вашим друзьям хотелось бы знать, соответствуют ли эти данные представлению о том, что каждая бабочка относится к одному из видов A или B. Говоря конкретнее, набор из m оценок объявляется непротиворечивым, если возможно пометить каждый образец A или B так, чтобы для каждой пары (i, j) с пометкой «одинаковые» i и j относились к одному виду, а для каждой пары (i, j) с пометкой «разные» i и j относились к разным видам. Ваши друзья возятся с проверкой непротиворечивости оценок, как вдруг один из них осознает: ведь вы сможете придумать алгоритм, который позволит с ходу получить ответ на этот вопрос.

Предложите алгоритм для проверки непротиворечивости оценок с временем выполнения $O(m + n)$.

5. *Бинарным деревом* называется корневое дерево, в котором каждый узел имеет не более двух дочерних узлов. Покажите посредством индукции, что в любом бинарном дереве количество узлов с двумя дочерними узлами ровно на 1 меньше количества листьев.

6. Имеется связный граф $G = (V, E)$ и конкретная вершина $u \in V$. Предположим, при вычислении дерева поиска в глубину с корнем u было получено дерево T, включающее все узлы G. После этого при вычислении дерева поиска в ширину с корнем u было получено то же дерево T. Докажите, что $G = T$. (Иначе говоря, если T одновременно является деревом поиска в глубину и в ширину с корнем в u, то G не может содержать ребер, не принадлежащих T.)

7. Ваши друзья, занимающиеся технологиями беспроводных сетей, в настоящее время изучают свойства сети из n мобильных устройств. Перемещаемые устройства (или вернее, их хозяева) в любой момент времени определяют граф по следующим правилам: существует узел, представляющий каждое из n устройств, и между i и j существует ребро, если физические позиции i и j находятся на расстоянии не более 500 метров. (В таком случае устройства находятся «в пределах дальности» друг друга.)

   Чтобы сеть устройств по возможности оставалась связной, перемещение устройств ограничивается по следующему правилу: в любое время каждое устройство i должно находиться на расстоянии не более 500 метров от по крайней мере $n/2$ других устройств (где n — четное число). Требуется узнать: гарантирует ли это свойство само по себе, что сеть будет оставаться связной?

Ниже приведена конкретная формулировка этого вопроса как утверждения применительно к графам.

Утверждение: пусть G — граф из n узлов, где n — четное число. Если каждый узел G имеет степень не менее n/2, то граф G является связным.

Считаете вы это утверждение истинным или ложным? Приведите доказательство истинности или ложности.

8. Во многих публикациях, посвященных структуре Интернета и Всемирной паутины, в том или ином виде обсуждался следующий вопрос: на каком расстоянии находятся типичные узлы этих сетей? Если внимательно читать такие статьи, выясняется, что многие из них путают понятия *диаметра сети* и *среднего расстояния* в сети; авторы используют эти понятия так, словно они являются взаимозаменяемыми.

   В соответствии с нашей терминологией *расстоянием* между двумя узлами u и v в графе $G = (V, E)$ называется минимальное количество ребер в пути, связывающем эти узлы; расстояние будет обозначаться $dist(u, v)$. *Диаметром G* называется максимальное расстояние между любой парой *узлов* G; эта характеристика будет обозначаться $diam(G)$.

   Определим сопутствующую характеристику, которая будет называться *средним попарным расстоянием* в G, обозначим ее $apd(G)$. Величина $apd(G)$ вычисляется как среднее арифметическое по всем $\binom{n}{2}$ множествам из двух разных узлов u и v для расстояния между u и v.

   Другими словами,

$$apd(G) = \left[\sum_{\{u,v\}\subseteq V} dist(u,v)\right] / \binom{n}{2}.$$

   Простой пример наглядно показывает, что существуют графы G, для которых $diam(G) \neq apd(G)$. Пусть G — граф с тремя узлами u, v, w и двумя ребрами $\{u, v\}$ и $\{v, w\}$. Тогда

$$diam(G) = dist(u, w) = 2,$$

   при том, что

$$apd(G) = [dist(u, v) + dist(u, w) + dist(v, w)] / 3 = 4/3.$$

   Конечно, эти два числа *не так уж* сильно отличаются в случае графа из трех узлов; будет естественно задаться вопросом, всегда ли между ними существуют тесные отношения. Следующее утверждение пытается формализовать этот вопрос.

   Утверждение: *существует положительное натуральное число c, при котором для любого связного графа G выполняется условие*

$$\frac{diam(G)}{apd(G)} \leq c.$$

Считаете вы это утверждение истинным или ложным? Приведите доказательство истинности или ложности.

9. Интуитивно понятно, что качество связи между двумя узлами коммуникационной сети, находящимися на большом расстоянии (с множеством переходов), хуже, чем у двух узлов, расположенных вблизи друг от друга. Существует ряд алгоритмических результатов, в определенной степени зависящих от разных способов формального отражения этих представлений. В следующем варианте учитывается устойчивость путей к удалению узлов.

   Допустим, ненаправленный граф $G = (V, E)$ из n узлов содержат два узла s и t, расстояние между которыми строго больше $n/2$. Покажите, что должен существовать некоторый узел v, отличный от s и t, такой, что удаление v из G приведет к уничтожению всех путей s–t. (Другими словами, граф, полученный из G посредством удаления v, не содержит путей от s к t.) Предложите алгоритм поиска такого узла v с временем выполнения $O(m + n)$.

10. Во многих музеях можно увидеть работы художника Марка Ломбарди (1951–2000), которые представляют собой художественно воспроизведенные графы. На этих графах изображены полученные в результате серьезных исследований отношения между людьми, замешанными в крупных политических скандалах за последние десятилетия: узлы представляют участников, а ребра — некие отношения в парах. Если внимательно присмотреться к этим рисункам, можно проследить довольно зловещие пути от высокопоставленных государственных служащих США через бывших деловых партнеров и швейцарские банки к подпольным торговцам оружием.

    Такие изображения служат ярким примером социальных сетей, в которых, как упоминалось в разделе 3.1, узлы представляют людей и организации, а ребра — различного рода отношения. Короткие пути, в изобилии встречающиеся в таких сетях, в последнее время привлекают общественное внимание: люди размышляют над их смыслом. В случае с графами Марка Ломбарди такие пути наводят на мысли о короткой последовательности шагов, приводящих от хорошей репутации к позору.

    Конечно, один короткий путь между узлами v и w в такой сети может объясняться простой случайностью; большое количество коротких путей между v и w выглядит куда убедительнее. По этой причине наряду с задачей вычисления одного кратчайшего пути v-w в графе G исследователи социальных сетей занимались рассмотрением задачи определения количества кратчайших путей v-w.

    Как выяснилось, у этой задачи есть эффективное решение. Допустим, имеется ненаправленный граф $G = (V, E)$, в котором выделены два узла v и w. Приведите алгоритм, который вычисляет количество кратчайших путей v-w. (Алгоритм не должен выводить все пути; достаточно указать их количества.) Алгоритм должен выполняться за время $O(m + n)$ для графа с n узлами и m ребрами.

11. Вы помогаете специалистам по безопасности отслеживать распространение компьютерного вируса. Система состоит из n компьютеров $C_1, C_2, ..., C_n$; на входе вы получаете набор данных трассировки с указанием времени взаимодействий

между парами компьютеров. Таким образом, данные представляют собой последовательность упорядоченных триплетов вида (C_i, C_j, t_k), сообщающих, что между компьютерами C_i и C_j передавались данные во время t_k. Общее количество триплетов равно m.

Предполагается, что триплеты передаются отсортированными по времени. Для простоты будем считать, что каждая пара компьютеров обменивается данными не более одного раза за интервал наблюдения. Специалисты по безопасности хотели бы иметь возможность получать ответы на вопросы следующего типа: если вирус был внедрен на компьютер C_a во время x, мог ли он достичь компьютера C_b ко времени y? Механика распространения вируса проста: если зараженный компьютер C_i взаимодействует с незараженным компьютером C_j во время t_k (другими словами, если в данных трассировки присутствует один из триплетов (C_i, C_j, t_k) или (C_j, C_i, t_k)), то компьютер C_j тоже становится зараженным начиная с времени t_k.

Таким образом, вирус передается с машины на машину через серию взаимодействий; предполагается, что ни один шаг в этой последовательности не подразумевает перемещения назад во времени. Например, если компьютер C_i заражен к моменту t_k, и данные трассировки содержат триплеты (C_i, C_j, t_k) и (C_j, C_q, t_r), где $t_k \leq t_r$, то компьютер C_q будет заражен через C_j. (Обратите внимание: время t_k может быть равно t_r; это означало бы, что у C_j были одновременно открыты подключения к C_i и C_q, так что вирус мог переместиться с C_i на C_q.)

Допустим, для $n = 4$ данные трассировки состоят из триплетов

$$(C_1, C_2, 4), (C_2, C_4, 8), (C_3, C_4, 8), (C_1, C_4, 12)$$

и вирус был внедрен на компьютер C_1 во время 2. В этом случае компьютер C_3 будет заражен ко времени 8 серией из трех шагов: сначала C_2 заражается во время 4, затем C_4 получает вирус от C_2 ко времени 8 и, наконец, C_3 получает вирус от C_4 ко времени 8. С другой стороны, если бы данные трассировки имели вид

$$(C_2, C_3, 8), (C_1, C_4, 12), (C_1, C_2, 14)$$

и вирус снова был бы внедрен на компьютер C_1 во время 2, то компьютер C_3 не был бы заражен за наблюдаемый период: хотя C_2 заражается к времени 14, мы видим, что C_3 взаимодействует с C_2 до заражения C_2. Никакая последовательность взаимодействий с прямым перемещением во времени не приведет к распространению вируса с C_1 на C_3 во втором примере.

Разработайте алгоритм, который позволит получить ответы на вопрос типа: для заданного набора данных трассировки необходимо определить, сможет ли вирус, внедренный на компьютере C_a во время x, заразить компьютер C_b ко времени y. Алгоритм должен выполняться за время $O(m + n)$.

12. Вы помогаете группе ученых-этнографов в анализе устных исторических данных о людях, живших в некоторой деревне за последние двести лет. В этих материалах ученые узнали о множестве из n людей (все эти люди уже умерли), которых мы будем обозначать $P_1, P_2, ..., P_n$. Также были собраны сведения о вре-

мени жизни этих людей друг относительно друга. Каждый факт представлен в одной из двух форм:

- для некоторых i и j человек P_i умер до рождения человека P_j; или
- для некоторых i и j сроки жизни P_i и P_j перекрывались по крайней мере частично.

Естественно, ученые не уверены в правильности *всех* этих фактов; память порой подводит, а многие сведения передавались из уст в уста. Они хотят, чтобы вы проверили хотя бы внутреннюю непротиворечивость собранных данных, то есть может ли существовать множество людей, для которых одновременно истинны все собранные факты.

Предложите эффективный алгоритм для решения этой задачи: алгоритм должен либо выдавать предполагаемые даты рождения и смерти каждого из n людей, чтобы все факты были истинны, либо сообщать (правильно), что такие даты существовать не могут, то есть что набор фактов, собранных этнографами, внутренне противоречив.

Примечания и дополнительная литература

Теория графов — большая тема, включающая как алгоритмические, так и неалгоритмические темы. Обычно считается, что теория графов началась с работы Эйлера (Euler, 1736), развивалась на основе интереса к представлению карт и химических компонентов в виде графов в XIX веке и стала областью систематических научных исследований в XX веке — сначала как ветвь математики, а позднее на основе практического применения в информатике. В книгах Берга (Berge, 1976), Боллобаса (Bollobas, 1998) и Дистела (Diestel, 2000) основательно рассматриваются многие вопросы теории графов. В последнее время появились обширные данные для изучения больших сетей, встречающихся в физике, биологии и социологии, а также возрос интерес к свойствам сетей, задействованных во всех этих областях. В книгах Барабаси (Barabasi, 2002) и Уоттса (Watts, 2002) эта перспективная область исследований обсуждается с примерами, ориентированным на общую аудиторию.

Базовые методы обхода графов, представленные в этой главе, находят множество практических применений. Примеры рассматриваются в следующих главах, также за дополнительной информацией рекомендуем обращаться к книге Тарьяна (Tarjan, 1983).

Примечания к упражнениям

Упражнение 12 основано на результатах Мартина Голумбика и Рона Шамира.

Глава 4
Жадные алгоритмы

В культовом фильме 1980-х годов «Уолл-стрит» Майкл Дуглас встает перед залом, заполненным акционерами, и провозглашает: «Жадность... это хорошо. Жадность — это правильно. Жадность работает». В этой главе мы будем руководствоваться более умеренным подходом к изучению достоинств и недостатков жадности при проектировании алгоритмов. Мы постараемся рассмотреть нескольких вычислительных задач через призму одних и тех же вопросов: жадность — это действительно хорошо? Жадность действительно работает?

Дать точное определение жадного алгоритма достаточно трудно, если вообще возможно. Алгоритм называется *жадным*, если он строит решение за несколько мелких шагов, а решение на каждом шаге принимается без опережающего планирования, с расчетом на оптимизацию некоторого внутреннего критерия. Часто для одной задачи удается разработать несколько жадных алгоритмов, каждый из которых локально и постепенно оптимизирует некоторую метрику на пути к решению.

Когда жадный алгоритм успешно находит оптимальное решение нетривиальной задачи, этот факт обычно дает интересную и полезную информацию о структуре самой задачи; существует локальное правило принятия решений, которое может использоваться для построения оптимальных решений. И как будет показано позже, в главе 11, это относится к задачам, в которых жадный алгоритм может привести к решению, гарантированно близкому к оптимальному, даже если он не достигает точного оптимума. Таковы основные проблемы, с которыми мы будем иметь дело в этой главе. Легко придумать жадный алгоритм почти для любой задачи; найти ситуации, в которых они хорошо работают, и доказать, что они работают действительно хорошо, будет сложнее.

В первых двух разделах этой главы представлены два основных метода доказательства того, что жадный алгоритм предоставляет оптимальное решение задачи. Первый метод можно обозначить формулой «*жадный алгоритм опережает*»: имеется в виду, что при пошаговой оценке прогресса жадного алгоритма становится видно, что на каждом шаге он работает лучше любого другого алгоритма; из этого следует, что алгоритм строит оптимальное решение. Второй метод, называемый *заменой*, имеет более общий характер: сначала рассматривается любое возможное решение задачи, которое последовательно преобразуется в решение, находимое жадным алгоритмом, без ущерба для его качества. И снова из этого следует, что решение, найденное жадным алгоритмом, по крайней мере не хуже любого другого решения.

После знакомства с этими двумя видами анализа мы сосредоточимся на нескольких хорошо известных практических применениях жадных алгоритмов: *поиске кратчайших путей* в графе, задаче нахождения *минимального остовного дерева* и построении *кодов Хаффмана* для сжатия данных. Также будут исследованы интересные отношения между минимальными остовными деревьями и давно изучавшейся задачей кластеризации. Напоследок будет рассмотрен более сложный пример — задача *ориентированного дерева минимальной стоимости*, которая расширит наши представления о жадных алгоритмах.

4.1. Интервальное планирование: жадный алгоритм опережает

Вспомните задачу интервального планирования — первую из пяти типичных задач, рассмотренных в главе 1. Имеется множество заявок $\{1, 2, ..., n\}$; i-я заявка соответствует интервалу времени, который начинается в $s(i)$ и завершается в $f(i)$. (Обратите внимание: мы слегка изменили запись по сравнению с разделом 1.2, где вместо $s(i)$ использовалось обозначение s_i, а вместо $f(i)$ — обозначение f_i; такое изменение упростит запись доказательств.) Подмножество заявок называется *совместимым*, если никакие две заявки не перекрываются по времени; наша цель — получить совместимое подмножество как можно большего размера. Совместимые множества максимального размера называются *оптимальными*.

Проектирование жадного алгоритма

Задача интервального планирования сделает наше обсуждение жадных алгоритмов более конкретным. Основной идеей жадного алгоритма для интервального планирования является простое правило выбора первой заявки i_1. После того как заявка i_1 будет принята, все заявки, несовместимые с i_1, отклоняются. Затем выбирается следующая заявка i_2, а все заявки, несовместимые с i_2, отклоняются. Процедура продолжается до тех пор, пока не будут исчерпаны все заявки. Основной сложностью проектирования хорошего жадного алгоритма является выбор простого правила, причем у этой задачи существует много естественных правил, которые не обеспечивают хороших решений.

Возьмем несколько естественных правил и посмотрим, как они работают.

- Самое очевидное правило — всегда выбирать доступную заявку с самым ранним начальным временем, то есть заявку с минимальным значением $s(i)$. При таком выборе ресурс начинает использоваться настолько быстро, насколько это возможно.

  Этот метод не обеспечивает оптимального решения. Если самая ранняя заявка i резервирует ресурс на очень долгое время, то принятие заявки i приведет к отклонению множества заявок на более короткие интервалы. Так как нашей целью

является удовлетворение максимально возможного количества заявок, решение получится субоптимальным. В очень плохом случае — скажем, если конечное время $f(i)$ максимально по всем заявкам — принятая заявка i зарезервирует ресурс на все время. В этом случае жадный алгоритм примет одну заявку, тогда как оптимальное решение могло бы принять много заявок. Ситуация изображена на рис. 4.1, *a*.

- Предыдущий результат наводит на мысль о том, чтобы начать с заявки с наименьшим интервалом времени, а именно той, для которой разность $f(i) - s(i)$ оказывается наименьшей из всех возможных. Как выясняется, это правило чуть лучше предыдущего, но и оно может приводить к субоптимальному расписанию. Например, на рис. 4.1, *b* выбор короткого интервала в середине помешает принятию двух других заявок, формирующих оптимальное решение.

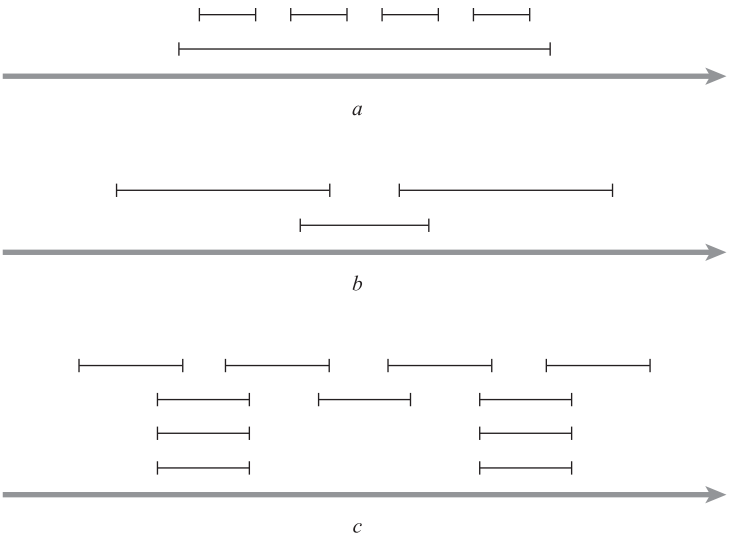


Рис. 4.1. Примеры ситуаций интервального планирования, в которых естественные жадные алгоритмы не приводят к оптимальному решению. В случае *a* не работает выбор интервала с самым ранним начальным временем; в случае *b* не работает выбор самого короткого интервала; в случае *c* не работает выбор интервала с минимальным количеством конфликтов

- В предыдущем примере проблема заключалась в том, что вторая заявка конкурировала с первой и третьей, то есть принятие этой заявки приводило к отклонению двух других заявок. Также можно спроектировать жадный алгоритм, основанный на следующей идее: для каждой заявки подсчитывается количество других несовместимых заявок и принимается заявка с минимальным количеством несовместимых результатов. (Другими словами, выбирается интервал с наименьшим количеством «конфликтов».) В рассмотренном примере жадный выбор ведет к оптимальному решению. Более того, для этого правила труднее найти контрпример; и все же это возможно, как показывает рис. 4.1, *c*. Оптимальное решение в этом примере — принятие четырех заявок из верхней

строки. Жадный алгоритм с минимизацией конфликтов принимаем среднюю заявку из второй строки, а следовательно, обеспечивает решение с размером не более 3.

Жадное правило, которое приводит к оптимальному решению, строится на четвертой идее: первой принимается заявка, которая завершается первой (то есть заявка i с наименьшим значением $f(i)$). Идея вполне естественная: она гарантирует как можно более раннее освобождение ресурса и при этом удовлетворяет одну заявку. Тем самым обеспечивается максимизация времени, оставшегося для удовлетворения других заявок.

Давайте определим алгоритм более формально. Пусть R — множество заявок, не принятых и не отклоненных на данный момент, а A — множество принятых заявок. Пример выполнения алгоритма изображен на рис. 4.2.

```
Инициализировать R множеством всех заявок, A – пустым множеством
Пока множество R не пусто
   Выбрать заявку i ∈ R с наименьшим конечным временем
   Добавить заявку i в A
   Удалить из R все заявки, несовместимые с запросом i
Конец Пока
Вернуть A как множество принятых заявок
```

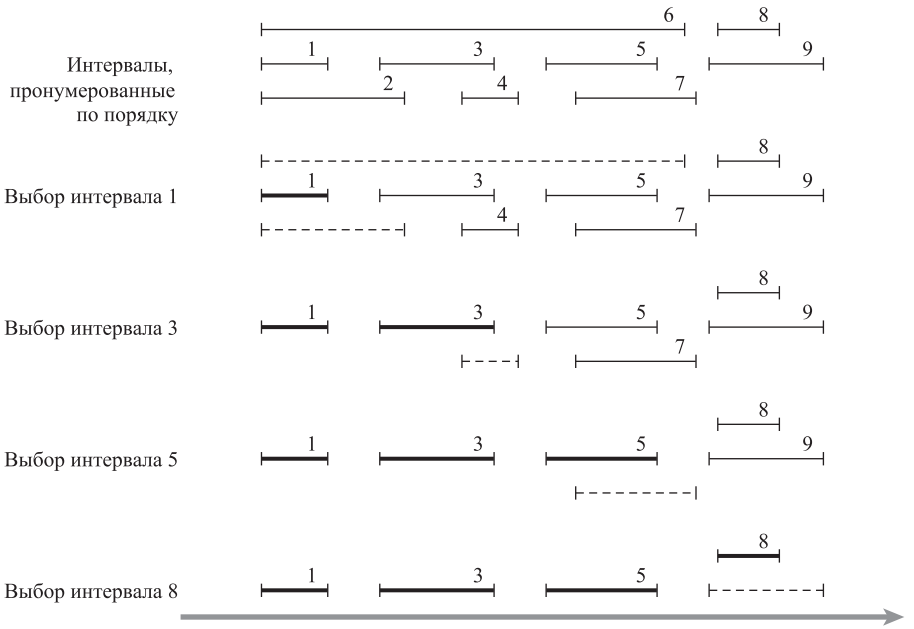


Рис. 4.2. Пример выполнения алгоритма интервального планирования. На каждом шаге выбранные интервалы обозначены темными отрезками, а удаляемые интервалы — пунктирными линиями

Анализ алгоритма

Хотя этот жадный метод базируется на вполне естественном принципе, оптимальность возвращаемого множества интервалов не столь очевидна. В самом деле, торопиться с вердиктом вообще не стоит: идеи, приведшие к описанным выше неоптимальным версиям жадного алгоритма, также на первый взгляд выглядели перспективно.

Для начала можно сразу утверждать, что интервалы множества A, возвращаемого алгоритмом, совместимы.

(4.1) Множество A состоит из совместимых заявок.

Нужно продемонстрировать, что это решение оптимально. Итак, пусть O — оптимальное множество интервалов. В идеале хотелось бы показать, что $A = O$, но это уже слишком: оптимальных решений может быть несколько, и в лучшем случае A совпадает с одним из них. Итак, вместо этого мы просто покажем, что $|A| = |O|$, то есть A содержит столько же интервалов, сколько и O, а следовательно, является оптимальным решением.

Как упоминалось в начале главы, идея, заложенная в основу этого доказательства, заключается в том, чтобы найти критерий, по которому наш жадный алгоритм опережает решение O. Мы будем сравнивать частичные решения, создаваемые жадным алгоритмом, с начальными сегментами решения O и шаг за шагом покажем, что жадный алгоритм работает не хуже.

Для упрощения доказательства будут введены дополнительные обозначения. Пусть $i_1, ..., i_k$ — множество заявок A в порядке их добавления в A. Заметьте, что $|A| = k$. Аналогичным образом множество заявок O будет обозначаться $j_1, ..., j_m$. Мы намерены доказать, что $k = m$. Допустим, заявки в O также упорядочены слева направо по соответствующим интервалам, то есть по начальным и конечным точкам. Не забывайте, что заявки в O совместимы, то есть начальные точки следуют в том же порядке, что и конечные.

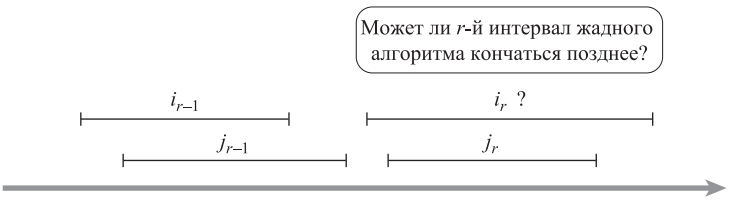


Рис. 4.3. Шаг индукции в доказательстве того, что жадный алгоритм идет впереди

Выбор жадного метода происходит от желания освобождать ресурс как можно раньше после удовлетворения первой заявки. В самом деле, жадное правило гарантирует, что $f(i_1) \le f(j_1)$. Именно в этом смысле мы стараемся показать, что жадное правило «идет впереди» означает, что каждый из его интервалов заканчивается по крайней мере не позднее соответствующего интервала множества O. Соответственно сейчас мы докажем, что для каждого $r \ge 1$ r-я принятая заявка в расписании алгоритма завершается не позднее r-й заявки оптимального расписания.

(4.2) Для всех индексов $r \le k$ выполняется условие $f(i_r) \le f(j_r)$.

Доказательство. Утверждение будет доказано методом индукции. Для $r = 1$ оно очевидным образом истинно: алгоритм начинается с выбора заявки i_1 с наименьшим временем завершения.

Теперь рассмотрим $r > 1$. Согласно гипотезе индукции, будем считать, что утверждение истинно для $r - 1$, и попробуем доказать его для r. Как видно из рис. 4.3, индукционная гипотеза позволяет считать, что $f(i_{r-1}) \le f(j_{r-1})$. Если r-й интервал алгоритма не завершается раньше, он должен «отставать». Но существует простая причина, по которой это невозможно: вместо того, чтобы выбирать интервал с более поздним завершением, жадный алгоритм всегда имеет возможность (в худшем случае) выбрать j_r и таким образом реализовать шаг индукции.

В формальном представлении этот аргумент выглядит так. Мы знаем (так как O состоит из совместимых интервалов), что $f(j_{r-1}) \le s(j_r)$. Объединяя это с индукционной гипотезой $f(i_{r-1}) \le f(j_{r-1})$, получаем $f(i_{r-1}) \le s(j_r)$. Следовательно, интервал j_r принадлежит множеству R доступных интервалов на тот момент, когда жадный алгоритм выбирает i_r. Жадный алгоритм выбирает доступный интервал с *наименьшим* конечным временем; так как j_r является одним из таких доступных интервалов, имеем $f(i_r) \le f(j_r)$. На этом шаг индукции завершается. ■

Таким образом, мы формализовали смысл, в котором жадный алгоритм «идет впереди» O: для всех r выбираемый r-й интервал завершается по крайней мере не позже r-го интервала в O. А сейчас мы увидим, почему из этого следует оптимальность множества A жадного алгоритма.

(4.3) Жадный алгоритм возвращает оптимальное множество A.

Доказательство. Утверждение будет доказано от обратного. Если множество A не оптимально, то оптимальное множество O должно содержать больше заявок, то есть $m > k$. Применяя (4.2) для $r = k$, получаем, что $f(i_k) \le f(j_k)$. Так как $m > k$, в O должна существовать заявка j_{k+1}. Она начинается после завершения заявки j_k, а следовательно, после завершения i_k. Получается, что после удаления всех заявок, несовместимых с заявками $i_1, ..., i_k$, множество возможных заявок R по-прежнему будет содержать j_{k+1}. Но тогда жадный алгоритм останавливается на заявке i_k, а должен останавливаться только на пустом множестве R, — противоречие. ■

Реализация и время выполнения

Алгоритм может выполняться за время $O(n \log n)$. Сначала n заявок следует отсортировать в порядке конечного времени и пометить их в этом порядке; соответственно предполагается, что $f(i) \le f(j)$, когда $i < j$. Этот шаг выполняется за время $O(n \log n)$. За дополнительное время $O(n)$ строится массив $S[1... n]$, в котором элемент $S[i]$ содержит значение $s(i)$.

Теперь выбор заявок осуществляется обработкой интервалов в порядке возрастания $f(i)$. Сначала всегда выбирается первый интервал; затем интервалы перебираются по порядку до тех пор, пока не будет достигнут первый интервал j, для которого $s(j) \ge f(1)$; он также включается в результат. В более общей форму-

лировке, если последний выбранный интервал заканчивается во время f, перебор последующих интервалов продолжается до достижения первого интервала j, для которого $s(j) \geq f$. Таким образом, описанный выше жадный алгоритм реализуется за один проход по интервалам с постоянными затратами времени на интервал. Следовательно, эта часть алгоритма выполняется за время $O(n)$.

Расширения

Задача интервального планирования, рассмотренная нами, относится к числу относительно простых задач планирования. На практике возникает много дополнительных сложностей. Ниже перечислены проблемы, которые в той или иной форме встретятся позднее в книге.

- В определении задачи предполагалось, что все заявки были известны алгоритму планирования при выборе совместимого подмножества. Конечно, также было бы естественно рассмотреть версию задачи, в которой планировщик должен принимать решения по принятию или отклонению заявок до того, как ему будет известен полный набор заявок. Если планировщик будет слишком долго собирать информацию обо всех заявках, клиенты (отправители заявок) могут потерять терпение, отказаться и уйти. Сейчас в области таких оперативных алгоритмов, которые должны принимать решения «на ходу», без полной информации о будущих заявках, ведутся активные исследования.
- Наша постановка задачи стремилась к максимизации удовлетворенных заявок. Однако также можно представить ситуацию, в которой заявки обладают разной ценностью. Например, каждой заявке i может быть присвоен вес v_i (доход от удовлетворения заявки i), и целью становится обеспечение максимального дохода: суммы весов всех удовлетворенных заявок. Так мы подходим к задаче взвешенного интервального планирования — второй из типичных задач, упоминавшихся в главе 1.

У задачи интервального планирования существует много других вариаций и разновидностей. Сейчас мы обсудим одну из таких вариаций, потому что она дает еще один пример ситуации, в которой жадный алгоритм может использоваться для получения оптимального решения.

Взаимосвязанная задача: планирование всех интервалов

Задача

В задаче интервального планирования используется один ресурс и много заявок в форме временных интервалов; требуется выбрать, какие заявки следует принять, а какие отклонить. Во взаимосвязанной задаче используется несколько идентичных ресурсов, для которых необходимо распланировать все заявки с использованием минимально возможного количества ресурсов. Так как в этом случае

все интервалы должны быть распределены по нескольким ресурсам, мы будем называть ее задачей интервального разбиения[1].

Представьте, например, что каждая заявка соответствует лекции, которая планируется для проведения в аудитории на заданный период времени. Требуется удовлетворить все заявки с использованием минимального количества аудиторий. Таким образом, находящиеся в вашем распоряжении аудитории являются множественными ресурсами, а основное ограничение заключается в том, что две лекции, перекрывающиеся по времени, должны проводиться в разных аудиториях. Или, например, интервальные заявки могут представлять задания, которые должны быть обработаны за некоторый период времени, а ресурсы представляют машины, способные эти задания обрабатывать. Далее в книге, в главе 10, будет представлено еще одно применение этой задачи, в котором интервалы представляют запросы, которые должны распределяться в соответствии с пропускной способностью оптоволоконного кабеля.

Для наглядности рассмотрим пример на рис. 4.4, *a*. Все заявки из этого примера могут быть распределены по трем ресурсам; такое распределение изображено на рис. 4.4, *b*: заявки размещаются в три строки, каждая из которых содержит набор неперекрывающихся интервалов. В общем случае решение с k ресурсами можно представить как задачу распределения заявок в k строк с неперекрывающимися интервалами; первая строка содержит все интервалы, назначенные первому ресурсу, вторая строка — интервалы, назначенные второму ресурсу, и т. д.

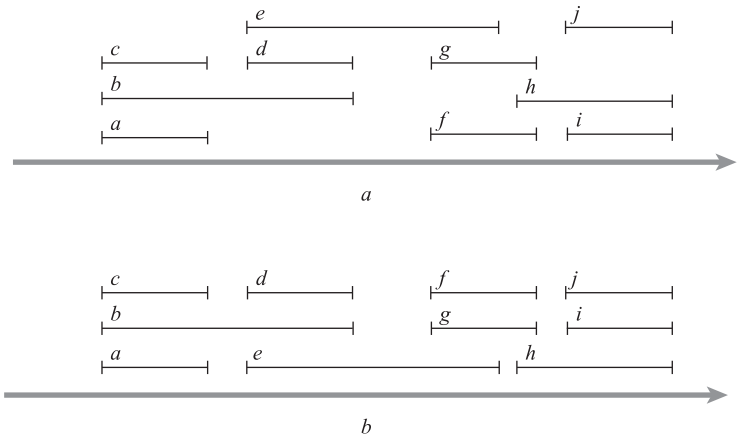


Рис. 4.4. *a* — пример задачи интервального планирования с десятью интервалами (от *a* до *j*); *b* — в приведенном решении интервалы распределяются по трем ресурсам: каждая строка представляет набор интервалов, которые назначаются одному ресурсу

Возможно ли обойтись только двумя ресурсами в этом конкретном примере? Разумеется, нет. Необходимо как минимум три ресурса, потому что, например,

[1] Также встречается название «задача интервального окрашивания»; термин происходит от представления о ресурсах как об окрашенных в разные цвета — всем интервалам, назначаемым конкретному ресурсу, присваивается соответствующий цвет.

интервалы *a*, *b* и *c* занимают общую точку временной шкалы, а следовательно, должны быть распределены по разным ресурсам. Последнее обстоятельство применимо к любой ситуации интервального разбиения. Допустим, *глубина* множества интервалов определяется как максимальное число интервалов, проходящих через одну точку временной шкалы. Тогда справедливо следующее утверждение:

(4.4) В любой ситуации интервального разбиения количество необходимых ресурсов не меньше глубины множества интервалов.

Доказательство. Допустим, множество интервалов имеет глубину *d*, а интервалы $I_1, ..., I_d$ проходят через одну общую точку временной шкалы. Все эти интервалы должны быть распределены по разным ресурсам, поэтому решению в целом необходимы как минимум *d* ресурсов.

А теперь рассмотрим два вопроса, которые, как выясняется, тесно связаны друг с другом. Во-первых, можно ли спроектировать эффективный алгоритм, который планирует все интервалы с минимально возможным количеством ресурсов? Во-вторых, всегда ли существует расписание с количеством ресурсов, равным глубине? Положительный ответ на второй вопрос будет означать, что препятствия по разбиению интервалов имеют чисто локальную природу — они сводятся к набору интервалов, занимающих одну точку. На первый взгляд непонятно, могут ли существовать другие, «отложенные» препятствия, которые дополнительно увеличивают количество необходимых ресурсов.

А теперь мы спроектируем простой жадный алгоритм, который распределяет все интервалы по количеству ресурсов, равному глубине. Из этого незамедлительно следует оптимальность алгоритма: в свете (4.4) никакое решение не может использовать количество ресурсов, меньшее глубины. Анализ алгоритма продемонстрирует еще один общий подход к доказательству оптимальности: сначала вы находите простую «структурную» границу, которая доказывает, что у любого возможного решения некоторая метрика должна быть не ниже определенного значения, а потом показываете, что рассматриваемый алгоритм всегда обеспечивает эту границу. ■

Проектирование алгоритма

Пусть *d* — глубина множества интервалов; каждому интервалу будет назначена метка из множества чисел {1, 2, ..., *d*} так, чтобы перекрывающиеся интервалы помечались разными числами. Так мы получим нужное решение, поскольку каждое число может интерпретироваться как имя ресурса, а метка каждого интервала — как имя ресурса, которому он будет назначен.

Используемый для этой цели алгоритм представляет собой однопроходную жадную стратегию упорядочения интервалов по начальному времени. Мы переберем интервалы в указанном порядке и попытаемся присвоить каждому обнаруженному интервалу метку, которая еще не была присвоена никакому из предыдущих интервалов, перекрывающемуся с текущим. Более конкретное описание алгоритма приводится ниже.

```
Отсортировать интервалы по начальному времени, с произвольным порядком совпадений
Пусть I_1, I_2, ..., I_n – обозначения интервалов в указанном порядке
Для j = 1,2,3, ..., n
```

```
   Для каждого интервала I_i, который предшествует I_j в порядке сортировки
и перекрывает его
      Исключить метку i из рассмотрения для I_j
   Конец цикла
   Если существует метка из множества {1,2, ..., d}, которая еще не исключена
      Присвоить неисключенную метку I_j
   Иначе
      Оставить I_j без метки
   Конец Если
Конец цикла
```

Анализ алгоритма

Рассмотрим следующее утверждение:

(4.5) При использовании описанного выше жадного алгоритма каждому интервалу будет назначена метка, и никаким двум перекрывающимся интервалам не будет присвоена одна и та же метка.

Доказательство. Начнем с доказательства того, что ни один интервал не останется не помеченным. Рассмотрим один из интервалов I_j и предположим, что в порядке сортировки существует t интервалов, которые начинаются ранее и перекрывают его. Эти t интервалов в сочетании с I_j образуют множество из i интервалов, которые все проходят через общую точку временной шкалы (а именно начальное время I_j), поэтому $t + 1 \leq d$. Следовательно, $t \leq d - 1$. Из этого следует, что по крайней мере одна из меток d не будет исключена из этого множества интервалов t, поэтому существует метка, которая может быть назначена I_j.

Далее утверждается, что никаким двум перекрывающимся интервалам не будут назначены одинаковые метки. В самом деле, возьмем два перекрывающихся интервала I и I' и предположим, что I предшествует I' в порядке сортировки. Затем, при рассмотрении алгоритмом I', интервал I принадлежит множеству интервалов, метки которых исключаются из рассмотрения; соответственно, алгоритм не назначит I' метку, которая использовалась для I. ■

Алгоритм и его анализ очень просты. По сути, в вашем распоряжении имеется d меток, затем при переборе интервалов слева направо с назначением доступной метки каждому обнаруженному интервалу никогда не возникнет ситуация, в которой все метки уже задействованы.

Так как наш алгоритм использует d меток, из (4.4) можно сделать вывод, что он использует минимально возможное количество меток. Данный результат обобщается следующим образом.

(4.6) Описанный выше жадный алгоритм связывает каждый интервал с ресурсом, используя количество ресурсов, равное глубине множества интервалов. Это количество ресурсов является минимально необходимым, то есть оптимальным.

4.2. Планирование для минимизации задержки: метод замены

А теперь обсудим задачу планирования, связанную с той, с которой началась эта глава. Несмотря на сходства в формулировке задачи и жадном алгоритме, используемом для ее решения, доказательство оптимальности алгоритма потребует более сложного анализа.

Задача

Вернемся к ситуации с одним ресурсом и набором из n заявок на его использование в течение интервала времени. Будем считать, что ресурс доступен, начиная с времени s. Однако в отличие от предыдущей задачи заявки становятся более гибкими — вместо начального и конечного времени заявка содержит *предельное время* d_i и непрерывный интервал времени длиной t_i, а начинаться она может в любое время до своего предельного времени. Каждой принятой заявке должен быть выделен интервал времени длиной t_i, и разным заявкам должны назначаться неперекрывающиеся интервалы.

Существуют разные объективные функции, к оптимизации которых можно было бы стремиться в такой ситуации, и некоторые из них по вычислительной сложности существенно превосходят другие. Здесь мы рассмотрим очень естественную цель, которая может оптимизироваться жадным алгоритмом. Допустим, мы намерены удовлетворить каждую заявку, но некоторые заявки могут быть отложены на более позднее время. Таким образом, начиная с общего начального времени s, каждой заявке i выделяется интервал времени t_i; обозначим этот интервал $[s(i), f(i)]$, где $f(i) = s(i) + t_i$. Однако в отличие от предыдущей задачи этот алгоритм должен определить начальное время (а следовательно, и конечное время) для каждого интервала.

Заявка i будет называться *задержанной*, если она не успевает завершиться к предельному времени, то есть если $f(i) > d_i$. *Задержка* такой заявки i определяется по формуле $l_i = f(i) - d_i$. Будем считать, что если заявка не является просроченной, то $l_i = 0$. Целью новой задачи оптимизации будет планирование всех заявок с неперекрывающимися интервалами, обеспечивающее минимизацию максимальной задержки $L = \max_i l_i$. Эта задача естественным образом встречается при планировании вычислительных заданий, которые должны выполняться на одном компьютере, поэтому мы будем называть заявки «заданиями».

На рис. 4.5 изображен пример с тремя заданиями: первое имеет длину $t_1 = 1$ с предельным временем $d_1 = 2$, у второго $t_2 = 2$ и $d_2 = 4$, и у третьего $t_3 = 3$ и $d_3 = 6$. Нетрудно убедиться в том, что при планировании заданий в порядке 1, 2, 3 максимальная задержка равна 0.

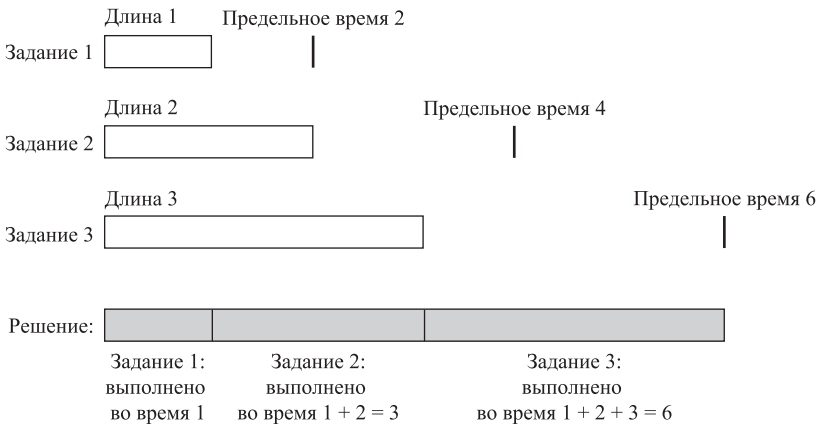


Рис. 4.5. Пример планирования с минимизацией задержки

Проектирование алгоритма

Как же должен выглядеть жадный алгоритм для такой задачи? Есть несколько естественных жадных решений, в которых мы просматриваем данные заданий (t_i, d_i) и используем эту информацию для упорядочения заданий по некоторому простому правилу.

- Одно из решений заключается в планировании заданий по возрастанию длины t_i для того, чтобы поскорее избавиться от коротких заданий. Впрочем, примитивность такого решения сразу же становится очевидной, поскольку оно полностью игнорирует предельное время выполнения заданий. В самом деле, рассмотрим пример с двумя заданиями: у первого $t_1 = 1$ и $d_1 = 100$, а у второго $t_2 = 10$ и $d_2 = 10$. Для достижения задержки $L = 0$ второе задание должно начаться немедленно (и на самом деле запуск второго задания в первую очередь является оптимальным решением).
- Предыдущий пример предполагает, что нам следует ориентироваться на задания с очень малым *запасом времени* $d_i - t_i$, то есть задания, которые должны запускаться с минимальной задержкой. Таким образом, более естественный жадный алгоритм должен сортировать задания в порядке возрастания запаса времени $d_i - t_i$.

К сожалению, это жадное правило тоже не работает. Рассмотрим пример из двух заданий: у первого задания $t_1 = 1$ и $d_1 = 2$, а у второго $t_2 = 10$ и $d_2 = 10$. Сортировка по возрастанию запаса времени поместит второе задание на первое место в расписании, и первое задание создаст задержку 9. (Оно завершается во время 11, на 9 единиц позже предельного времени.) С другой стороны, если начать с планирования первого задания, то оно завершится вовремя, а второе задание ограничится задержкой 1.

Тем не менее существует столь же простой жадный алгоритм, который всегда приводит к оптимальному решению. Задания просто сортируются в порядке воз-

растания предельного времени d_i и планируются в этом порядке. Это правило основано на интуитивно понятном принципе: задания с более ранним предельным временем завершаются в первую очередь. В то же время немного трудно поверить, что этот алгоритм всегда выдает оптимальные решения, — просто потому, что он вообще не рассматривает длину заданий. Ранее мы уже раскритиковали метод сортировки длин на основе того, что он игнорировал половину входных данных (предельное время), а сейчас рассматриваем решение, которое теряет другую половину данных. Тем не менее правило первоочередного выбора самого раннего предельного времени выдает оптимальные решения, и сейчас мы это докажем.

Но сначала введем обозначение, которое пригодится при рассмотрении алгоритма. Переименовывая задания в случае необходимости, можно принять, что задания помечены в порядке следования их предельного времени, то есть выполняется условие

$$d_1 \leq \ldots \leq d_n.$$

Все задания просто планируются в этом порядке. И снова пусть s является начальным временем для всех заданий. Задание 1 начинается во время $s = s(1)$ и заканчивается во время $f(1) = s(1) + t_1$; задание 2 начинается во время $s(2) = f(1)$ и завершается во время $f(2) = s(2) + t_2$ и т. д. Время завершения последнего спланированного задания будет обозначаться f. Ниже приведена формулировка алгоритма на псевдокоде.

```
Упорядочить задания по предельному времени
Для простоты считается, что d_1 ≤ ... ≤ d_n
В исходном состоянии f = s
Рассмотреть задания i = 1, ..., n в таком порядке
   Назначить задание i временному интервалу от s(i) = f до f(i) = f + t_i
   Присвоить f = f + t_i
Конец
Вернуть множество спланированных интервалов [s(i), f(i)] для i = 1, ...,n
```

Анализ алгоритма

Чтобы рассуждать об оптимальности алгоритма, сначала следует заметить, что он не оставляет интервалов, когда машины простаивают, хотя еще остались задания. Время, проходящее в таких интервалах, будет называться *временем простоя*: работа еще осталась, но по какой-то причине машина ничего не делает. У расписания A, построенного по нашему алгоритму, время простоя отсутствует; но также очень легко понять, что существует оптимальное расписание, обладающее таким свойством. Доказательство этого утверждения не приводится.

(4.7) Существует оптимальное расписание без времени простоя.

Как теперь доказать, что наше расписание A оптимально, то есть его максимальная задержка L настолько мала, насколько это возможно? Как и в предыдущих случаях, начнем с рассмотрения оптимального расписания O. Нашей целью будет постепенная модификация O, которая будет сохранять оптимальность на каждом

шаге, но в конечном итоге преобразует его в расписание, идентичное расписанию A, найденному жадным алгоритмом. Такой метод анализа называется *заменой*, и вскоре вы увидите, что он весьма эффективен для описания жадных алгоритмов.

Для начала попробуем охарактеризовать полученные расписания. Будем говорить, что расписание A' содержит *инверсию*, если задание i с предельным временем d_i предшествует другому заданию j с более ранним предельным временем $d_j < d_i$. Обратите внимание: по определению расписание A, построенное нашим алгоритмом, не содержит инверсий. Если существует несколько заданий с одинаковыми значениями предельного времени, то это означает, что может существовать несколько разных расписаний без инверсий. Тем не менее мы можем показать, что все эти расписания обладают одинаковой максимальной задержкой L.

(4.8) Все расписания, не содержащие инверсий и простоев, обладают одинаковой максимальной задержкой.

Доказательство. Если два разных расписания не содержат ни инверсий, ни простоев, то задания в них могут следовать в разном порядке, но при этом различаться будет только порядок заданий с одинаковым предельным временем. Рассмотрим такое предельное время d. В обоих расписаниях задания с предельным временем d планируются последовательно (после всех заданий с более ранним предельным временем и до всех заданий с более поздним предельным временем). Среди всех заданий с предельным временем d последнее обладает наибольшей задержкой, причем эта задержка не зависит от порядка заданий. ■

Главным шагом демонстрации оптимальности алгоритма будет установление наличия оптимального расписания, не содержащего инверсий и простоев. Для этого мы начнем с любого оптимального расписания, не имеющего времени простоя; затем оно будет преобразовано в расписание без инверсий без увеличения максимальной задержки. Следовательно, расписание, полученное после этого преобразования, также будет оптимальным.

(4.9) Существует оптимальное расписание, не имеющее инверсий и времени простоя.

Доказательство. Согласно (4.7), существует оптимизация расписание O без времени простоя. Доказательство будет состоять из серии утверждений, первое из которых доказывается легко.

(а) Если O содержит инверсию, то существует такая пара заданий i и j, для которых j следует в расписании немедленно после i и $d_j < d_i$.

Рассмотрим инверсию, в которой задание a стоит в расписании где-то до задания b и $d_a > d_b$. При перемещении в порядке планирования заданий от a к b где-то встретится точка, в которой предельное время уменьшается в первый раз. Она соответствует паре последовательных заданий, образующих инверсию.

Теперь допустим, что O содержит как минимум одну инверсию, и в соответствии с (а) пусть i и j образуют пару инвертированных запросов, занимающих последовательные позиции в расписании. Меняя местами заявки i и j в расписании O, мы уменьшим количество инверсий в O на 1. Пара (i, j) создавала инверсию

в O, перестановка эту инверсию устраняет, и новые инверсии при этом не создаются. Таким образом,

(b) После перестановки i и j образуется расписание, содержащее на 1 инверсию меньше.

В этом доказательстве труднее всего обосновать тот факт, что расписание после устранения инверсии также является оптимальным.

(c) Максимальная задержка в новом расписании, полученном в результате перестановки, не превышает максимальной задержки O.

Очевидно, что если удастся доказать (c), то задачу можно считать выполненной. Исходное расписание O может иметь не более $\binom{n}{2}$ инверсий (если инвертированы все пары), а следовательно, после максимум $\binom{n}{2}$ перестановок мы получим оптимальное расписание без инверсий. ■

Итак, докажем (c) и продемонстрируем, что перестановка пары последовательных инвертированных заданий не приведет к возрастанию максимальной задержки L расписания.

Доказательство (c). Введем вспомогательное обозначение для описания расписания O: предположим, каждая заявка r планируется на интервал времени $[s(r), f(r)]$ и обладает задержкой l'_r. Пусть $L' = \max_r l'_r$ обозначает максимальную задержку этого расписания. Обозначим расписание с перестановкой $\overline{O}$; обозначения $\overline{s}(r)$, $\overline{f}(r)$, $\overline{l}_r$ и $\overline{L}$ будут представлять соответствующие характеристики расписания с перестановкой.

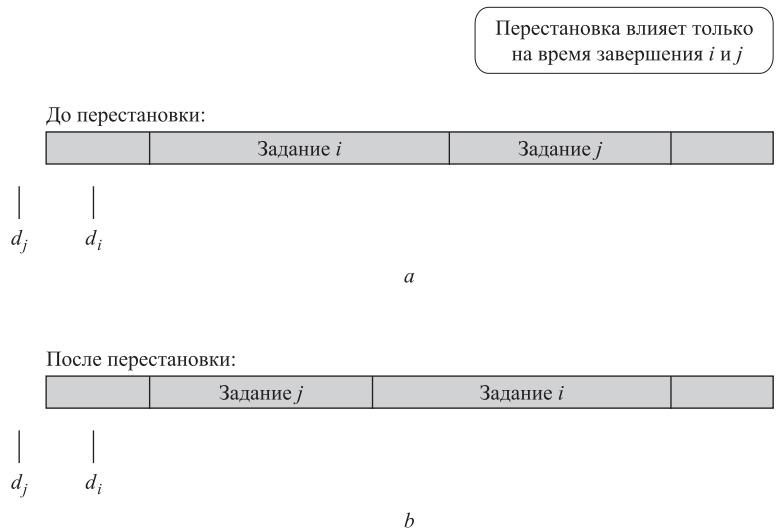


Рис. 4.6. Эффект перестановки двух последовательных заданий с инверсией

Теперь вернемся к двум смежным инвертированным заданиям *i* и *j*. Ситуация выглядит примерно так, как показано на рис. 4.6: время завершения *j* до перестановки в точности равно времени завершения *i* после перестановки. Таким образом, все задания, кроме *i* и *j*, в двух расписаниях завершаются одновременно. Кроме того, задание *j* в новом расписании завершится раньше, а следовательно, перестановка не увеличивает задержку задания *j*.

Следовательно, остается беспокоиться только о задании *i*: его задержка могла увеличиться. Что, если это привело к возрастанию максимальной задержки всего расписания? После перестановки задание *i* завершается во время $f(j)$, в которое завершалось задание *j* в расписании *O*. Если задание *i* «опаздывает» в новом расписании, его задержка составляет $l_i = f(i) - d_i = f(j) - d_i$. Но здесь принципиально то, что *i* не может задерживаться в расписании O больше, чем задерживалось задание *j* в расписании *O*. А конкретнее, из нашего предположения $d_i > d_j$ следует, что

$$l_i = f(j) - d_i < f(j) - d_j = l'_j.$$

Так как задержка расписания *O* была равна $L' \geq l'_j > l_i$, из этого следует, что перестановка не увеличивает максимальную задержку расписания. ■

Из этого немедленно следует оптимальность нашего жадного алгоритма.

(4.10) Расписание *A*, построенное жадным алгоритмом, обеспечивает оптимальную максимальную задержку *L*.

Доказательство. Пункт (4.9) доказывает, что оптимальное расписание без инверсий существует. С другой стороны, согласно (4.8), все расписания без инверсий имеют одинаковую максимальную задержку, так что расписание, построенное жадным алгоритмом, является оптимальным. ■

Расширения

Существует много разных обобщений задачи планирования. Например, мы предположили, что все задания были готовы к запуску, начиная с общего начального времени *s*. Естественная, но более сложная версия этой задачи содержит заявки *i*, которые, в дополнение к предельному времени d_i и запрашиваемому времени t_i, также содержат минимально возможное начальное время r_i. Самое раннее возможное начальное время обычно называется *временем разблокировки*. Задачи с временем разблокировки естественным образом встречаются в ситуациях с заявками вида: «Можно ли зарезервировать аудиторию для проведения двухчасовой лекции в интервале от 13 до 17 часов?» Наше доказательство того, что жадный алгоритм находит оптимальное решение, принципиально зависит от факта доступности всех заявок в общее начальное время *s*. (А вы видите, где именно?) К сожалению, как будет показано далее в главе 8, для более общего варианта задачи найти оптимальное решение намного сложнее.

4.3. Оптимальное кэширование: более сложный пример замены

А теперь рассмотрим задачу с обработкой последовательности заявок разной формы и разработаем алгоритм, анализ которого требует более хитроумного применения метода замены. Речь идет о задаче поддержания кэша.

Задача

Чтобы понять принцип кэширования, рассмотрим следующую ситуацию. Вы работаете над длинной научной статьей, а жесткие правила вашей библиотеки позволяют брать не более восьми книг одновременно. Вы знаете, что восьми книг вам не хватит, но в любой момент времени вам хотелось бы иметь доступ к восьми книгам, наиболее актуальным на данный момент. Как определить, какие книги следует заказать, когда вернуть часть из них в обмен на другие и как свести к минимуму количество обменов книг в библиотеке?

Именно такая ситуация возникает при работе с иерархиями памяти: существует небольшой объем данных, к которым можно обращаться очень быстро, и большой объем данных, для обращения к которым требуется больше времени; вы должны решить, какие данные следует держать «под рукой».

Иерархии памяти повсеместно встречались на компьютерах с первых дней их истории. Для начала стоит отметить, что обращения к данным в основной памяти происходят существенно быстрее, чем обращения к данным на жестком диске; с другой стороны, диск обладает много большей емкостью. А это означает, что часто используемые данные лучше хранить в основной памяти и обращаться к диску как можно реже. Тот же принцип на качественном уровне используется встроенными кэшами современных процессоров. Обращение к содержимому осуществляется за несколько тактов, и выборка данных из кэша происходит намного быстрее, чем выборка из основной памяти. Так появляется еще один уровень иерархии; чтение из малого кэша осуществляется быстрее, чем чтение из основной памяти, а последняя работает быстрее диска (но имеет меньший объем). Расширения этой иерархии также встречаются во многих других ситуациях. Например, при работе в браузере диск часто выполняет функции кэша для часто посещаемых веб-страниц, так как чтение данных с диска происходит намного быстрее, чем загрузка по Интернету.

Общим термином «кэширование» называется процесс хранения малых объемов данных в быстрой памяти, чтобы уменьшить время, которое тратится на взаимодействия с медленной памятью. В приведенных примерах встроенный кэш снижает необходимость в выборке данных из основной памяти, основная память выполняет функции кэша для диска, а диск служит кэшем для Интернета. (Подобно тому, как ваш рабочий стол служит кэшем для библиотеки, а отдельные факты, которые вам удается держать в голове без обращения к книге, образуют своего рода кэш для книг на столе.)

Чтобы кэширование было по возможности эффективным, желательно, чтобы при обращении к данным информация уже находилась в кэше. Для этого алгоритм

управления кэшем определяет, какая информация должна храниться, а какую информацию следует удалить из кэша, когда в него потребуется внести новые данные.

Конечно, с возникновением проблемы кэширования в разных контекстах приходится учитывать различные факторы, зависящие от технологии. Впрочем, здесь будет использоваться абстрактное представление задачи, лежащее в основе большинства контекстов. В нем рассматривается множество U из n фрагментов данных, хранящихся в основной памяти. Также существует более быстрая память — *кэш*, способная в любой момент времени хранить $k < n$ фрагментов данных. Будем считать, что кэш изначально содержит множество из k элементов. Мы получаем последовательность элементов данных $D = d_1, d_2, ..., d_m$ из U — это последовательность обращений к памяти, которую требуется обработать; при обработке следует постоянно принимать решение о том, какие k элементов должны храниться в кэше. Запрашиваемый элемент d_i очень быстро читается, если он уже находится в кэше; в противном случае его приходится копировать из основной памяти в кэш и, если кэш полон, вытеснять из него какой-то фрагмент данных, чтобы освободить место для d_i. Такая ситуация называется *кэш-промахом*; естественно, мы стремимся к тому, чтобы количество промахов было сведено к минимуму.

Итак, для конкретной последовательности обращений к памяти алгоритм управления кэшем определяет *план вытеснения* (какие элементы в каких точках последовательности должны вытесняться из кэша), а последний определяет содержимое кэша и частоту промахов. Рассмотрим пример этого процесса.

✦ Допустим, имеются три элемента $\{a, b, c\}$, размер кэша $k = 2$ и последовательность

$$a, b, c, b, c, a, b.$$

В исходном состоянии кэш содержит элементы a и b. Для третьего элемента последовательности можно вытеснить a, чтобы включить c в кэш; на шестом элементе можно вытеснить c, чтобы вернуть a; таким образом, вся последовательность включает два промаха. Если поразмыслить над этой последовательностью, становится понятно, что любой план вытеснения для этой последовательности должен включать как минимум два промаха.

В условиях реальных вычислений алгоритм управления кэшем должен обрабатывать обращения к памяти $d_1, d_2, ...$, не располагая информацией о том, какие обращения встретятся в будущем; но для оценки качества алгоритма системные аналитики с первых дней старались понять природу оптимального решения задачи кэширования. Если известна полная последовательность S обращений к памяти, какой план вытеснения обеспечит минимальное количество кэш-промахов?

Разработка и анализ алгоритма

В 1960-х годах Лес Белади показал, что следующее простое правило всегда приводит к минимальному количеству промахов:

```
Когда элемент d_i должен быть внесен в кэш,
   вытеснить элемент, который будет использоваться позднее всех остальных
```

Назовем его «алгоритмом отдаленного использования». Когда приходит время вытеснить данные из кэша, алгоритм перебирает все элементы в кэше и выбирает тот из них, который будет использоваться как можно позднее.

Это очень естественный алгоритм. В то же время факт его оптимальности для всех последовательностей не столь очевиден, как может показаться на первый взгляд. Почему нужно вытеснять элемент, который используется в самом отдаленном будущем, вместо, скажем, элемента, который будет реже всех использоваться в будущем? Или рассмотрим последовательность вида

$$a, b, c, d, a, d, e, a, d, b, c$$

с $k = 3$ и элементами $\{a, b, c\}$, изначально находящимися в кэше. Правило отдаленного использования создаст план S, которое вытесняет c на четвертом шаге и b на седьмом шаге. Однако существуют и другие планы вытеснения, не уступающие этому. Например, план S', который вытесняет b на четвертом шаге и c на седьмом шаге, обеспечивает такое же количество промахов. Очень легко найти ситуации, в которых планы, построенные по другим правилам, также будут оптимальными; не может ли оказаться, что при такой гибкости отступление от правила отдаленного использования обеспечит реальный выигрыш где-то в конце последовательности? Например, на седьмом шаге нашего примера план S' вытесняет элемент (c), который используется позднее, чем элемент, вытесненный в этой точке алгоритмом отдаленного использования, так как последний вытеснил c ранее.

Это лишь некоторые факторы, которые необходимо учесть, прежде чем делать вывод об оптимальности алгоритма отдаленного использования. Если подумать над приведенным примером, вы быстро поймете: на самом деле несущественно, какой из элементов — b или c — будет вытеснен на четвертом шаге, потому что второй элемент будет вытеснен на седьмом; получив план, в котором элемент b вытесняется первым, вы можете поменять местами вытеснение b и c без изменения эффективности. Эта причина — замена одного решения другим — дает начальное представление о *методе замены*, который используется для доказательства оптимальности правила отдаленного использования.

Прежде чем углубляться в анализ, стоит прояснить один важный момент. Все алгоритмы управления кэшем, упоминавшиеся выше, строят планы, которые включают элемент d в кэш на шаге i, если данные d запрашиваются на шаге i и d еще не находится в кэше. Назовем такой план *сокращенным* — он выполняет минимальный объем работы, необходимый для заданного шага. В общем случае можно представить себе алгоритм, который строит несокращенные планы, а элементы включаются в кэш на тех шагах, на которых они не запрашиваются. Теперь мы покажем, что для любого несокращенного плана существует сокращенный план, который не уступает ему.

Пусть S — план, который может не являться сокращенным. Новый план $\overline{S}$ — сокращение S — определяется следующим образом: на каждом шаге i, на котором S включает в кэш незапрошенный элемент d, наш алгоритм построения $\overline{S}$ «притво-

ряется», что он это делает, но в действительности оставляет d в основной памяти. Фактически d попадает в кэш на следующем шаге j после того, как элемент d был запрошен. В этом случае кэш-промах, инициированный $\overline{S}$ на шаге j, может быть отнесен на счет более ранней операции с кэшем, выполненной S на шаге i. Следовательно, мы приходим к следующему факту:

(4.11) Сокращенный план $\overline{S}$ заносит в кэш не больше элементов, чем план S.

Заметьте, что для каждого сокращенного плана количество элементов, включаемых в кэш, точно совпадает с количеством промахов.

Доказательство оптимальности алгоритма отдаленного использования

Воспользуемся методом замены для доказательства оптимальности алгоритма отдаленного использования. Рассмотрим произвольную последовательность обращений к памяти D; пусть S_{FF} обозначает план, построенный алгоритмом отдаленного использования, а S^* — план с минимально возможным количеством промахов. Сейчас мы постепенно «преобразуем» S^* в S_{FF}, обрабатывая одно решение по вытеснению за раз, без увеличения количества промахов.

Ниже приведен базовый факт, который будет использоваться для выполнения одного шага преобразования.

(4.12) Пусть S — сокращенный план, который принимает те же решения по вытеснению, что и S_{FF}, в первых j элементах последовательности для некоторого j. Тогда существует сокращенный план S', который принимает те же решения, что и S_{FF}, в первых $j + 1$ элементах и создает не больше промахов, чем S.

Доказательство. Рассмотрим $(j + 1)$-е обращение к элементу $d = d_{j+1}$. Так как S и S_{FF} до этого момента были согласованы, содержимое кэша в них не различается. Если d находится в кэше в обоих планах, то решения по вытеснению не требуются (оба плана являются сокращенными), так что S согласуется с S_{FF} на шаге $j + 1$, и, следовательно, $S' = S$. Аналогичным образом, если элемент d должен быть включен в кэш, но S и S_{FF} вытеснят один и тот же элемент, чтобы освободить место для d, и снова $S' = S$.

Итак, интересная ситуация возникает тогда, когда d необходимо добавить в кэш, причем для этого S вытесняет элемент f, а S_{FF} вытесняет элемент $e \neq f$. Здесь S и S_{FF} уже не согласуются к шагу $j + 1$, потому что у S в кэше находится e, а у S_{FF} в кэше находится f. А это означает, что для построения S' необходимо сделать что-то нетривиальное.

Сначала нужно сделать так, чтобы план S' вытеснял e вместо f. Затем нужно убедиться в том, что количество промахов у S' не больше S. Простым решением было бы согласование S' с S в оставшейся части последовательности; однако это невозможно, так как S и S', начиная с этого момента, имеют разное содержимое кэша. Поэтому мы должны постараться привести кэш S' в такое же состояние, как у S, не создавая ненужных промахов. Когда содержимое кэша будет совпадать, можно будет завершить построение S', просто повторяя поведение S.

А если говорить конкретнее, от запроса $j + 2$ и далее S' ведет себя в точности как S, пока впервые не будет выполнено одно из следующих условий:

(i) Происходит обращение к элементу $g \neq e, f$, который не находится в кэше S, и S вытесняет e, чтобы освободить для него место. Так как S' и S различаются только в e и f, это означает, что g также не находится в кэше S'; тогда из S' вытесняется f и кэши S и S' совпадают. Таким образом, в оставшейся части последовательности S' ведет себя точно так же, как S.

(ii) Происходит обращение к f, и S вытесняет элемент e'. Если $e' = e$, все просто: S' просто обращается к f из кэша, и после этого шага кэши S и S' совпадают. Если $e' \neq e$, то S' также вытесняет e' и добавляет e из основной памяти; в результате S и S' тоже имеют одинаковое содержимое кэша. Однако здесь необходима осторожность, так как S' уже не является сокращенным планом: элемент e переносится в кэш до того, как в нем возникнет прямая необходимость. Итак, чтобы завершить эту часть построения, мы преобразуем S' в сокращенную форму $\overline{S'}$, используя (4.11); количество элементов, включаемых S', при этом не увеличивается, и S' по-прежнему согласуется с S_{FF} на шаге $j + 1$.

Итак, в обоих случаях мы получаем новый сокращенный план S', который согласуется с S_{FF} на первых $j + 1$ элементах и обеспечивает не большее количество промахов, чем S. И здесь принципиально (в соответствии с определяющим свойством алгоритма отдаленного использования) то, что один из этих двух случаев возникнет до обращения к e. Дело в том, что на шаге $j + 1$ алгоритм отдаленного использования вытеснил элемент (e), который понадобится в самом отдаленном будущем; следовательно, обращению к e должно предшествовать обращение к f, и тогда возникнет ситуация (ii). ■

Этот результат позволяет легко завершить доказательство оптимальности. Мы начнем с оптимального плана S^* и воспользуемся (4.12) для построения плана S_1, который согласуется с S_{FF} на первом шаге. Затем мы продолжаем применять (4.12) индуктивно для $j = 1, 2, 3, ..., m$, строя планы S_j, согласующиеся с S_{FF} на первых j шагах. Каждый план не увеличивает количество промахов по сравнению с предыдущим, а по определению $S_m = S_{FF}$, поскольку планы согласуются на всей последовательности.

Таким образом, получаем:

(4.13) S_{FF} порождает не больше промахов, чем любой другой план S^*, а следовательно, является оптимальным.

Расширения: кэширование в реальных рабочих условиях

Как упоминалось в предыдущем подразделе, оптимальный алгоритм Белади предоставляет контрольную точку для оценки эффективности кэширования; однако на практике решения по вытеснению должны приниматься «на ходу», без информации о будущих обращениях.

Эксперименты показали, что лучшие алгоритмы кэширования в описанной ситуации представляют собой вариации на тему принципа LRU (Least-Recently-Used), по которому из кэша вытесняется элемент с наибольшим временем, прошедшим с момента последнего обращения.

Если задуматься, речь идет о том же алгоритме Белади с измененным направлением времени, — только наибольший продолжительный промежуток времени относится к прошлому, а не к будущему. Он эффективно работает, потому что для приложений обычно характерна *локальность обращений*: выполняемая программа, как правило, продолжает обращаться к тем данным, к которым она уже обращалась ранее. (Легко придумать аномальные исключения из этого принципа, но они относительно редко встречаются на практике.) По этой причине в кэше обычно хранятся данные, которые использовались недавно.

Спустя долгое время после практического принятия алгоритма LRU Слитор и Тарьян показали, что для эффективности LRU можно предоставить теоретический анализ, ограничивающий количество промахов относительно алгоритма отдаленного использования. Мы рассмотрим этот анализ, равно как и анализ рандомизированной разновидности LRU, при возвращении к задаче кэширования в главе 13.

4.4. Кратчайшие пути в графе

Некоторые базовые алгоритмы графов основаны на жадных принципах. В этом разделе рассматривается жадный алгоритм для задачи поиска кратчайших путей, а в следующем разделе будет рассматриваться построение остовных деревьев минимальной стоимости.

Задача

Как вы уже знаете, графы часто используются для моделирования сетей, в которых происходит переход от одной точки к другой: например, дорог на транспортных развязках или каналов связи с промежуточными маршрутизаторами. В результате задача поиска кратчайшего пути между узлами в графе вошла в число базовых алгоритмических задач. Даны узлы u и v; какой из путей u-v является кратчайшим? Также можно запросить более подробную информацию: задан *начальный узел* s, как выглядят кратчайшие пути от s ко всем остальным узлам?

Формальная постановка задачи кратчайших путей выглядит так: имеется направленный граф $G = (V, E)$ с обозначенным начальным узлом s. Предполагается, что в графе существует путь от s к любому другому узлу G. Каждому ребру e назначена длина $\ell_e \geq 0$, которая обозначает время (или расстояние, или затраты), необходимое для перемещения по e. Для пути P длина P (обозначаемая $\ell(P)$) вычисляется как сумма длин всех ребер, входящих в P. Наша задача — определить кратчайший путь от s к каждому из других узлов графа. Следует упомянуть, что хотя задача сформулирована для направленного графа, случай ненаправленного графа обе-

спечивается простой заменой каждого ненаправленного ребра $e = (u, v)$ длины ℓ_e двумя направленными ребрами (u, v) и (v, u), каждое из которых имеет длину ℓ_e.

Разработка алгоритма

В 1959 году Эдгар Дейкстра предложил очень простой жадный алгоритм для решения задачи нахождения кратчайшего пути с одним источником. Мы начнем с описания алгоритма, который просто находит длину кратчайшего пути от s к каждому из остальных узлов графа; после этого можно будет легко получить сами пути. Алгоритм хранит множество S вершин u, для которых было определено расстояние $d(u)$ кратчайшего пути от s; это множество представляет «исследованную» часть графа. Изначально $S = \{s\}$, а $d(s) = 0$. Затем для каждого узла $v \in V - S$ определяется кратчайший путь, который может быть построен перемещением по пути в исследованной части S к некоторому $u \in S$, за которым следует одно ребро (u, v). Далее рассматривается характеристика $d'(v) = \min_{e=(u,v):u\in S} d(u) + l_e$. Алгоритм выбирает узел $v \in V - S$, для которого эта величина минимальна, добавляет v в S и определяет $d(v)$ как значение $d'(v)$.

```
Алгоритм Дейкстры (G, ℓ)
Пусть S – множество исследованных узлов
   Для каждого u ∈ S хранится расстояние d(u)
Изначально S = {s} и d(s) = 0
Пока S ≠ V
   Выбрать узел v ∉ S с хотя бы одним ребром из S, для которого
      значение d'(v) = min_{e=(u, v):u∈S} d(u) + ℓ_e минимально
   Добавить v в S и определить d(v) = d'(v)
Конец Пока
```

Получить пути s–u для расстояний, вычисленных алгоритмом Дейкстры, несложно. При добавлении каждого узла v во множество S просто сохраняется ребро (u, v), для которого было достигнуто значение $\min_{e=(u,v):u\in S} d(u) + l_e$. Путь P_v неявно представляется этими ребрами: если (u, v) — ребро, сохраненное для v, то P_v — это (рекурсивно) путь P_u, за которым следует одно ребро (u, v). Иначе говоря, чтобы построить P_v, мы просто начинаем с v; переходим по ребру, сохраненному для v, в обратном направлении к u; затем переходим по ребру, сохраненному для u, в обратном направлении к его предшественнику, и т. д., пока не доберемся до s. Заметьте, что узел s должен быть достижим, так как обратный переход от v посещает узлы, добавлявшиеся в S ранее и ранее.

Чтобы лучше понять, как работает алгоритм, рассмотрите «снимок» его выполнения на рис. 4.7. На момент создания «снимка» были выполнены две итерации: первая добавила узел u, а вторая — узел v. В предстоящей итерации будет добавлен узел x, потому что с ним достигается наименьшее значение $d'(x)$; благодаря ребру (u, x) имеем $d'(x)=d(u) + l_{ux} = 2$. Обратите внимание: попытка добавить y или z в множество S в этой точке приведет к неправильным расстояниям кратчайших путей; в конечном итоге они будут добавлены из-за ребер от x.

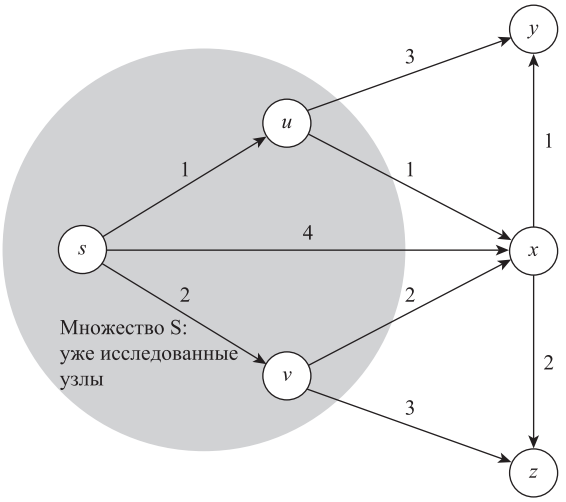


Рис. 4.7. Снимок выполнения алгоритма Дейкстры. Следующим узлом, добавляемым в множество S, будет узел x (из-за пути через u)

Анализ алгоритма

Из примера видно, что алгоритм Дейкстры работает правильно и избегает ловушек: добавление в множество S неправильного узла может привести к переоценке расстояния кратчайшего пути к этому узлу. Остается ответить на вопрос: всегда ли при добавлении алгоритмом Дейкстры узла v мы получаем истинное расстояние кратчайшего пути к v?

Ответ доказывает правильность алгоритма и то, что пути P_u действительно являются кратчайшими. Алгоритм Дейкстры является жадным в том смысле, что он всегда строит кратчайший новый путь s–v, состоящий из пути из S, за которым следует одно ребро. Для доказательства правильности будет использована разновидность первого метода анализа: мы продемонстрируем, что это решение «идет впереди» всех остальных. Для этого мы покажем методом индукции, что каждый выбираемый им путь к узлу v будет короче любого другого пути к v.

(4.14) Рассмотрим множество S в любой точке выполнения алгоритма. Для каждого узла $u \in S$ путь P_u является кратчайшим путем s–u.

Обратите внимание: этот факт немедленно доказывает правильность алгоритма Дейкстры, так как его можно применить к моменту завершения алгоритма, когда S включает все узлы.

Доказательство. Для доказательства будет использоваться индукция по размеру S. Случай $|S| = 1$ тривиален: в этом состоянии $S = \{s\}$ и $d(s) = 0$. Предположим, утверждение истинно для $|S| = k$ при некотором значении $k \geq 1$; теперь S увеличивается до размера $k + 1$ добавлением узла v. Пусть (u, v) — последнее ребро пути s–v P_v.

Согласно индукционной гипотезе, P_u является кратчайшим путем s–u для всех $u \in S$. Теперь возьмем любой другой путь s–v P; мы хотим показать, что он по крайней мере не короче P_v. Чтобы достичь узла v, путь P должен где-то выйти из множества S; пусть y — первый узел P, не входящий в S, и $x \in S$ — узел, непосредственно предшествующий y.

Ситуация изображена на рис. 4.8, и суть доказательства очень проста: путь P не может быть короче P_v, потому что он уже был не короче P_y к моменту выхода из множества S. Действительно, в итерации $k + 1$ алгоритм Дейкстры уже должен был рассмотреть добавление узла y в множество S по ребру (x, y) и отклонить этот вариант в пользу добавления v. Это означает, что не существует пути от s к y через x, который был бы короче P_v. Но подпуть P до узла y является как раз таким путем, так что этот путь по крайней мере не короче P_v. А поскольку длины ребер неотрицательные, полный путь P по крайней мере не короче P_v.

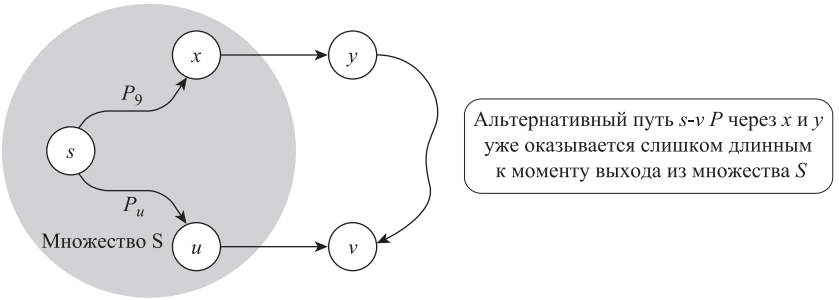


Рис. 4.8. Кратчайший путь P_v и альтернативный путь s–v P через узел y

Это полное доказательство; также можно изложить аргумент из предыдущего абзаца с использованием следующих неравенств. Пусть P' — подпуть P от s до x. Так как $x \in S$, из индукционной гипотезы известно, что P_x является кратчайшим путем s-x (с длиной $d(x)$), поэтому $\ell(P') \geq \ell(P_x) = d(x)$. Следовательно, подпуть P к узлу y имеет длину $l(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq d'(y)$, и полный путь P по крайней мере не короче этого подпути. Наконец, раз алгоритм Дейкстры выбрал v на этой итерации, мы знаем, что $d'(y) \geq d'(v) = \ell(P_v)$. Объединение этих неравенств показывает, что $\ell(P) \geq \ell(P') + \ell(x, y) \geq \ell(P_v)$. ■

Приведем пару замечаний относительно алгоритма Дейкстры и его анализа. Во-первых, алгоритм не всегда находит кратчайшие пути, если некоторые ребра имеют отрицательную длину. (А вы видите, где нарушается работа алгоритма?) Отрицательная длина ребер встречается во многих задачах нахождения кратчайшего пути, и в таких ситуациях требуется более сложный алгоритм, предложенный Беллманом и Фордом. Мы рассмотрим этот алгоритм, когда займемся темой динамического программирования.

Второе замечание заключается в том, что алгоритм Дейкстры в некотором смысле еще проще приведенного описания. Алгоритм Дейкстры в действительности является «непрерывной» версией алгоритма поиска в ширину для обхода графа; это можно пояснить физической аналогией. Допустим, ребра G образуют

систему труб, заполненных водой и состыкованных в узлах; каждое ребро e обладает длиной ℓ_e и фиксированным поперечным сечением. Теперь допустим, что в узле s падает капля воды, в результате чего от s начинает расходиться волна. Так как волна распространяется от s с постоянной скоростью, расширяющаяся сфера волнового фронта достигает узлов в порядке возрастания их расстояния от s. Интуиция подсказывает, что путь, по которому проходит фронт для достижения любого узла v, является кратчайшим (и это действительно так). Как нетрудно убедиться, именно этот путь v будет найден алгоритмом Дейкстры, а расширяющаяся волна доходит до узлов в том же порядке, в котором они обнаруживаются алгоритмом Дейкстры.

Реализация и время выполнения

Обсуждение алгоритма Дейкстры завершается анализом его времени выполнения. Для графа с n узлами цикл состоит из $n - 1$ итераций, так как каждая итерация добавляет в S новый узел v. С эффективностью выбора правильного узла v дело обстоит сложнее. Первая мысль — при каждой итерации рассмотреть новый узел $v \in S$, перебрать все ребра между S и v с вычислением минимума $\min_{e=(u,\,v):u\in S} d(u) + \ell_e$, чтобы выбрать узел v с наименьшим значением. Для графа с m ребрами вычисление минимумов будет выполняться за время $O(m)$, так что полученная реализация будет выполняться за время $O(mn)$.

Правильный выбор структуры данных позволяет существенно улучшить результат. Прежде всего, значения минимумов $d'(v) = \min_{e=(u,\,v):u\in S} d(u) + \ell_e$ следует явно хранить для каждого узла $v \in V - S$, чтобы не вычислять их заново при каждой итерации. Также эффективность можно повысить посредством хранения узлов $V–S$ в приоритетной очереди с ключами $d'(v)$. Приоритетные очереди рассматривались в главе 2; эти структуры данных предназначены для хранения множеств из n элементов с ключами. Приоритетная очередь обеспечивает эффективное выполнение вставки элементов, удаления элементов, изменения ключей и извлечения элементов с минимальным ключом. Нам понадобятся третья и четвертая из перечисленных операций:`ChangeKey` и `ExtractMin`.

Как реализовать алгоритм Дейкстры с использованием приоритетной очереди? Узлы V помещаются в приоритетную очередь с ключом $d'(v)$ для $v \in V$. Для выбора узла v, который должен быть добавлен в множество S, понадобится операция `ExtractMin`. Чтобы понять, как обновлять ключи, рассмотрим итерацию, в которой узел v добавляется в S; пусть $w \notin S$ — узел, остающийся в приоритетной очереди. Что нужно сделать для обновления значения $d'(w)$? Если (v, w) не является ребром, то делать ничего не нужно: множество ребер, рассматриваемых в минимуме $\min_{e=(u,\,v):u\in S} d(u) + \ell_e$, в точности совпадает до и после добавления v в S. С другой стороны, если $e' = (v,w) \in E$, то новым значением ключа будет $\min(d'(w), d(v) + \ell_{e'})$. Если $d'(w) > d(v) + \ell_{e'}$, то нужно использовать операцию `ChangeKey` для соответствующего изменения ключа узла w. Операция `ChangeKey` может выполняться не более одного раза на ребро, при добавлении «хвоста» e' в S. Таким образом, мы получаем следующий результат.

(4.15) При использовании приоритетной очереди алгоритм Дейкстры для графа с n узлами и m ребрами может быть реализован с выполнением за время $O(m)$ с добавлением времени n операций `ExtractMin` и m операций `ChangeKey`.

При реализации приоритетной очереди на базе кучи (см. главу 2) все операции будут выполняться за время $O(\log n)$. Таким образом, общее время выполнения составит $O(m \log n)$.

4.5. Задача нахождения минимального остовного дерева

А теперь мы воспользуемся методом замены в контексте второй фундаментальной задачи графов: задачи нахождения минимального остовного дерева.

Задача

Имеется множество точек $V = \{v_1, v_2, ..., v_n\}$; требуется построить для них коммуникационную сеть. Сеть должна быть связной (то есть в ней должен существовать путь между любой парой узлов), но при соблюдении этого требования сеть должна быть построена с минимальными затратами.

Для некоторых пар (v_i, v_j) можно построить прямой канал с определенными затратами $c(v_i, v_j) > 0$. Таким образом, множество всех возможных каналов представляется графом $G = (V, E)$, в котором с каждым ребром $e = (v_i, v_j)$ связывается положительная стоимость c_e. Задача заключается в нахождении такого подмножества ребер $T \subseteq E$, чтобы граф (V, T) был связным, а общая стоимость $\sum_{e \in T} c_e$ была как можно меньшей. (Предполагается, что полный граф G является связным; в противном случае решение невозможно.)

Ниже приведено базовое наблюдение.

(4.16) Пусть T — решение описанной выше задачи проектирования сети с минимальной стоимостью. В этом случае (V, T) является деревом.

Доказательство. По определению граф (V, T) должен быть связным; мы покажем, что он также не содержит циклов. Предположим, граф содержит цикл C, а e — любое ребро C. Утверждается, что граф $(V, T - \{e\})$ остается связным, потому что любой путь, который использовал ребро e, теперь может воспользоваться «обходным путем» по оставшейся части цикла C. Из этого следует, что $(V, T - \{e\})$ также является действительным решением задачи, и при этом оно имеет меньшую стоимость, — обнаружено противоречие. ■

Если допустить, что некоторые ребра могут иметь нулевую стоимость (то есть предполагается лишь то, что значения c_e не отрицательны), то решение задачи проектирования сети с минимальной стоимостью может содержать лишние ребра — такие ребра имеют нулевую стоимость и при желании могут быть удалены.

Начиная с любого оптимального решения, мы можем удалять ребра в циклах до тех пор, пока не возникнет дерево; с неотрицательными ребрами общая стоимость при этом не увеличится.

Подмножество $T \subseteq E$ будет называться *остовным деревом* графа G, если (V, T) является деревом. Утверждение (4.16) указывает на то, что цель задачи проектирования сети можно переформулировать как задачу поиска остовного дерева графа с минимальной стоимостью; по этой причине эта задача обычно называется задачей нахождения *минимального остовного дерева*. Во всех графах G, кроме очень простых, количество разных остовных деревьев растет по экспоненциальному закону, а их структуры могут сильно отличаться друг от друга. Совершенно непонятно, как эффективно найти дерево с минимальной стоимостью среди всех этих вариантов.

Разработка алгоритма

Как и в приведенных выше задачах, для этой задачи нетрудно предложить несколько естественных жадных алгоритмов. Но к счастью, это один из тех случаев, в которых многие из проверяемых в первую очередь жадных алгоритмов оказываются правильными: все они решают задачу оптимально. Сейчас мы рассмотрим некоторые из этих алгоритмов и при помощи метода замены выясним, какие причины лежат в основе этого семейства простых оптимальных алгоритмов.

Итак, перед вами три жадных алгоритма, каждый из которых правильно находит минимальное остовное дерево.

- Один простой алгоритм начинает с пустого дерева и строит остовное дерево, последовательно вставляя ребра из E в порядке возрастания стоимости. При перемещении по ребрам в этом порядке каждое ребро e вставляется в том случае, если при добавлении к ранее вставленным ребрам оно не создает цикл. Если же, напротив, вставка e порождает цикл, то ребро e просто игнорируется, а выполнение алгоритма продолжается. Такое решение называется *алгоритмом Крускала*.
- Другой простой жадный алгоритм проектируется по аналогии с алгоритмом Дейкстры для путей, хотя на самом деле он определяется проще. Алгоритм начинает с корневого узла s и пытается наращивать дерево. На каждом шаге в уже существующее частичное дерево просто добавляется узел, который может быть присоединен с минимальной стоимостью.

  Или говоря конкретнее, алгоритм поддерживает множество $S \subseteq V$, для которого уже было построено остовное дерево. В исходном состоянии $S = \{s\}$. При каждой итерации S наращивается на один узел; при этом добавляется узел, минимизирующий «стоимость присоединения» $min_{e=(u, v):u \in S} c_e$, и ребро $e = (u, v)$, обеспечивающее этот минимум в остовном дереве. Такое решение называется *алгоритмом Прима*.
- Наконец, жадный алгоритм может представлять собой своего рода «алгоритм Крускала наоборот», а именно: он начинает с полного графа (V, E) и удаляет

ребра в порядке уменьшения стоимости. При переходе к каждому ребру *e* (начиная с самого дорогого) это ребро удаляется, если это не приводит к потере связности текущего графа. Обычно этого алгоритм называется *алгоритмом обратного удаления* (насколько нам известно, с именем конкретного человека он никогда не связывался).

На рис. 4.9 изображены первые четыре ребра, добавленные алгоритмами Прима и Крускала соответственно, для геометрического экземпляра задачи нахождения минимального остовного дерева, в котором стоимость каждого ребра пропорциональна геометрическому расстоянию на плоскости.

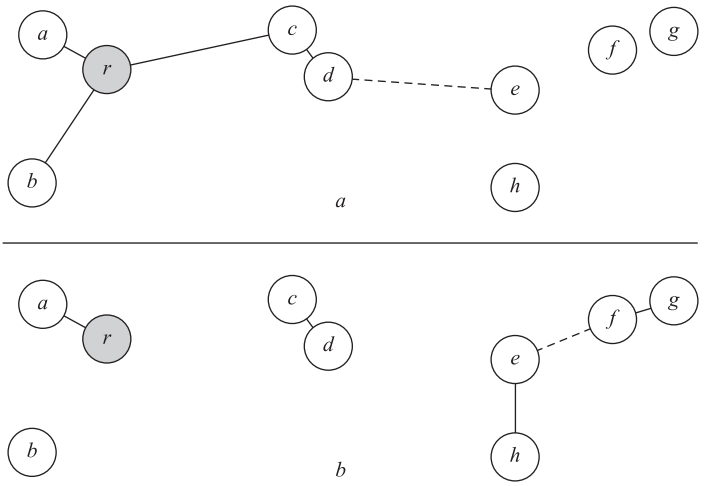


Рис. 4.9. Пример выполнения алгоритмов нахождения минимального остовного дерева: *a* — алгоритм Прима; *b* — алгоритм Крускала, для общих входных данных. Первые четыре ребра, добавленных в остовное дерево, обозначены сплошными линиями; следующее добавляемое ребро обозначено пунктиром

Тот факт, что каждый из этих алгоритмов гарантированно строит оптимальное решение, предполагает определенную «устойчивость» задачи нахождения минимального остовного дерева — ответ можно получить разными способами. Затем мы исследуем некоторые причины, из-за которых минимальные остовные деревья могут строиться настолько разными алгоритмами.

Анализ алгоритмов

Работа всех упомянутых алгоритмов основана на многократной вставке или удалении ребер из частичного решения. Итак, чтобы проанализировать их, желательно иметь под рукой базовые факты, определяющие, когда включение ребра в минимальное остовное дерево безопасно и, соответственно, когда ребро можно безопасно удалить на том основании, что оно не может входить в минимальное остовное дерево. Для данного анализа будет сделано упрощающее предположение, что все

стоимости ребер различаются (то есть в графе нет двух ребер с одинаковой стоимостью). Это предположение упрощает последующие рассуждения, причем как будет показано позднее в этом разделе, это предположение достаточно легко снимается.

Когда же включение ребра в минимальное остовное дерево безопасно? Важнейший факт, относящийся к вставке ребра, в дальнейшем будет называться «свойством сечения».

(4.17) Допустим, все стоимости ребер различаются. Пусть S — любое подмножество узлов, не пустое и не равное V, и пусть ребро $e = (v,w)$ — ребро минимальной стоимости, один конец которого принадлежит S, а другой — $V - S$. В этом случае ребро e входит в каждое минимальное остовное дерево.

Доказательство. Пусть T — остовное дерево, не содержащее e; необходимо показать, что стоимость T не является минимально возможной. Для этого мы воспользуемся методом замены: найдем в T ребро e' с большей стоимостью, чем у e, и с тем свойством, что замена e на e' приводит к другому остовному дереву. Полученное остовное дерево будет обладать меньшей стоимостью, чем у T, как и требовалось.

Таким образом, вся суть в том, чтобы найти ребро, которое можно успешно обменять с e. Вспомните, что концами e являются узлы v и w. T — остовное дерево, поэтому в T должен существовать путь из v в w. Предположим, начиная с v, мы последовательно переходим по узлам P; в P существует первый узел w', который принадлежит $V - S$. Пусть $v' \in S$ — узел, непосредственно предшествующий w' в P, а $e' = (v', w')$ — соединяющее их ребро. Следовательно, e' — ребро T, один конец которого принадлежит S, а другой — $V - S$. Ситуация для этой стадии доказательства представлена на рис. 4.10.

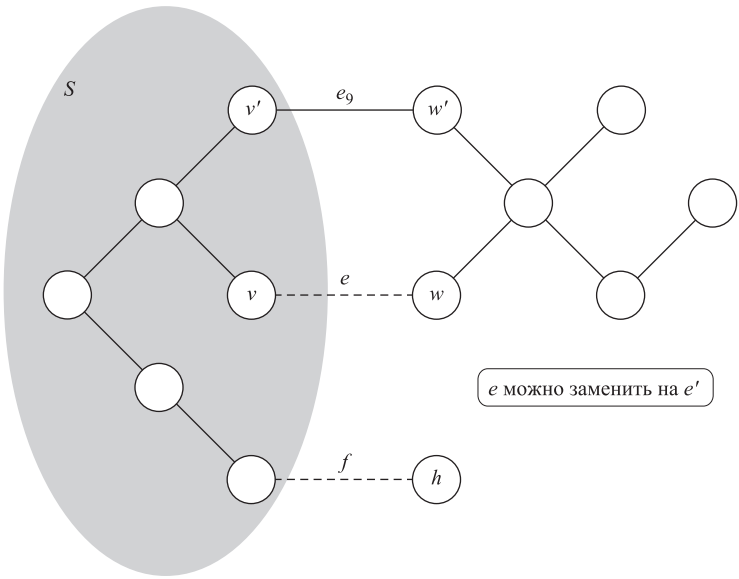


Рис. 4.10. Ребро e меняется с ребром e' в остовном дереве T, как описано в доказательстве (4.17)

Заменив e на e', мы получаем множество ребер $T' = T - \{e'\} \cup \{e\}$. Утверждается, что T' — остовное дерево. Очевидно, граф (V, T') является связным, поскольку связным является (V, T), а любой путь в (V, T), который использовал ребро $e' = (v', w')$, теперь может быть «перенаправлен» через (V, T') для прохождения части P от v' до v, ребра e и затем части P от w до w'. Чтобы понять, что граф (V, T') также является ациклическим, следует заметить, что единственный цикл в $(V, T' \cup \{e'\})$ состоит из e и пути P, но этот цикл не присутствует в (V, T') из-за удаления e'.

Ранее было отмечено, что один конец ребра e' принадлежит S, а другой — $V - S$. Но e является ребром с минимальной стоимостью, обладающим этим свойством, поэтому $c_e < c_{e'}$. (Неравенство строгое, потому что никакие два ребра не обладают одинаковой стоимостью.) Следовательно, общая стоимость T' меньше общей стоимости T, как и требуется. ■

Доказательство (4.17) немного сложнее, чем кажется на первый взгляд. Чтобы понять эту сложность, возьмем следующее (более короткое, но неправильное) обоснование для (4.17). Пусть T — остовное дерево, которое не содержит e. Так как T является остовным деревом, оно должно содержать ребро f, один конец которого принадлежит S, а другой — $V - S$. Так как e является самым «дешевым» ребром, обладающим таким свойством, выполняется условие $c_e < c_f$, а значит, $T - \{f\} \cup \{e\}$ — остовное дерево с меньшей стоимостью, чем у T.

Проблема с этим обоснованием связана не с утверждением о том, что f существует или что $T - \{f\} \cup \{e\}$ имеет меньшую стоимость, чем T. Проблема в том, что $T - \{f\} \cup \{e\}$ может не быть остовным деревом, как видно на примере ребра f на рис. 4.10. Утверждение (4.17) нельзя доказать простым выбором любого ребра в T, переходящего из S в $V - S$; необходимо принять меры к тому, чтобы найти правильное ребро.

Оптимальность алгоритмов Крускала и Прима

Теперь мы можем легко доказать оптимальность алгоритмов Крускала и Прима. Оба алгоритма включают ребро только в том случае, когда это включение оправдывается свойством сечения (4.17).

(4.18) Алгоритм Крускала строит минимальное остовное дерево графа G.

Доказательство. Рассмотрим любое ребро $e = (v, w)$, добавленное алгоритмом Крускала. Пусть S — множество всех узлов, к которым существует путь из v непосредственно перед добавлением e. Очевидно, $v \in S$, но и $w \in S$, так как добавление e не приводит к созданию цикла.

Более того, никакое ребро из S в $V - S$ еще не было обнаружено, так как любое такое ребро может быть добавлено без создания цикла, а значит, было бы добавлено алгоритмом Крускала. Следовательно, e — ребро с минимальной стоимостью, у которого один конец принадлежит S, а другой — $V - S$, и, согласно (4.17), оно принадлежит минимальному остовному дереву.

Итак, если нам удастся показать, что результат (V, T) алгоритма Крускала действительно является остовным деревом графа G, то дело будет сделано. Очевидно, (V, T) не содержит циклов, поскольку алгоритм проектировался для предотвраще-

ния создания циклов. Кроме того, если граф (V, T) не был связным, то существовало бы непустое множество узлов S (не равное всему множеству V), такое, что не существует ребра из S в $V - S$. Но это противоречит поведению алгоритма: мы знаем, что поскольку граф G является связным, между S и $V - S$ существует как минимум одно ребро, и алгоритм добавит первое из таких ребер при его обнаружении. ■

(4.19) Алгоритм Прима строит минимальное остовное дерево графа G.

Доказательство. Для алгоритма Прима также очень легко показать, что он добавляет только ребра, принадлежащие любому возможному минимальному остовному дереву. В самом деле, при каждой итерации алгоритма существует множество $S \subseteq V$, на базе которого было построено частичное остовное дерево, и добавляется узел v и ребро e, которые минимизируют величину $\min_{e=(u, v):u \in S} c_e$. По определению e является ребром с минимальной стоимостью, у которого один конец принадлежит S, а другой — $V - S$, поэтому по свойству сечения (4.17) оно присутствует в каждом минимальном остовном дереве.

Также тривиально показывается, что алгоритм Прима строит остовное дерево графа G — а следовательно, он строит минимальное остовное дерево. ■

Когда можно гарантировать, что ребро не входит в минимальное остовное дерево?

При удалении ребер критичен следующий факт, который мы будем называть «свойством цикла».

(4.20) Предполагается, что стоимости всех ребер различны. Пусть C — любой цикл в G, а ребро $e = (v, w)$ — ребро с максимальной стоимостью, принадлежащее C. В этом случае e не принадлежит никакому минимальному остовному дереву графа G.

Доказательство. Пусть T — остовное дерево, содержащее e; необходимо показать, что T не обладает минимальной возможной стоимостью. По аналогии с доказательством свойства сечения (4.17) мы воспользуемся методом замены, подставляя на место e ребро с меньшей стоимостью так, чтобы не разрушить остовное дерево.

Итак, мы сталкиваемся с тем же вопросом: как найти ребро с меньшей стоимостью, которое можно поменять местами с e? Начнем с удаления e из T; узлы при этом делятся на два подмножества: S, содержащее узел v; и $V - S$, содержащее узел w. Теперь один конец ребра, используемого вместо e, должен принадлежать S, а другой — $V - S$, чтобы снова объединить дерево.

Такое ребро можно найти переходом по циклу C. Ребра C, отличные от e, по определению образуют путь P, один конец которого находится в узле v, а другой в узле w. Если перейти по P от v к w, переход начнется в S и закончится в $V - S$, поэтому в P существует некоторое ребро e', соединяющее S с $V - S$. Ситуация изображена на рис. 4.11.

Теперь рассмотрим множество ребер $T' = T - \{e\} \cup \{e'\}$. По причинам, изложенным при доказательстве свойства сечения (4.17), граф (V, T') является связным и не содержит циклов, поэтому T' является остовным деревом G. Кроме того, поскольк-

ку e — самое «дорогое» ребро цикла C, а e' принадлежит C, стоимость e' должна быть ниже, чем у e, а следовательно, стоимость T' ниже стоимости T, как и требовалось. ■

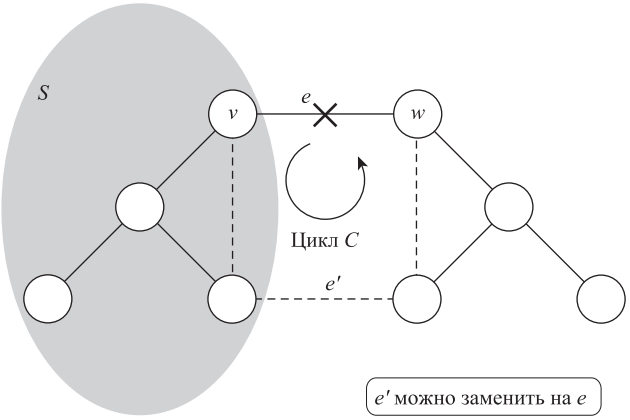


Рис. 4.11. Замена ребра e ребром e' в остовном дереве T в соответствии с доказательством (4.20)

Оптимальность алгоритма обратного удаления

Располагая свойством цикла (4.20), можно легко доказать, что алгоритм обратного удаления строит минимальное остовное дерево. Базовая идея аналогична доказательствам оптимальности двух предыдущих алгоритмов: алгоритм обратного удаления добавляет ребро только в том случае, если это оправданно согласно (4.20).

(4.21) Алгоритм обратного удаления строит минимальное остовное дерево графа G.

Доказательство. Рассмотрим любое ребро $e = (v, w)$, удаляемое алгоритмом обратного удаления. На момент удаления e находится в цикле C; и поскольку это первое ребро, обнаруженное алгоритмом в порядке убывания стоимости ребер, оно должно быть ребром с максимальной стоимостью в C. Следовательно, согласно (4.20), ребро e не принадлежит никакому минимальному остовному дереву.

Итак, если показать, что результат (V, T) алгоритма обратного удаления является остовным деревом графа G, дело будет сделано. Очевидно, граф (V, T) является связным, потому что алгоритм не станет удалять ребро, если это приведет к потере связности графа. Действуя от обратного, предположим, что (V, T) содержит цикл C. Рассмотрим ребро e с максимальной стоимостью в C, которое будет первым обнаружено этим алгоритмом. Это ребро должно быть удалено, потому что его удаление не приведет к потере связности графа, но это противоречит поведению алгоритма обратного удаления. ■

Хотя эта тема далее рассматриваться не будет, сочетание свойства сечения (4.17) со свойством цикла (4.20) наводит на мысль, что здесь действует еще

более общая закономерность. Любой алгоритм, который строит остовное дерево посредством включения ребер, соответствующих свойству сечения, и удаления ребер, соответствующих свойству цикла (в совершенно произвольном порядке), в конечном итоге сформирует минимальное остовное дерево. Этот принцип позволяет разрабатывать естественные жадные алгоритмы для этой задачи, кроме трех уже рассмотренных, и объясняет, почему существует так много жадных алгоритмов, создающих для нее оптимальные решения.

Отказ от предположения о различной стоимости всех ребер

До настоящего момента предполагалось, что все стоимости ребер различны; такое предположение упростило анализ в ряде мест. Но предположим, имеется экземпляр задачи минимального остовного дерева, которого некоторые ребра имеют одинаковую стоимость, — как убедиться в том, что рассмотренные алгоритмы по-прежнему обеспечивают оптимальные решения?

Оказывается, существует простой способ: нужно взять граф и изменить все стоимости ребер на разные, очень малые величины, чтобы все они стали различными. Теперь любые две стоимости, которые различались изначально, будут находиться в том же относительном порядке, поскольку отклонения очень малы; а раз все наши алгоритмы базируются на простом сравнении стоимости ребер, отклонения позволят разрешить конфликты при сравнении ребер со стоимостями, которые до этого были одинаковыми.

Более того, можно утверждать, что любое минимальное остовное дерево T для нового, измененного экземпляра также должно являться минимальным остовным деревом для исходного экземпляра. Убедиться в этом несложно: если стоимость T больше стоимости некоторого дерева T^* в исходном экземпляре, то для достаточно малых отклонений изменение стоимости T не может быть достаточным, чтобы оно стало лучше T^* с новыми стоимостями. Таким образом, при выполнении любого из наших алгоритмов минимального остовного дерева с измененными стоимостями для сравнения ребер будет получено минимальное остовное дерево T, которое также является оптимальным для исходного экземпляра.

Реализация алгоритма Прима

Обсудим реализацию рассмотренных алгоритмов и попробуем получить хорошие оценки времени выполнения. Как вы увидите, алгоритмы Прима и Крускала с правильным выбором структур данных могут быть реализованы с временем выполнения $O(m \log n)$. В этом разделе показано, как это делается для алгоритма Прима, а обсуждение реализации алгоритма Крускала приводится в следующем разделе. Для алгоритма обратного удаления получить время выполнения, близкое к этому, непросто, поэтому этим алгоритмом мы заниматься не будем.

Хотя для алгоритма Прима доказательство правильности сильно отличалось от доказательства алгоритма Дейкстры для кратчайшего пути, реализации алгоритмов Прима и Дейкстры почти идентичны. По аналогии с алгоритмом Дейкстры для принятия решения о том, какой узел v добавить следующим в растущее множество S, следует хранить стоимости присоединения $a(v) = \min_{e=(u, v):u \in S} c_e$ для каждого узла $v \in V - S$. Как и в предыдущем случае, узлы хранятся в приоритетной очереди, использующей стоимости присоединения $a(v)$ в качестве ключей; узел выбирается операцией `ExtractMin`, а для обновления стоимости присоединения используются операции `ChangeKey`. Всего существует $n - 1$ итераций, на которых выполняется операция `ExtractMin`, а операция `ChangeKey` выполняется не более одного раза для каждого ребра. Следовательно:

(4.22) При использовании приоритетной очереди алгоритм Прима для графа с n узлами и m ребрами может быть реализован с выполнением за время $O(m)$ с добавлением времени n операций `ExtractMin` и m операций `ChangeKey`.

Как и в случае с алгоритмом Дейкстры, приоритетная очередь на базе кучи позволяет реализовать операции `ExtractMin` и `ChangeKey` за время $O(\log n)$, а следовательно, обеспечивает общее время выполнения $O(m \log n)$.

Расширения

Задача нахождения минимального остовного дерева появилась как конкретная формулировка более общей цели проектирования сети — поиска хорошего способа соединения множества точек с прокладкой ребер между ними. Минимальное остовное дерево оптимизирует конкретную цель: обеспечение связности при минимальной суммарной стоимости ребер. Однако возможны и другие цели, о которых тоже стоит упомянуть.

Например, при планировании сети могут быть важны расстояния между узлами в построенном остовном дереве; возможно, предпочтение будет отдаваться их сокращению даже в том случае, если придется больше заплатить за набор ребер. В этом случае появляются новые проблемы, так как легко строятся примеры, в которых минимальное остовное дерево не обеспечивает минимальных расстояний между узлами, что предполагает некоторое противоречие между двумя целями.

Также может возникнуть вопрос перегрузки ребер. Если ситуация требует маршрутизации трафика между парами узлов, может возникнуть задача поиска остовного дерева, в котором трафик по любому ребру не превышает заданный порог. И в этом случае легко находится пример, в котором в минимальном остовном дереве присутствуют ребра с большим уровнем трафика.

На более общем уровне будет разумно спросить: а является ли остовное дерево правильным решением для подобных задач планирования сетей? Дерево обладает тем свойством, что уничтожение любого ребра ведет к потере связности; это означает, что дерево не обеспечивает защиты от сбоев. Вместо минимальной стоимости можно выбрать критерий жизнеспособности — например, искать связную сеть

с минимальной стоимостью, которая останется связной после удаления одного любого ребра.

Задачи, к которым ведут все эти расширения, существенно превосходят задачу нахождения минимального остовного дерева по вычислительной сложности. Тем не менее ввиду их практической ценности велись исследования по поиску хороших эвристик в этих областях.

4.6. Реализация алгоритма Крускала: структура Union-Find

Одной из основных задач графов является поиск множества связных компонент. В главе 3 рассматривались алгоритмы с линейным временем, использующие поиск BFS и DFS для нахождения компонент связности графа.

В этом разделе будет рассмотрена ситуация, в которой граф развивается посредством добавления ребер. Другими словами, граф имеет фиксированный набор узлов, но со временем между некоторыми парами узлов появляются новые ребра. Требуется обеспечить хранение множества компонент связности такого графа в процессе эволюции. При добавлении в граф нового ребра было бы нежелательно заново вычислять компоненты связности. Вместо этого мы воспользуемся структурой данных, называемой *структурой* `Union-Find`; эта структура позволяет хранить информацию компонент с возможностью быстрого поиска и обновления.

Именно такая структура данных необходима для эффективной реализации алгоритма Крускала. При рассмотрении каждого ребра $e = (v, w)$ необходимо эффективно идентифицировать компоненты связности, содержащие v и w. Если эти компоненты различны, то пути между v и w не существует и ребро e следует включить; но если компоненты совпадают, то путь v-w существует на ранее включенных ребрах, поэтому ребро e опускается. При включении e структура данных также должна обеспечивать эффективное слияние компонент v и w в одну новую компоненту.

Задача

Структура данных `Union-Find` обеспечивает хранение информации о непересекающихся множествах (например, компонентах графа) в следующем смысле. Для заданного узла u операция `Find`(u) возвращает имя множества, содержащего u. При помощи этой операции можно проверить, принадлежат ли два узла u и v к одному множеству, просто проверяя условие `Find`(u) = `Find`(v). Структура данных также реализует операцию `Union`(A, B), которая берет два множества A и B и сливает их в одно множество.

Эти операции могут использоваться для хранения информации о компонентах эволюционирующего графа $G = (V, E)$ по мере добавления новых ребер. Множества представляют компоненты связности графа. Для узла u операция `Find`(u) возвраща-

ет имя компоненты, содержащей u. Чтобы добавить в граф ребро (u, v), мы сначала проверяем, принадлежат ли u и v одной компоненте связности (условие `Find`(u) = `Find`(v)). Если они не принадлежат одной компоненте, то операция `Union`(`Find`(u), `Find`(v)) используется для слияния двух компонент. Важно заметить, что структура `Union-Find` может использоваться только для управления компонентами графа при добавлении ребер: она не предназначена для обработки удаления ребер, которое может привести к «разделению» одной компоненты надвое.

Итак, структура `Union-Find` будет поддерживать три операции.

- `MakeUnionFind`(S) для множества S возвращает структуру `Union-Find`. Например, эта структура соответствует набору компонент связности графа без ребер. Мы постараемся реализовать `MakeUnionFind` за время $O(n)$, где $n = |S|$.
- Для элемента $u \in S$ операция `Find`(u) возвращает имя множества, содержащего u. Нашей целью будет реализация `Find`(u) за время $O(\log n)$. Некоторые реализации, которые будут упомянуты в тексте, позволяют выполнить эту операцию за время $O(1)$.
- Для двух множеств A и B операция `Union`(A, B) изменяет структуру данных, объединяя множества A и B в одно. Нашей целью будет реализация `Union` за время $O(\log n)$.

Стоит разобраться, что же подразумевается под именем множества — например, возвращаемого операцией `Find`. Имена множеств определяются достаточно гибко; они просто должны быть непротиворечивыми в том отношении, что `Find`(v) и `Find`(w) должны возвращать одно имя, если v и w принадлежат одному множеству, и разные имена в противном случае. В наших реализациях имя каждого множества будет определяться одним из содержащихся в нем элементов.

Простая структура данных для структуры Union-Find

Возможно, самым простым вариантом реализации структуры данных `Union-Find` является массив `Component`, в котором хранится имя множества, содержащего каждый элемент. Пусть S — множество, состоящее из n элементов $\{1, ..., n\}$. Мы создаем массив `Component` размера n, в каждом элементе которого `Component`[s] хранится имя множества, содержащего s. Чтобы реализовать операцию `MakeUnionFind`(S), мы создаем массив и инициализируем его элементы `Component`[s] = s для всех $s \in S$. Эта реализация упрощает `Find`(v): все сводится к простой выборке по ключу, занимающей время $O(1)$. Однако операция `Union`(A, B) для двух множеств A и B может занимать время до $O(n)$ из-за необходимости обновления значений `Component`[s] для всех элементов множеств A и B.

Чтобы улучшить эту границу, мы проведем несколько простых оптимизаций. Во-первых, будет полезно явно хранить список элементов каждого множества, чтобы для нахождения обновляемых элементов не пришлось просматривать весь массив. Кроме того, можно сэкономить немного времени, выбирая в качестве имени объединения имя одного из исходных множеств, скажем A: в этом случае

достаточно обновить значения Component[s] для $s \in B$, но не для любых $s \in A$. Конечно, при большом множестве B выигрыш от этой идеи будет невелик, поэтому мы добавляем еще одну оптимизацию: для большого множества B будет сохраняться его имя, а значения Component[s] будут изменяться для $s \in A$. В более общей формулировке хранится дополнительный массив size размера n, где size[A] содержит размер множества A, а при выполнении операции Union(A, B) для объединения используется имя большего множества. Такой подход сокращает количество элементов, для которых требуется обновить значение Component.

Даже с такими оптимизациями операция Union в худшем случае выполняется за время $O(n)$: это происходит при объединении двух больших множеств A и B, содержащих одинаковые доли всех элементов. Тем не менее подобные плохие случаи не могут встречаться очень часто, поскольку итоговое множество $A \cup B$ имеет еще больший размер. Как точнее сформулировать это утверждение? Вместо того чтобы ограничивать время выполнения одной операции Union для худшего случая, мы можем ограничить общее (или среднее) время выполнения серии из k операций Union.

(4.23) Имеется реализация структуры данных Union-Find на базе массива для некоторого множества S размера n; за объединением сохраняется имя большего множества. Операция Find выполняется за время $O(1)$, MakeUnionFind(S) — за время $O(n)$, а любая последовательность из k операций Union выполняется за время не более $O(k \log k)$.

Доказательство. Истинность утверждений относительно операций MakeUnionFind и Find очевидна. Теперь рассмотрим серию из k операций Union. Единственной частью операции Union, которая выполняется за время, большее $O(1)$, является обновление массива Component. Вместо того чтобы ограничивать время одной операции Union, мы ограничим общее время, потраченное на обновление Component[v] для элемента v в серии из k операций.

Вспомните, что структура данных изначально находится в состоянии, при котором все n элементов находятся в отдельных множествах. В одной операции Union может быть задействовано не более двух исходных одноэлементных множеств, поэтому после любой последовательности из k операций Union все, кроме максимум $2k$ элементов S, останутся неизменными. Теперь возьмем конкретный элемент v. Так как множество v участвует в серии операций Union, его размер увеличивается. Может оказаться, что одни объединения изменяют значение Component[v], а другие нет. Но по нашему соглашению объединение использует имя большего множества, поэтому при каждом обновлении Component[v] размер множества, содержащего v, как минимум удваивается Размер множества v начинается с 1, а его максимально возможное значение равно $2k$ (как было указано выше, все, кроме максимум $2k$ элементов, не будут затронуты операциями Union). Это означает, что Component[v] в этом процессе обновляется не более $\log_2(2k)$ раз. Кроме того, в любых операциях Union будут задействованы не более $2k$ элементов, поэтому мы получаем границу $O(k \log k)$ для времени, проведенного за обновлением значений Component в серии из k операций Union. ■

Эта граница среднего времени выполнения серии из k операций достаточно хороша для многих приложений, включая реализацию алгоритма Крускала, мы попробуем ее улучшить, и сократить время худшего случая. Это будет сделано за счет повышения времени, необходимого для операции `Find`, до $O(\log n)$.

Усовершенствованная структура данных Union-Find

В структуре данных альтернативной реализации используются указатели. Каждый узел $v \in S$ будет храниться в записи с указателем на имя множества, содержащего v. Как и прежде, мы будем использовать элементы множества S в качестве имен множеств, а каждое множество будет названо по имени одного из своих элементов. Для операции `MakeUnionFind`(S) запись каждого элемента $v \in S$ инициализируется указателем, который указывает на себя (или определяется как `null`); это означает, что v находится в собственном множестве.

Рассмотрим операцию `Union` для двух множеств A и B. Предположим, что имя, использованное для множества A, определяется узлом $v \in A$, тогда как множество B названо по имени узла $u \in B$. Идея заключается в том, чтобы объединенному множеству было присвоено имя u или v; допустим, в качестве имени выбирается v. Чтобы указать, что мы выполнили объединение двух множеств, а объединенному множеству присвоено имя v, мы просто обновляем указатель u так, чтобы он ссылался на v. Указатели на другие узлы множества B при этом не обновляются.

В результате для элементов $w \in B$, отличных от u, имя множества, которому они принадлежат, приходится вычислять переходом по цепочке указателей, которая сначала ведет к «старому имени» u, а потом по указателю от u — к «новому имени» v. Пример такого представления изображен на рис. 4.12. Например, два множества на рис. 4.12 могут представлять результат следующей серии операций `Union`: `Union`(w, u), `Union`(s, u), `Union`(t, v), `Union`(z, v), `Union`(i, x), `Union`(y, j), `Union`(x, j) и `Union`(u, v).

Структура данных с указателями реализует операцию `Union` за время $O(1)$: требуется лишь обновить один указатель. Но операция `Find` уже не выполняется за постоянное время, потому что для перехода к текущему имени приходится отслеживать цепочку указателей с «историей» старых имен множества. Сколько времени может занимать операция `Find`(u)? Количество необходимых шагов в точности равно количеству изменений имени множества, содержащего узел u, то есть количеству обновлений позиции в массиве `Component`[u] в нашем представлении в виде массива. Оно может достигать $O(n)$, если мы не проявим достаточной осторожности с выбором имен множеств. Для сокращения времени, необходимого для операции `Find`, мы воспользуемся уже знакомой оптимизацией: имя наибольшего множества сохраняется как имя объединения. Последовательность объединений, приведших к структуре на рис. 4.12, подчинялась этому правилу. Чтобы эффективно реализовать этот вариант, необходимо хранить с узлами дополнительное поле: размер соответствующего множества.

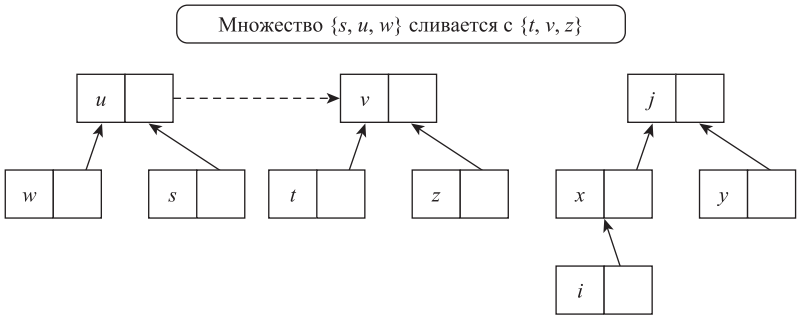


Рис. 4.12. Структура Union-Find с указателями. В данный момент структура содержит только два множества, названных по именам элементов v и j. Пунктирные стрелки от u к v представляют результат последней операции Union. Для обработки запроса Find мы перемещаемся по стрелкам, пока не доберемся до узла без исходящих стрелок. Например, для ответа на запрос Find(i) потребуется перейти по стрелке i к x, а затем от x к j

(4.24) Имеется описанная выше реализация структуры данных Union-Find с указателями для некоторого множества S размера n; за объединением сохраняется имя большего множества. Операция Union выполняется за время $O(1)$, MakeUnionFind(S) — за время $O(n)$, а операция Find — за время $O(\log n)$.

Доказательство. Истинность утверждений относительно операций Union и MakeUnionFind очевидна. Время выполнения Find(v) для узла v определяется количеством изменений имени множества, содержащего узел v, в процессе. По правилам за объединением сохраняется имя большего из объединяемых множеств, из чего следует, что при каждом изменении имени множества, содержащего узел v, размер этого множества по крайней мере удваивается. Поскольку множество, содержащее v, начинается с размера 1 и никогда не оказывается больше n, его размер может удваиваться не более $\log_2 n$ раз, а значит, имя может изменяться не более $\log_2 n$ раз. ■

Дальнейшие улучшения

Далее мы кратко рассмотрим естественную оптимизацию структуры данных Union-Find на базе указателей, приводящую к ускорению операций Find. Строго говоря, это улучшение не является необходимым для наших целей: для всех рассматриваемых применений структур данных Union-Find время $O(\log n)$ на операцию достаточно хорошо в том смысле, что дальнейшее улучшение времени операций не преобразуется в улучшение общего времени выполнения алгоритмов, которые их используют. (Операции Union-Find не единственное вычислительное «узкое место» во времени выполнения этих алгоритмов.)

Чтобы понять, как работает улучшенная версия этой структуры данных, для начала рассмотрим «плохой» случай времени выполнения для структуры данных Union-Find на базе указателей. Сначала мы строим структуру, в которой одна из операций Find выполняется приблизительно за время $\log n$. Для этого повторно

выполняются операции Union для множеств равных размеров. Предположим, v — узел, для которого операция Find(v) выполняется за время $\log n$. Теперь представьте, что Find(v) выполняется многократно, и каждое выполнение занимает время $\log n$. Необходимость переходить по одному пути из $\log n$ указателей каждый раз для получения имени множества, содержащего v, оказывается лишней: после первого обращения Find(v) имя x множества, содержащего v, уже «известно»; также известно, что все остальные узлы, входящие в путь от v к текущему имени, тоже содержатся в множестве x. Итак, в улучшенной реализации путь, отслеживаемый для каждой операции Find, «сжимается»: все указатели в пути переводятся на текущее имя множества. Потери информации при этом не происходит, а последующие операции Find будут выполняться быстрее. Структура данных Union-Find и результат выполнения Find(v) с использованием сжатия изображены на рис. 4.13.

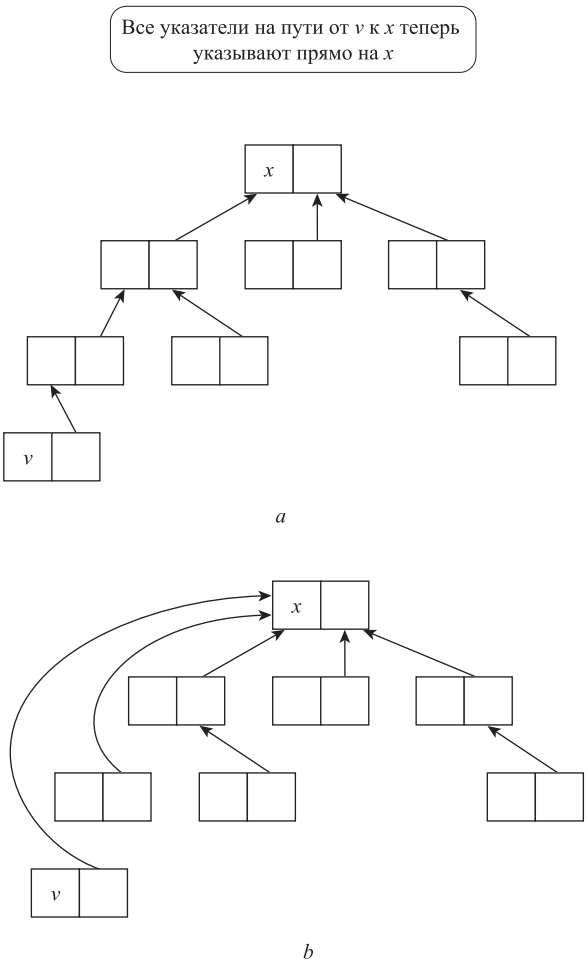


Рис. 4.13. (a) Пример структуры данных Union-Find; (b) результат выполнения операции Find(v) с использованием сжатия пути

Теперь рассмотрим время выполнения операций для такой реализации. Как и прежде, операция `Union` выполняется за время $O(1)$, а операция `MakeUnionFind`(S) занимает время $O(n)$ для создания структуры данных, представляющей множество размера n. Как изменилось время, необходимое для операции `Find`(v)? Некоторые операции `Find` по-прежнему занимают время до $\log n$; для некоторых операций `Find` это время увеличилось, потому что после нахождения имени x множества, содержащего v, приходится возвращаться по той же цепочке указателей от v к x и переводить каждый указатель прямо на x. Но эта дополнительная работа в худшем случае удваивает необходимое время, а следовательно, не изменяет того факта, что операция `Find` выполняется за максимальное время $O(\log n)$. Выигрыш от сжатия путей проявляется при последующих вызовах `Find`, а для его оценки можно воспользоваться тем же методом, который был применен в (4.23): ограничением общего времени серии из n операций `Find` (вместо худшего времени одной операции). И хотя мы не будем вдаваться в подробности, серия из n операций `Find` со сжатием требует времени, чрезвычайно близкого к линейной зависимости от n; фактическая верхняя граница имеет вид $O(n\alpha(n))$, где $\alpha(n)$ — очень медленно растущая функция n, называемая *обратной функцией Аккермана*. (В частности, $\alpha(n) \leq 4$ для любых значений n, которые могут встретиться на практике.)

Реализация алгоритма Крускала

А теперь воспользуемся структурой данных `Union-Find` для реализации алгоритма Крускала. Сначала нужно отсортировать ребра по стоимости; эта операция выполняется за время $O(m \log m)$. Так как каждая пара узлов соединяется максимум одним ребром, $m \leq n^2$, а следовательно, общее время выполнения также равно $O(m \log n)$.

После операции сортировки структура данных `Union-Find` используется для хранения информации о компонентах связности (V, T) при добавлении ребер. При рассмотрении каждого ребра $e = (v, w)$ мы вычисляем `Find`(u) и `Find`(v) и проверяем результаты на равенство, чтобы узнать, принадлежат ли v и w разным компонентам. Если алгоритм решает включить ребро e в дерево T, операция `Union`(`Find`(u), `Find`(v)) объединяет две компоненты.

В процессе выполнения алгоритма Крускала выполняются не более $2m$ операций `Find` и $n - 1$ операций `Union`. Используя либо (4.23) для реализации `Union-Find` на базе массива, либо (4.24) для реализации с указателями, можно заключить, что общее время реализации равно $O(m \log n)$. (Хотя возможны и более эффективные реализации структуры `Union-Find`, они не улучшат время выполнения алгоритма Крускала, который содержит неизбежный фактор $O(m \log n)$ из-за исходной сортировки ребер по стоимости.)

Подведем итог:

(4.25) Алгоритм Крускала может быть реализован для графа с n узлами и m ребрами со временем выполнения $O(m \log n)$.

4.7. Кластеризация

Практическая ценность задачи нахождения минимального остовного дерева была продемонстрирована на примере построения низкозатратной сети, связывающей ряд географических точек. Однако минимальные остовные деревья возникают во многих других ситуациях, порой внешне не имеющих ничего общего. Интересный пример использования минимальных остовных деревьев встречается в области кластеризации.

Задача

Под задачей *кластеризации* обычно понимается ситуация, в которой некую коллекцию объектов (допустим, набор фотографий, документов, микроорганизмов и т. д.) требуется разделить на несколько логически связных групп. В такой ситуации естественно начать с определения метрик сходства или расхождения каждой пары объектов. Одно из типичных решений основано на определении функции расстояния между объектами; предполагается, что объекты, находящиеся на большем расстоянии друг от друга, в меньшей степени похожи друг на друга. Для точек в реальном мире расстояние может быть связано с географическим расстоянием, но во многих случаях ему приписывается более абстрактный смысл. Например, расстояние между двумя биологическими видами может измеряться временны́м интервалом между их появлением в ходе эволюции; расстояние между двумя изображениями в видеопотоке может измеряться количеством пикселов, в которых интенсивность цвета превышает некоторый порог.

Для заданной функции расстояния между объектами процесс кластеризации пытается разбить их на группы так, чтобы объекты одной группы интуитивно воспринимались как «близкие», а объекты разных групп — как «далекие». Этот несколько туманный набор целей становится отправной точкой для множества технически различающихся методов, каждый из которых пытается формализовать общее представление о том, как должен выглядеть хороший набор групп.

Кластеризация по максимальному интервалу

Минимальные остовные деревья играют важную роль в одной из базовых формализаций, которая будет описана ниже. Допустим, имеется множество U из n объектов $p_1, p_2, ..., p_n$. Для каждой пары p_i и p_j определяется числовое расстояние $d(p_i, p_j)$. К функции расстояния предъявляются следующие требования: $d(p_i, p_i) = 0$; $d(p_i, p_j) > 0$ для разных p_i и p_j, а также $d(p_i, p_j) = d(p_j, p_i)$ (симметричность).

Предположим, объекты из U требуется разделить на k групп для заданного параметра k. Термином «k-кластеризация U» обозначается разбиение U на k непустых множеств $C_1, C_2, ..., C_k$. «Интервалом k-кластеризации» называется минимальное расстояние между любой парой точек, находящихся в разных кластерах. С учетом того, что точки разных кластеров должны находиться далеко друг от друга, есте-

ственной целью является нахождение k-кластеризации с максимально возможным интервалом.

Мы приходим к следующему вопросу. Количество разных k-кластеризаций множества U вычисляется по экспоненциальной формуле; как эффективно найти кластеризацию с максимальным интервалом?

Разработка алгоритма

Чтобы найти кластеризацию с максимальным интервалом, мы рассмотрим процедуру расширения графа с множеством вершин U. Компоненты связности соответствуют кластерам, и мы постараемся как можно быстрее объединить близлежащие точки в один кластер (чтобы они не оказались в разных кластерах, находящихся поблизости друг от друга). Начнем с создания ребра между ближайшей парой точек; затем создается ребро между парой точек со следующей ближайшей парой и т. д. Далее мы продолжаем добавлять ребра между парами точек в порядке увеличения расстояния $d(p_i, p_j)$.

Таким образом граф H с вершинами U наращивает ребро за ребром, при этом компоненты связности соответствуют кластерам. Учтите, что нас интересуют только компоненты связности графа H, а не полное множество ребер; если при добавлении ребра (p_i, p_j) выяснится, что p_i и p_j уже принадлежат одному кластеру, ребро добавляться не будет — это не нужно, потому что ребро не изменит множество компонент. При таком подходе в процессе расширения графа никогда не образуется цикл, так что H будет объединением деревьев. Добавление ребра, концы которого принадлежат двум разным компонентам, фактически означает слияние двух соответствующих кластеров. В литературе по кластеризации подобный итеративный процесс слияния кластеров называется *методом одиночной связи* — частным случаем иерархической агломератной кластеризации. (Под «агломератностью» здесь понимается процесс объединения кластеров, а под «одиночной связью» — то, что для объединения кластеров достаточно одной связи.) На рис. 4.14 изображен пример для $k = 3$ кластеров с разбиением точек на естественные группы.

Какое отношение все это имеет к минимальным остовным деревьям? Очень простое: хотя процедура расширения графа базировалась на идее слияния кластеров, она в точности соответствует алгоритму минимального остовного дерева Крускала. Делается в точности то, что алгоритм Крускала сделал бы для графа G, если бы между каждой парой узлов (p_i, p_j) существовало ребро стоимостью $d(p_i, p_j)$. Единственное отличие заключается в том, что мы стремимся выполнить k-кластеризацию, поэтому процедура останавливается при получении k компонент связности.

Другими словами, выполняется алгоритм Крускала, но он останавливается перед добавлением последних $k - 1$ ребер. Происходящее эквивалентно построению полного минимального остовного дерева T (в том виде, в каком оно было бы построено алгоритмом Крускала), удалению $k - 1$ ребер с наибольшей стоимостью (которое наш алгоритм вообще не добавлял) и определению k-кластеризации по

полученным компонентам связности C_1, C_2, ..., C_k. Итак, итеративное слияние кластеров эквивалентно вычислению минимального остовного дерева с удалением самых «дорогостоящих» ребер.

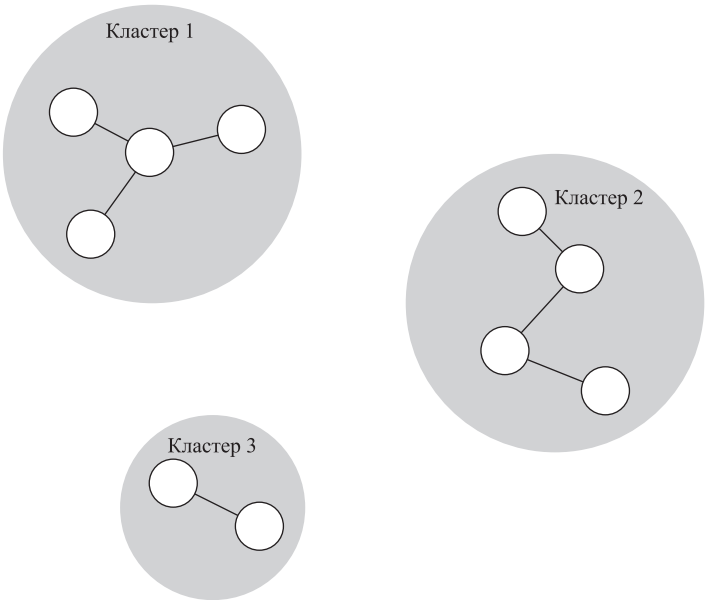


Рис. 4.14. Пример кластеризации методом одиночной связи с $k = 3$ кластерами. Кластеры образуются добавлением ребер между точками в порядке увеличения расстояния

Анализ алгоритма

Удалось ли нам достичь своей цели — создать кластеры, разделенные максимально возможным интервалом? Следующее утверждение доказывает этот факт.

(4.26) Компоненты C_1, C_2, ..., C_k, образованные удалением $k - 1$ ребер минимального остовного дерева T с максимальной стоимостью, образуют k-кластеризацию с максимальным интервалом.

Доказательство. Пусть C — кластеризация C_1, C_2, ..., C_k. Интервал C в точности равен длине d^* $(k-1)$-го ребра с максимальной стоимостью в минимальном остовном дереве; это длина ребра, которое алгоритм Крускала добавил бы на следующем шаге (в тот момент, когда мы его остановили).

Возьмем другую k-кластеризацию C', которая разбивает U на непустые множества C'_1, C'_2, ..., C'_k. Требуется показать, что интервал C' не превышает d^*.

Так как две кластеризации C и C' не совпадают, из этого следует, что один из кластеров Cr не является подмножеством ни одного из k множеств C'_s в C'. Следовательно, должны существовать точки $p_i, p_j \in C_r$, принадлежащие разным кластерам в C', — допустим, $p_i \in C'_s$ и $p_j \in C'_t \neq C'_s$.

Теперь взгляните на рис. 4.15. Так как p_i и p_j принадлежат одной компоненте C_r, алгоритм Крускала должен был добавить все ребра p_i-p_j пути P перед его остановкой. В частности, это означает, что длина каждого ребра P не превышает d^*. Мы знаем, что $p_i \in C'_s$, но $p_j \notin C'_s$; пусть p' — первый узел P, который не принадлежит C'_s, а p — узел из P, непосредственно предшествующий p'. Только что было замечено, что $d(p,p') \le d^*$, так как ребро (p, p') было добавлено алгоритмом Крускала. Но p и p' принадлежат разным множествам кластеризации C', а значит, интервал C' не превышает $d(p, p') \le d^*$, что и завершает доказательство. ■

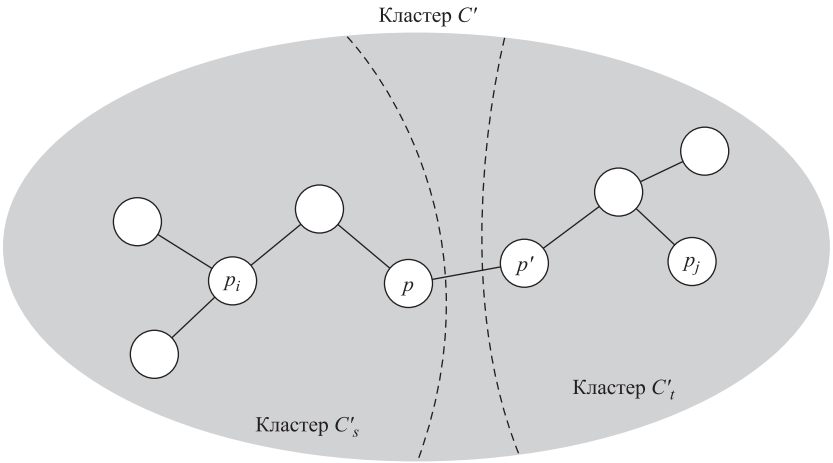


Рис. 4.15. Иллюстрация доказательства (4.26), показывающая, что интервал никакой другой кластеризации не может быть больше интервала кластеризации, найденной алгоритмом одиночной связи

4.8. Коды Хаффмана и сжатие данных

В задачах нахождения кратчайшего пути и минимального остовного дерева было показано, как жадные алгоритмы позволяют ограничиться определенными частями решения (ребрами графа в этих случаях), основываясь исключительно на относительно «недальновидных» соображениях. А сейчас будет рассмотрена задача, в которой «ограничение» понимается еще свободнее: по сути, жадное правило используется для сокращения размера экземпляра задачи, чтобы эквивалентную меньшую задачу можно было решить посредством рекурсии. Будет доказано, что такая жадная операция «безопасна» в том смысле, что решение меньшего экземпляра приводит к оптимальному решению исходной задачи, однако глобальные последствия исходного жадного решения проявляются только при завершении полной рекурсии.

Сама задача относится к одной из базовых тем из области *сжатия данных* — области, формирующей технологическую основу цифровой передачи данных.

Задача

Компьютеры в конечном счете работают с последовательностями битов (то есть последовательностями, состоящими только из 0 и 1), поэтому должна существовать некая схема кодирования, которая берет текст, написанный на более полном алфавите (например, алфавите одного из мировых языков), и преобразует его в длинную цепочку битов.

Самое простое решение — использовать фиксированное число битов для каждого символа в алфавите, а затем просто объединить цепочки битов для каждого символа. Рассмотрим простой пример: допустим, мы хотим закодировать 26 букв английского языка, а также пробел (для разделения слов) и пять знаков препинания: запятую, точку, вопросительный знак, восклицательный знак и апостроф. Таким образом, закодировать нужно 32 символа. Из b битов можно сформировать 2^b разных последовательностей, поэтому при использовании 5 битов на символ можно закодировать $2^5 = 32$ символа — вполне достаточно для наших целей. Например, цепочка битов 00000 будет представлять букву «a», цепочка 00001 — букву «b», и т. д. вплоть до цепочки 11111, которая представляет апостроф. Следует учитывать, что соответствие битовых цепочек и символов выбирается произвольно; суть в том, что пяти битов на символ достаточно. Такие схемы кодирования, как ASCII, работают именно так, не считая того, что они используют больше количество битов на символ для поддержки наборов с большим количеством символов, включая буквы верхнего регистра, скобки и все специальные символы, которые вы видите на клавиатуре пишущей машинки или компьютера.

Вернемся к простейшему примеру с 32 символами. Какие еще требования могут предъявляться к схеме кодирования? Ограничиться 4 битами на символ не удастся, потому что $2^4 = 16$ — недостаточно для имеющегося количества символов. С другой стороны, неочевидно, что в больших объемах текста код одного символа в среднем должен состоять из 5 битов. Если задуматься, буквы большинства алфавитов используются с разной частотой. Скажем, в английском языке буквы *e*, *t*, *a*, *o*, *i* и *n* используются намного чаще букв *q*, *j*, *x* и *z* (более чем на порядок). Получается, что кодировать их одинаковым количеством битов крайне расточительно; разумнее было бы использовать малое количество битов для часто встречающихся букв, а большее количество битов для редких букв и надеяться на то, что в длинных блоках типичного текста для представления одной буквы в среднем будет использоваться менее 5 бит.

Сокращение среднего количества битов на символ является одной из фундаментальных задач в области *сжатия данных*. Когда большие файлы нужно передать по коммуникационной сети или сохранить на жестком диске, важно найти для них как можно более компактное представление — при условии, что сторона, которая будет читать файл, сможет правильно восстановить его. Значительные объемы исследований были посвящены разработке алгоритмов сжатия, которые получают файлы и сокращают объем занимаемого ими пространства за счет применения эффективных схем кодирования.

А теперь мы рассмотрим одну из фундаментальных формулировок этой задачи, через которую мы постепенно придем к вопросу о том, как оптимальным образом

использовать факт неравномерности относительных частот букв. В каком-то смысле такое оптимальное решение становится хорошим ответом для задачи сжатия данных: из неравномерности частот извлекается вся возможная польза. В завершающей части этого раздела речь пойдет о том, как повысить эффективность сжатия за счет использования других возможностей помимо неравномерности частот.

Схемы кодирования с переменной длиной

До появления Интернета, цифровых компьютеров, радио и телефона существовал телеграф. По телеграфу данные передавались намного быстрее, чем в современных ему альтернативах доставки (железная дорога или лощади). Но телеграф позволял передавать по проводу только электрические импульсы, поэтому для передачи сообщения необходимо каким-то образом закодировать текст сообщения в виде последовательности импульсов.

Пионер телеграфной передачи данных Сэмюэл Морзе разработал код, в котором каждая буква соответствовала серии точек (короткие импульсы) и тире (длинные импульсы). В нашем контексте точки и тире можно заменить нулями и единицами, так что задача сводится к простому отображению символов на цепочки битов, как и в кодировке ASCII. Морзе понимал, что эффективность передачи данных можно повысить за счет кодирования более частых символов короткими строками, и выбрал именно такое решение. (За информацией об относительных частотах букв английского языка он обратился к местным типографиям.) В коде Морзе букве «*e*» соответствует 0 (точка), букве «*t*» — 1 (тире), букве «*a*» — 01 (точка-тире), и вообще более частым буквам соответствуют более короткие цепочки.

В коде Морзе для представления букв используются настолько короткие цепочки, что кодирование слов становится неоднозначным. Например, из того, что мы знаем о кодировке букв «*e*», «*t*» и «*a*», становится видно, что цепочка 0101 может соответствовать любой из последовательностей «*eta*», «*aa*», «*etet*» или «*aet*». (Также существуют другие возможные варианты с другими буквами.) Для разрешения этой неоднозначности при передаче кода Морзе буквы разделяются короткими паузами (так что текст «*aa*» будет кодироваться последовательностью точка-тире-пауза-точка-тире-пауза). Это решение — очень короткие битовые цепочки с паузами — выглядит разумно, но оно означает, что буквы кодируются не 0 и 1; фактически применяется алфавит из трех букв — 0, 1 и «пауза». Следовательно, если потребуется что-то закодировать только битами 0 и 1, потребуется дополнительное кодирование, которое ставит в соответствие паузе последовательность битов.

Префиксные коды

Проблема неоднозначности в коде Морзе возникает из-за существования пар букв, у которых битовая цепочка, кодирующая одну букву, является префиксом в битовой цепочке, кодирующей другую букву. Чтобы устранить эту проблему (а следовательно, получить схему кодирования с четко определенной интерпретацией каждой последовательности битов), достаточно отображать буквы на цепочки битов так, чтобы ни одна кодовая последовательность не была префиксом другой кодовой последовательности.

Префиксным кодом для множества букв S называется функция γ, которая отображает каждую букву $x \in S$ на некую последовательность нулей и единиц так, что для разных $x,y \in S$ последовательность $\gamma(x)$ не является префиксом последовательности $\gamma(y)$.

Предположим, имеется текст, состоящий из последовательности букв $x_1, x_2, x_3, ..., x_n$. Чтобы преобразовать его в последовательность битов, достаточно закодировать каждую букву последовательностью битов с использованием γ, а затем объединить все эти последовательности: $\gamma(x_1)\gamma(x_2)...\gamma(x_n)$. Если это сообщение передается получателю, которому известна функция γ, то получатель сможет восстановить текст по следующей схеме:

- Просканировать последовательность битов слева направо.
- Как только накопленных битов будет достаточно для кодирования некоторой буквы, вывести ее как первую букву текста. Ошибки исключены, потому что никакой более короткий или длинный префикс последовательности битов не может кодировать никакую другую букву.
- Удалить соответствующую последовательность битов от начала сообщения и повторить сначала.

В этом случае получатель сможет получить правильную последовательность букв, не прибегая к искусственным мерам вроде пауз для разделения букв. Предположим, мы пытаемся закодировать набор из пяти букв $S=\{a, b, c, d, e\}$. Кодировка γ_1, заданная следующим образом:

$$\gamma_1(a) = 11$$

$$\gamma_1(b) = 01$$

$$\gamma_1(c) = 001$$

$$\gamma_1(d) = 10$$

$$\gamma_1(e) = 000$$

является префиксным кодом, поскольку ни один код не является префиксом другого. Например, строка «*cecab*» будет закодирована в виде 0010000011101. Получатель, зная γ_1, начнет читать сообщение слева направо.

Ни 0, ни 00 не являются кодом буквы, но для 001 такое соответствие обнаруживается, и получатель приходит к выводу, что первой буквой является «*c*». Такое решение безопасно, поскольку ни одна более длинная последовательность битов, начинающаяся с 001, не может обозначать другую букву. Затем получатель перебирает остаток сообщения, 0000011101; он приходит к выводу, что второй буквой является «*e*» (000), и т. д.

Оптимальные префиксные коды

Все это делается из-за того, что некоторые буквы встречаются чаще других и мы хотим воспользоваться тем обстоятельством, что более частые буквы можно коди-

ровать более короткими последовательностями. Чтобы точно сформулировать эту цель, сейчас мы введем вспомогательные обозначения для выражения частот букв.

Предположим, для каждой буквы $x \in S$ существует относительная частота f_x, представляющая долю в тексте букв, равных x. Другими словами, если текст состоит из n букв, то nf_x из этих букв равны x. Сумма всех относительных частот равна 1, то есть $\sum_{x \in S} f_x = 1$.

Если префиксный код γ используется для кодирования заданного текста, то какой будет общая длина кодирования? Она будет равна сумме (по всем буквам $x \in S$) количества вхождений x, умноженной на длину битовой цепочки $\gamma(x)$, используемой для кодирования x. Если $|\gamma(x)|$ обозначает длину $\gamma(x)$, этот факт можно записать в виде

$$\text{Длина кода} = \sum_{x \in S} nf_x \left|\gamma(x)\right| = n \sum_{x \in S} f_x \left|\gamma(x)\right|.$$

Исключая коэффициент n из последнего выражения, получаем $\sum_{x \in S} f_x \left|\gamma(x)\right|$ — среднее количество битов, необходимых для представления буквы. В дальнейшем эта величина обозначается ABL(γ) (Average Bits per Letter).

В продолжение этого примера допустим, что имеется текст с буквами $S = \{a, b, c, d, e\}$ и следующим набором относительных частот:

$$f_a = 0{,}32, f_b = 0{,}25, f_c = 0{,}20, f_d = 0{,}18, f_e = 0{,}05.$$

Тогда среднее количество битов на букву для префиксного кода γ_1, определенного ранее, составит

$$0{,}32 \cdot 2 + 0{,}25 \cdot 2 + 0{,}20 \cdot 3 + 0{,}18 \cdot 2 + 0{,}05 \cdot 3 = 2{,}25.$$

Интересно сравнить эту величину со средним количеством битов на букву при использовании кодировки с фиксированной длиной. (Заметьте, что кодировка с фиксированной длиной является префиксным кодом: если все буквы кодируются последовательностями одинаковой длины, то никакой код не может быть префиксом любого другого.) Очевидно, в множестве S из пяти букв для кодировки с фиксированной длиной понадобится три бита на букву, так как два бита позволят закодировать только четыре буквы. Следовательно, кодировка γ_1 сокращает количество битов на букву с 3 до 2,25 — экономия составляет 25 %.

На самом деле γ_1 не лучшее, что можно сделать в этом примере. Рассмотрим префиксный код γ_2, определяемый следующим образом:

$$\gamma_2(a) = 11$$

$$\gamma_2(b) = 10$$

$$\gamma_2(c) = 01$$

$$\gamma_2(d) = 001$$

$$\gamma_2(e) = 000$$

Среднее количество битов на букву для γ_2 составит

$$0{,}32 \cdot 2 + 0{,}25 \cdot 2 + 0{,}20 \cdot 2 + 0{,}18 \cdot 3 + 0{,}05 \cdot 3 = 2{,}23.$$

Теперь можно сформулировать естественный вопрос. Для заданного алфавита и множества относительных частот букв нам хотелось бы получить префиксный код, обладающий наибольшей эффективностью, а именно префиксный код, минимизирующий среднее количество битов на букву $\mathrm{ABL}(\gamma) = \sum_{x \in S} f_x |\gamma(x)|$. Такой префиксный код называется *оптимальным*.

Разработка алгоритма

Пространство поиска в этой задаче весьма сложно: оно включает все возможные способы отображения букв на цепочки битов с учетом определяющего свойства префиксных кодов. Для алфавитов с очень малым количеством букв возможен перебор этого пространства методом «грубой силы», но такой подход быстро становится неприемлемым.

А теперь будет описан жадный метод построения оптимального префиксного кода с высокой эффективностью. Полезно начать с создания способа представления префиксных кодов на базе дерева, который репрезентирует их структуру более четко, чем списки значений из предыдущих примеров.

Представление префиксных кодов в форме бинарных деревьев

Дерево T, в котором каждый узел, не являющийся листовым, имеет не более двух дочерних узлов, называется *бинарным деревом*. Допустим, количество листовых узлов равно размеру алфавита S, а каждый лист помечен одной из букв S.

Такое бинарное дерево T естественно описывает префиксный код. Для каждой буквы $x \in S$ отслеживается путь от корня до листа с меткой x; каждый раз, когда путь переходит от узла к его левому дочернему узлу, в результирующую цепочку битов записывается 0, а при переходе к правому узлу — 1. Полученная цепочка битов рассматривается как кодировка x.

Отметим следующий факт:

(4.27) Кодировка алфавита S, построенная на основе T, представляет собой префиксный код.

Доказательство. Чтобы кодировка x была префиксом кодировки y, путь от корня к узлу x должен быть префиксом пути от корня к узлу y. Но для этого узел x должен лежать на пути от корня к y, а это невозможно, если узел x является листовым. ■

Отношения между бинарными деревьями и префиксными кодами также работают и в обратном направлении: для префиксного кода γ можно рекурсивно построить бинарное дерево. Начните с корня; все буквы $x \in S$, кодировка которых начинается с 0, будут представлены листовыми узлами в левом поддереве, а все

буквы $y \in S$, кодировка которых начинается с 1, будут представлены листьями в правом поддереве. Далее эти два поддерева строятся рекурсивно по указанному правилу.

Например, дерево на рис. 4.16, *a* соответствует следующему префиксному коду γ_0:

$$\gamma_0(a) = 1$$

$$\gamma_0(b) = 011$$

$$\gamma_0(c) = 010$$

$$\gamma_0(d) = 001$$

$$\gamma_0(e) = 000$$

Чтобы убедиться в этом, заметьте, что к листу с меткой *a* можно перейти, просто выбрав правую ветвь от корневого узла (кодировка 1); для перехода к узлу *e* следует трижды выбрать левую ветвь от корневого узла; такие же объяснения действуют для *b*, *c* и *d*. Аналогичные рассуждения показывают, что дерево на рис. 4.16, *b* соответствует префиксному коду γ_1, а дерево на рис. 4.16, *c* соответствует префиксному коду γ_2 (оба кода определялись выше). Также следует заметить, что бинарные деревья двух префиксных кодов γ_1 и γ_2 имеют идентичную структуру; различаются только метки листьев. С другой стороны, дерево γ_0 имеет отличающуюся структуру.

Таким образом, поиск оптимального префиксного кода может рассматриваться как нахождение бинарного дерева T в сочетании с пометкой узлов T, минимизирующей среднее количество битов на букву. Кроме того, эта средняя величина имеет естественную интерпретацию в контексте структуры T: длина кодировки буквы $x \in S$ попросту равна длине пути от корневого узла до листа с меткой x. Длина пути будет называться *глубиной* листа, а глубина листа v в T будет обозначаться $\text{depth}_T(v)$. (Для удобства записи мы будем опускать нижний индекс T там, где он очевиден из контекста, и часто будем использовать букву $x \in S$ для обозначения помечаемого ею узла.) Итак, мы пытаемся найти дерево с метками узлов, обладающее минимальным средним значением глубин всех листьев, взвешенных по частотам букв, которыми помечены листья: $\sum_{x \in S} f_x \cdot \text{depth}_T(x)$. Для обозначения этой метрики будет использоваться запись $\text{ABL}(T)$.

Первым шагом рассмотрения алгоритмов для этой задачи станет констатация простого факта, относящегося к оптимальному дереву. Для этого нам понадобится вспомогательное определение: бинарное дерево называется *полным*, если каждый узел, который не является листовым, имеет два дочерних узла. (Другими словами, не существует узлов, имеющих ровно один дочерний узел.) Обратите внимание: все три бинарных дерева на рис. 4.16 являются полными.

(4.28) Бинарное дерево, соответствующее оптимальному префиксному коду, является полным.

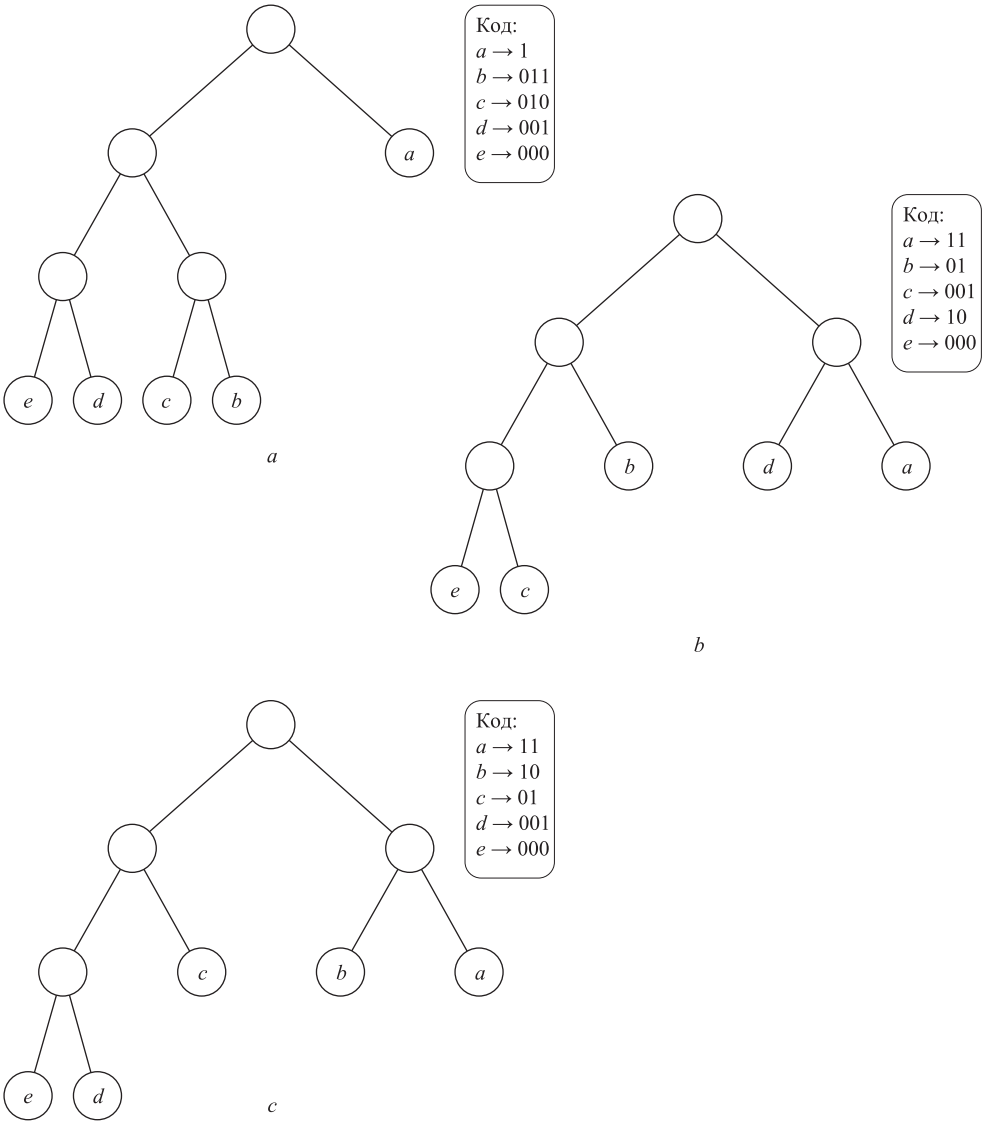


Рис. 4.16. Три разных префиксных кода для алфавита $S = \{a, b, c, d, e\}$

Доказательство. Это утверждение легко доказывается методом замены. Пусть T — бинарное дерево, соответствующее оптимальному префиксному коду; допустим, оно содержит узел u, имеющий ровно один дочерний узел v. Преобразуем T в дерево T', заменив узел u узлом v.

Чтобы анализ был точным, необходимо различать два случая. Если u — корень дерева, узел u просто удаляется, а v используется в качестве корневого узла. Если u корнем не является, пусть w является родителем u в T. В этом случае u удаляется,

а v становится дочерним узлом w вместо u. Это изменение уменьшает количество битов, необходимых для кодирования любого листа в поддереве, корнем которого является u, и не влияет на другие листья. Итак, префиксный код, соответствующий T', имеет меньшее среднее количество битов на букву, чем префиксный код T, а это противоречит предположению об оптимальности T. ■

Первая попытка: нисходящий метод

На интуитивном уровне мы стремимся построить размеченное бинарное дерево, листья которого должны располагаться как можно ближе к корню. Именно такое строение обеспечит наименьшую среднюю глубину листа.

Естественный подход к решению этой задачи будет основан на построении дерева сверху вниз с как можно более плотной «упаковкой» листьев. Допустим, мы пытаемся разбить алфавит S на два множества S_1 и S_2 так, чтобы сумма частот букв в каждом наборе была равна в точности $\frac{1}{2}$. Если такое идеальное разбиение невозможно, то мы можем опробовать следующую разбивку, ближайшую к сбалансированности. Затем для S_1 и S_2 рекурсивно строятся префиксные коды, которые образуют два поддерева от корневого узла. (В контексте цепочек битов это означает, что кодировки для S_1 будут начинаться с 0, а кодировки для S_2 — с 1.)

Пока не совсем ясно, как конкретно определить это «почти сбалансированное» разбиение алфавита, однако более точная формулировка возможна. Такие схемы кодирования называются *кодами Шеннона–Фано* по именам Клода Шеннона и Роберта Фано — двух крупных ученых, работавших в области теории передачи информации (дисциплины, изучающей способы представления и кодирования цифровых данных). Такие префиксные коды неплохо работают на практике, но для наших текущих целей они создают своего рода «тупик»: никакая разновидность такой стратегии нисходящего разбиения не будет гарантированно создавать оптимальный префиксный код. Вернемся к примеру с алфавитом из пяти букв $S = \{a, b, c, d, e\}$ и частотами

$$f_a = 0{,}32,\ \ f_b = 0{,}25,\ \ f_c = 0{,}20,\ \ f_d = 0{,}18,\ \ f_e = 0{,}05.$$

Существует всего один способ разбиения этого алфавита на два множества с равными частотами: $\{a, d\}$ и $\{b, c, e\}$. В подмножестве $\{a, d\}$ каждая буква может кодироваться одним дополнительным битом. Для $\{b, c, e\}$ кодирование должно продолжаться рекурсивно, и снова существует уникальный способ разбиения множества на два подмножества с равной суммой частот. Полученный код соответствует кодировке γ_1, представленной размеченным деревом на рис. 4.16, *b*; как вы уже знаете, кодировка γ_1 менее эффективна, чем префиксная кодировка γ_2, соответствующая дереву на рис. 4.16, *c*.

Шеннон и Фано знали, что их метод не всегда порождает оптимальный префиксный код, но не знали, как вычислить оптимальный код без поиска методом «грубой силы». Эту задачу через несколько лет решил аспирант Дэвид Хаффман — он узнал о задаче на учебном курсе, который вел Фано.

Рассмотрим некоторые идеи, которые привели Хаффмана к обнаружению жадного алгоритма построения оптимальных префиксных кодов.

А если бы структура дерева оптимального префиксного кода была известна?

Существует метод, который часто помогает при поиске эффективного алгоритма: вы мысленно предполагаете, что у вас имеется частичная информация об оптимальном решении, и смотрите, как воспользоваться этой неполной информацией для поиска полного решения. (Как будет показано позднее, в главе 6, этот метод является одной из главных основ метода динамического программирования при разработке алгоритмов.)

В контексте текущей задачи будет полезно задаться вопросом: а если бы нам было известно бинарное дерево T^*, соответствующее оптимальному префиксному коду, но не разметка листьев? Осталось бы разобраться, какой буквой должен быть помечен каждый из листьев T^*, и код был бы найден. Насколько сложно это сделать?

На самом деле это достаточно просто, но сначала следует сформулировать базовый факт.

(4.29) Допустим, u и v — листовые узлы T^*, такие что $\text{depth}(u) < \text{depth}(v)$. Также предположим, что в разметке T^*, соответствующей оптимальному префиксному коду, лист u помечается $y \in S$, а лист v помечается $z \in S$. В этом случае $f_y \geq f_z$.

Доказательство. Следующее короткое доказательство основано на методе замены. Допустим, $f_y < f_z$; рассмотрим код, который будет получен, если поменять местами метки узлов u и v. В выражении для среднего количества битов на букву $\text{ABL}(T^*) \sum_{x \in S} f_x \text{depth}(x)$ замена приводит к следующему эффекту: множитель у f_y возрастает (с $\text{depth}(u)$ до $\text{depth}(v)$), а множитель у f_z уменьшается на ту же величину (с $\text{depth}(v)$ до $\text{depth}(u)$). Следовательно, сумма изменится на величину $(\text{depth}(v) - \text{depth}(u))(f_y - f_z)$. Если $f_y < f_z$, это изменение будет отрицательным числом, что противоречит предположению об оптимальности префиксного кода до замены. ■

Идея, заложенная в основу (4.29), продемонстрирована на рис. 4.16, *b*. Как легко показать, что код неоптимален? Достаточно показать, что его можно улучшить, поменяв местами метки *c* и *d*. Наличие низкочастотной буквы на строго меньшей глубине, чем у другой буквы с более высокой частотой, — именно тот признак, который исключается у оптимального решения согласно (4.29).

Из (4.29) следует интуитивно понятный оптимальный способ разметки готового дерева T^* (при условии, что кто-то нам его предоставил). Сначала берем все листья глубины 1 (если они есть) и размечаем их буквами с наибольшей частотой в любом порядке. Затем берем все листья глубины 2 (если они есть) и размечаем их буквами с наибольшей частотой в любом порядке. Процедура продолжается в порядке повышения глубины, с назначением букв в порядке уменьшения частоты. Здесь важно то, что это не может привести к субоптимальной разметке T^*,

поскольку к любой предположительно лучшей разметке может быть применена замена (4.29). Также принципиально то, что при назначении меток в блоке листьев одной глубины не важно, какая метка будет назначена тому или иному узлу. Поскольку глубины совпадают, соответствующие множители в выражении $\sum_{x \in S} f_x |\gamma(x)|$ тоже совпадают, а выбор присваивания между листьями одной глубины не влияет на среднее количество битов на букву.

Но как все это поможет нам? Структура оптимального дерева T^* нам неизвестна, а из-за экспоненциального количества возможных деревьев (от размера алфавита) перебрать все возможные деревья методом «грубой силы» не удастся.

Однако все рассуждения о T^* становятся очень полезными, если думать не о начале процесса разметки (с листьями минимальной глубины), а о его завершении с листьями максимальной глубины — теми, которым достаются буквы с наименьшей частотой. А конкретнее, рассмотрим лист v в дереве T^* с наибольшей возможной глубиной. Лист v имеет родителя u, но, согласно (4.28), T^* является полным бинарным деревом, поэтому у u есть другой дочерний узел — w. Узлы v и w будут называться *одноранговыми* (или *соседями*). Тогда получаем

(4.30) Узел w является листом T^*.

Доказательство. Если узел w не является листовым, где-то в поддереве ниже него должен быть узел w'. Но тогда глубина w' будет превышать глубину v, а это противоречит нашему предположению о том, что v является листом максимальной глубины в T^*. ■

Итак, v и w — одноранговые узлы, имеющие максимально возможную глубину в T^*. Следовательно, процесс разметки T^* в соответствии с (4.29) доберется до уровня, содержащего v и w, в последнюю очередь. Для этого уровня останутся буквы с наименьшей частотой. Так как порядок назначения этих букв листьям внутри уровня роли не играет, существует оптимальная разметка, при которой v и w достанутся две буквы с наименьшей частотой из всех возможных.

Подведем итог:

(4.31) Существует оптимальный префиксный код с соответствующим деревом T^*, в котором две буквы с наименьшей частотой назначаются листьям, которые являются одноранговыми в T^*.

Алгоритм построения оптимального префиксного кода

Предположим, y^* и z^* — две буквы S с наименьшими частотами (с произвольной разбивкой «ничьих»). Утверждение (4.31) важно, так как оно сообщает важную информацию о местонахождении y^* и z^* в оптимальном решении; оно говорит, что при анализе решениях их можно «держать вместе», потому что мы знаем, что они станут одноранговыми узлами под общим родителем. Получается, что этот общий родитель становится «метабуквой», частота которой равна сумме частот y^* и z^*.

Отсюда напрямую следует идея алгоритма: y^* и z^* заменяются этой метабуквой с получением алфавита, уменьшенного на одну букву. Затем рекурсивно вычисляется префиксный код для сокращенного алфавита, метабуква снова «раскрыва-

ется» на y^* и z^* с получением префиксного кода для S. Эта рекурсивная стратегия изображена на рис. 4.17.

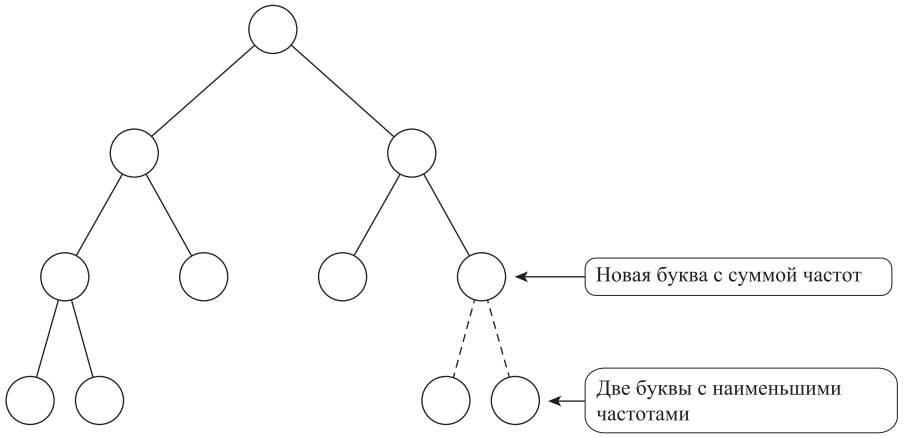


Рис. 4.17. Существует оптимальное решение, в котором две буквы с наименьшими частотами используются для пометки одноранговых листьев; если удалить их и пометить родительский узел новой буквой с суммарной частотой, это порождает экземпляр задачи с сокращенным алфавитом

Ниже приводится формальное описание алгоритма.

```
Построение префиксного кода для алфавита S с заданными частотами:
   Если S состоит из двух букв
      Закодировать одну букву 0, а другую 1
   Иначе
      Пусть y* и z* – две буквы с наименьшими частотами
      Образовать новый алфавит S′: удалить y* и z* и
         заменить их новой буквой ω с частотой f_y* + f_z*
      Рекурсивно сконструировать префиксный код γ′ для S′, с деревом T′
      Определить префиксный код для S следующим образом:
         Начать с T′
         Взять лист с меткой ω и добавить под ним два дочерних узла
           с метками y* и z*
   Конец Если
```

Эта процедура будет называться *алгоритмом Хаффмана*, а префиксный код, создаваемый ею для заданного алфавита, соответственно, будет называться *кодом Хаффмана*. В общем случае очевидно, что выполнение алгоритма всегда завершается, потому что он рекурсивно вызывает себя для алфавита, уменьшенного на одну букву. Более того, с (4.31) нетрудно доказать, что алгоритм действительно создает оптимальный префиксный код. Но перед этим стоит сделать пару замечаний по поводу алгоритма.

Для начала рассмотрим поведение алгоритма на примере с $S = \{a, b, c, d, e\}$ и частотами

$$f_a = 0{,}32,\ \ f_b = 0{,}25,\ \ f_c = 0{,}20,\ \ f_d = 0{,}18,\ \ f_e = 0{,}05.$$

Алгоритм объединяет d и e в одну букву — обозначим ее (de) — с частотой 0,18 + 0,05 = 0,23. Теперь мы имеем экземпляр задачи с четырехбуквенным алфавитом $S' = \{a, b, c, (de)\}$. Двумя буквами S' с наименьшей частотой являются буквы c и (de), поэтому на следующем шаге они объединяются в одну букву (cde) с частотой 0,20 + 0,23 = 0,43; образуется трехбуквенный алфавит $\{a, b, (cde)\}$. Далее объединяются буквы a и b с получением алфавита из двух букв, и в этой точке активизируется база рекурсии. Развернув результат по цепочке рекурсивных вызовов, мы получим дерево, изображенное на рис. 4.16, *c*.

Интересно, как жадное правило, заложенное в основу алгоритма Хаффмана (слияние букв с двумя наименьшими частотами), укладывается в структуру алгоритма в целом. Фактически на момент слияния двух букв мы не знаем точно, какое место они займут в общем коде. Вместо этого мы просто считаем их дочерними узлами одного родителя, и этого хватает для построения новой эквивалентной задачи, сокращенной на одну букву.

Кроме того, алгоритм образует естественный контраст с более ранним решением, приводящим к субоптимальным кодам Шеннона–Фано. Этот метод базировался на нисходящей стратегии, которая прежде всего обращала внимание на разбиение верхнего уровня в бинарном дереве, а именно на два подузла, находящиеся непосредственно под корнем. С другой стороны, алгоритм Хаффмана использует восходящий метод: он находит листья, представляющие две буквы с самой низкой частотой, а затем продолжает работу по рекурсивному принципу.

Анализ алгоритма

Оптимальность

Начнем с доказательства оптимальности алгоритма Хаффмана. Поскольку алгоритм работает рекурсивно, вызывая самого себя для последовательно сокращаемых алфавитов, будет естественно попытаться установить его оптимальность посредством индукции по размеру алфавита. Очевидно, он оптимален для всех двухбуквенных алфавитов (так как использует всего один бит на букву). Итак, предположим посредством индукции, что он оптимален для всех алфавитов размера $k - 1$, и рассмотрим входной экземпляр, состоящий из алфавита S размера k.

Напомним, как работает алгоритм для этого экземпляра: он объединяет две буквы с наименьшей частотой $y^*, z^* \in S$ в одну букву ω, рекурсивно вызывает себя для сокращенного алфавита S' (в котором y^* и z^* заменены ω) и посредством индукции строит оптимальный префиксный код для S', представленный размеченным бинарным деревом T'. Полученный код расширяется в дерево T для S, для чего листья y^* и z^* присоединяются как дочерние узлы к узлу T' с меткой ω.

Между метриками $\mathrm{ABL}(T)$ и $\mathrm{ABL}(T')$ существует тесная связь (первая — среднее количество битов, используемых для кодирования букв S, а вторая — среднее количество битов, используемых для кодирования букв S').

(4.32) $\mathrm{ABL}(T') = \mathrm{ABL}(T) - f_\omega$.

Доказательство. Глубина каждой буквы x, за исключением y^*, z^*, одинакова в T и T'. Кроме того, каждая из глубин y^* и z^* в T на 1 больше глубины ω в T'. Используя это отношение, а также тот факт, что $f_\omega = f_{y^*} + f_{z^*}$, получаем

$$\begin{aligned}
\mathrm{ABL}(T) &= \sum_{x \in S} f_x \cdot \mathrm{depth}_T(x) \\
&= f_{y^*} \cdot \mathrm{depth}_T(y^*) + f_{z^*} \cdot \mathrm{depth}_T(z^*) + \sum_{x \neq y^*, z^*} f_x \cdot \mathrm{depth}_T(x) \\
&= \left(f_{y^*} + f_{z^*}\right) \cdot \left(1 + \mathrm{depth}_{T'}(\omega)\right) + \sum_{x \neq y^*, z^*} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega \cdot \left(1 + \mathrm{depth}_{T'}(\omega)\right) + \sum_{x \neq y^*, z^*} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + f_\omega \cdot \mathrm{depth}_{T'}(\omega) + \sum_{x \neq y^*, z^*} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + \sum_{x \in S'} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + ALT(T'). \blacksquare
\end{aligned}$$

При помощи этой формулы теперь можно переходить к доказательству оптимальности.

(4.33) Код Хаффмана для заданного алфавита обеспечивает минимальное среднее количество битов на букву в любом префиксном коде.

Доказательство. Действуя от обратного, предположим, что построенное жадным алгоритмом дерево T не является оптимальным. Это означает, что существует размеченное бинарное дерево Z, такое что $\mathrm{ABL}(Z) < \mathrm{ABL}(T)$; а согласно (4.31), существует такое дерево Z, в котором листья, представляющие y^* и z^*, являются одноранговыми.

Теперь легко прийти к противоречию: если удалить из Z листья с метками y^* и z^* и пометить их бывшего родителя ω, мы получаем дерево Z', определяющее префиксный код S'. По тому же принципу, по которому дерево T получается из T', дерево Z получается из Z' посредством добавления листьев y^* и z^* под ω; следовательно, равенство из (4.32) также применимо к Z и Z': $\mathrm{ABL}(Z') = \mathrm{ABL}(Z) - f_\omega$.

Но мы предполагали, что $\mathrm{ABL}(Z) < \mathrm{ABL}(T)$; вычитая f_ω из обеих сторон неравенства, получаем $\mathrm{ABL}(Z') < \mathrm{ABL}(T')$, что противоречит оптимальности T' как префиксного кода S'. ■

Реализация и время выполнения

Очевидно, что алгоритм Хаффмана может выполняться за полиномиальное время в зависимости от k — количества букв в алфавите. Рекурсивные вызовы алгоритма определяют серию $k - 1$ итераций по последовательно уменьшающимся алфавитам. Каждая итерация, кроме последней, состоит из простого нахождения двух букв с наименьшей частотой и слияния их в одну букву с суммарной частотой. Даже

при простейшей реализации выявление букв с наименьшей частотой осуществляется за один проход по алфавиту с временем $O(k)$, так что суммирование по $k - 1$ итерациям дает время $O(k^2)$.

Но в действительности алгоритм Хаффмана идеально подходит для использования приоритетной очереди. Вспомните, что приоритетная очередь используется для хранения множества из k элементов, каждый из которых имеет числовой ключ, и поддерживает операции вставки новых элементов и извлечения элемента с минимальным ключом. Таким образом, алфавит S может храниться в приоритетной очереди с использованием частоты каждой буквы в качестве ключа. При каждой итерации дважды извлекается минимальный элемент (две буквы с наименьшими частотами), после чего вставляется новая буква с ключом, равным сумме двух минимальных частот. В результате приоритетная очередь содержит представление алфавита, необходимое для следующей итерации.

С реализацией приоритетных очередей на базе кучи (см. главу 2) операции вставки и извлечения минимума выполняются за время $O(\log k)$; следовательно, каждая итерация, в которой выполняются всего три такие операции, выполняется за время $O(\log k)$. Суммирование по всем k итерациям дает общее время выполнения $O(k \log k)$.

Расширения

Структура оптимальных префиксных кодов, которые рассматривались в этом разделе, является фундаментальным результатом исследований в области сжатия данных. Но также важно понимать, что оптимальность никоим образом не означает, что обнаружен лучший способ сжатия данных в любых обстоятельствах.

Когда оптимального префиксного кода может оказаться недостаточно? Для начала представьте ситуацию с передачей черно-белых изображений: каждое изображение представляет собой массив из 1000 × 1000 пикселов и каждый пиксел принимает одно из двух значений (для черного или белого цвета). Также будем считать, что типичное изображение состоит почти полностью из белых пикселов: приблизительно 1000 из миллиона пикселов имеют черный цвет, а все остальные белые. При сжатии такого изображения весь метод префиксных кодов практически бесполезен: имеется текст из миллиона символов с двухбуквенным алфавитом {*черный*, *белый*}. Получается, что текст уже закодирован с использованием одного бита на букву — минимальной возможной длины в нашей системе кодирования.

Понятно, что такие изображения должны хорошо сжиматься. Интуитивно хотелось бы использовать «часть бита» для каждого из столь многочисленных белых пикселов, даже если каждый черный пиксел будет представляться несколькими битами. (В предельном случае отправка списка координат (x,y) всех черных пикселов будет работать эффективнее отправки изображения в виде текста с миллионом битов.) Проблема в том, чтобы определить схему кодирования с четко определенным понятием «части бита». В этой области сжатия данных достигнуты определенные результаты; для подобных ситуаций разработан метод *арифметического кодирования*, а также ряд других способов.

Второй недостаток префиксных кодов в том виде, в каком они определяются здесь, заключается в том, что они не могут *адаптироваться* к изменениям в тексте. Снова рассмотрим простой пример. Допустим, мы хотим закодировать вывод программы, которая строит длинную последовательность букв из набора {*a*, *b*, *c*, *d*}. Также предположим, что в первой половине последовательности буквы *a* и *b* встречаются с равной частотой, а буквы *c* и *d* не встречаются вообще; во второй половине последовательности буквы *c* и *d* встречаются с равной частотой, а буквы *a* и *b* не встречаются вообще. В системе кодирования, разработанной в этом разделе, мы стремились организовать сжатие текста с четырехбуквенным алфавитом {*a*, *b*, *c*, *d*}, с равной частотой букв.

Но в этом примере в действительности частоты остаются стабильными в половине текста, а потом радикально изменяются. Например, следующая схема позволяет обойтись всего одним битом на букву, с небольшими затратами на хранение дополнительной информации:

- Начать с кодировки, в которой бит 0 представляет *a*, а бит 1 представляет *b*.
- На середине последовательности вставить инструкцию, которая сообщает: «Кодировка изменяется, с этого момента бит 0 представляет *c*, а бит 1 представляет *d*».
- Использовать новую кодировку для оставшейся части последовательности.

Суть в том, что небольшие затраты на описание новой кодировки могут многократно окупиться, если они сокращают среднее количество битов на букву в длинном тексте. Методы, изменяющие кодировку внутри потока, называются *адаптивными схемами сжатия*. Для многих видов данных они приводят к существенным улучшениям по сравнению со статическим методом, который был рассмотрен выше.

Все эти аспекты дают представление о направлениях, в которых велась работа над сжатием данных. Как правило, существует компромисс между эффективностью метода сжатия и его вычислительной сложностью. В частности, многие усовершенствования только что описанных кодов Хаффмана приводят к соответствующему росту вычислительных затрат, необходимых для получения сжатой версии данных, их декодирования и восстановления исходного текста. Сейчас в области поиска баланса между ними идут активные исследования.

4.9.* Ориентированные деревья с минимальной стоимостью: многофазный жадный алгоритм

Мы рассмотрели уже много примеров жадных алгоритмов. Как вы могли убедиться, принципы их работы основательно различаются. Многие жадные алгоритмы принимают решение об исходной «упорядоченности» своих входных данных, а затем обрабатывают все за один проход. Другие алгоритмы принимают большее

количество пошаговых решений — также локальных и «недальновидных», не подчиненных некоему «глобальному плану». В этом разделе будет рассмотрена задача, которая расширит ваши интуитивные представления о жадных алгоритмах.

Задача

Требуется вычислить для направленного графа ориентированное дерево с минимальной стоимостью. Фактически это аналог задачи нахождения минимального остовного дерева для направленных графов; как вы убедитесь, переход к направленным графам значительно усложняет ситуацию. В то же время алгоритм ее решения сильно напоминает другие жадные алгоритмы, потому что решение строится по локальному, недальновидному правилу.

Начнем с базовых определений. Пусть $G = (V, E)$ — направленный граф, в котором один узел $r \in V$ выделен как корневой. *Ориентированное дерево* (по отношению к r) фактически представляет собой направленное остовное дерево с корнем в r. А конкретнее — это такой подграф $T = (V, F)$, что T является остовным деревом графа G, если не считать направленности ребер; и существует путь T из r к каждому другому узлу $v \in V$, если учитывать направленность ребер. На рис. 4.18 приведен пример двух разных ориентированных деревьев одного направленного графа.

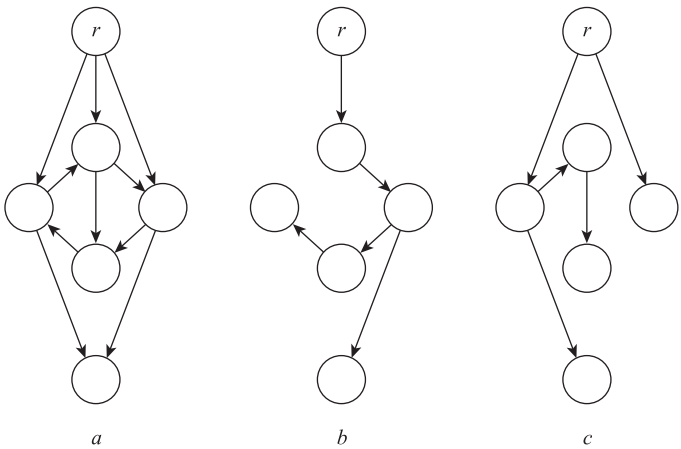


Рис. 4.18. Направленный граф может иметь много разных ориентированных деревьев. На рис. *b* и *c* изображены два разных ориентированных дерева с корнем в узле r для графа *a*

Существует полезный эквивалентный способ описания ориентированных деревьев.

(4.34) *Подграф* $T = (V, F)$ графа G является ориентированным деревом по отношению к корню r в том и только в том случае, если T не содержит циклов и для каждого узла $v \neq r$ существует ровно одно ребро из F, входящее в v.

Доказательство. Если T — ориентированное дерево с корнем r, то в любой другой узел v входит ровно одно ребро: это просто последнее ребро уникального пути r-v.

И наоборот, допустим, T не содержит циклов, а каждый узел $v = r$ имеет ровно одно входное ребро. Чтобы установить, что T является ориентированным деревом, достаточно показать, что существует направленный путь от r к любому другому узлу v. Этот путь строится следующим образом: мы начинаем с v и последовательно переходим по ребрам в обратном направлении. Так как T не содержит циклов, мы никогда не вернемся к ранее посещенному узлу, а следовательно, этот процесс должен завершиться. Но r — единственный узел без входных ребер, поэтому процесс должен завершиться с достижением r; последовательность посещенных узлов образует путь (в обратном направлении) от r к v. ■

По аналогии с тем, как у каждого связного графа существует остовное дерево, направленный граф имеет ориентированное дерево с корнем r при условии, что из r можно достигнуть любого узла. Действительно, в этом случае ребра дерева поиска в ширину с корнем r образуют ориентированное дерево.

(4.35) В направленном графе G существует ориентированное дерево с корнем r в том, и только в том случае, если существует направленный путь от r к любому другому узлу.

Базовая задача, которую мы здесь рассматриваем, такова: имеется направленный граф $G = (V, E)$ с выделенным корневым узлом r и неотрицательной стоимостью $c_e \geq 0$ каждого ребра. Требуется вычислить ориентированное дерево с корнем r и минимальной суммарной стоимостью (оно будет называться *оптимальным ориентированным деревом*). В дальнейшем будем считать, что G по крайней мере содержит ориентированное дерево с корнем r; согласно (4.35), это можно легко проверить с самого начала.

Разработка алгоритма

С учетом отношений между деревьями и ориентированными деревьями задача нахождения ориентированного дерева с минимальной стоимостью, конечно, сильно напоминает задачу нахождения минимального остовного дерева для ненаправленных графов. Следовательно, будет естественно для начала поинтересоваться, нельзя ли напрямую применить идеи, разработанные для той задачи, в этой ситуации. Например, должно ли ориентированное дерево с минимальной стоимостью содержать ребро с наименьшей стоимостью во всем графе? Можно ли безопасно удалить самое «дорогостоящее» ребро в цикле и быть полностью уверенным в том, что оно отсутствует в оптимальном ориентированном дереве?

Разумеется, ребро с наименьшей стоимостью в G не будет принадлежать оптимальному ориентированному дереву, если e входит в корневой узел, поскольку искомое ориентированное дерево не должно иметь ребер, входящих в корень. Но даже если ребро с наименьшей стоимостью в G принадлежит ориентированному дереву с корнем r, оно не обязательно принадлежит оптимальному, как показывает

пример на рис. 4.19. Действительно, включение ребра со стоимостью 1 на рис. 4.19 не позволит включить ребро со стоимостью 2 из корня r (так как узел может иметь только одно входное ребро), а это, в свою очередь, приводит к неприемлемой стоимости 10 при включении другого ребра из r. Такие аргументы никогда не затрудняли решение задачи минимального остовного дерева, в которой всегда было безопасно включить ребро с минимальной стоимостью; это наводит на мысль, что поиск оптимального ориентированного дерева может быть намного более сложной задачей. (Стоит заметить, что оптимальное ориентированное дерево на рис. 4.19 также включает ребро цикла с максимальной стоимостью; при другой структуре оптимальное ориентированное дерево даже может содержать ребро с наибольшей стоимостью во всем графе.)

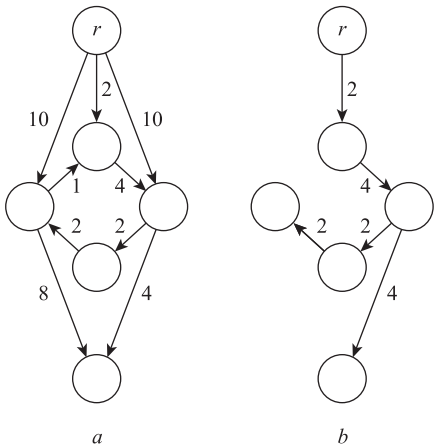


Рис. 4.19. *a* — направленный граф с обозначенной стоимостью ребер; и *b* — оптимальное ориентированное дерево с корнем в узле r этого графа

Несмотря на все сложности, для этой задачи можно спроектировать алгоритм жадного типа, просто наше недальновидное правило выбора ребер должно быть чуть более сложным. Для начала разберемся, почему не работает общая стратегия включения ребер с наименьшей стоимостью. Формальное описание этой стратегии выглядит так: для каждого узла $v \neq r$ выбирается ребро с минимальной стоимостью, входящее в v (с произвольной разбивкой «ничьих»); обозначим это множество из $n - 1$ ребер F^*. Рассмотрим подграф (V, F^*). Так как мы знаем, что оптимальное ориентированное дерево должно иметь ровно одно ребро, входящее в каждый узел $v \neq r$, а (V, F^*) представляет способ принятия решений с минимальной стоимостью, мы приходим к следующему факту.

(4.36) Если (V, F^*) — ориентированное дерево, то это ориентированное дерево с минимальной стоимостью.

Итак, сложность в том, что (V, F^*) может и не быть ориентированным деревом. В этом случае из (4.34) следует, что в (V, F^*) должен присутствовать цикл C, не включающий корень. Теперь нужно решить, что делать в такой ситуации.

Чтобы рассуждения стали более понятными, начнем со следующего наблюдения. Каждое ориентированное дерево содержит ровно одно ребро, входящее в каждый узел $v \neq r$; если выбрать некоторое ребро v и вычесть постоянную величину из стоимости каждого ребра, входящего в v, то общая стоимость каждого ориентированного дерева изменится на одинаковую величину. По сути, это означает, что фактическая стоимость всех остальных ребер, входящих в v, относительна этой величины. Пусть y_v означает минимальную стоимость любого ребра, входящего в v. Для каждого ребра $e = (u, v)$ со стоимостью $c_e \geq 0$ определяется *модифицированная стоимость* c'_e, равная $c_e - y_v$. Поскольку $c_e \geq y_v$, все модифицированные стоимости остаются неотрицательными. Но важнее то, что из этого вытекает следующий факт.

(4.37) *T является оптимальным ориентированным деревом в G со стоимостями $\{c_e\}$ в том, и только в том случае, если оно также является оптимальным ориентированным деревом с модифицированными стоимостями $\{c'_e\}$.*

Доказательство. Возьмем произвольное ориентированное дерево T. Разность между его стоимостью с набором стоимостей $\{c_e\}$ и $\{c'e\}$ равна в точности $\sum_{v \neq r} y_v$, то есть

$$\sum_{e \in T} c_e - \sum_{e \in T} c'_e = \sum_{v \in r} y_v.$$

Это объясняется тем, что у ориентированного дерева в каждый узел v из суммы входит ровно одно ребро. Поскольку разность между двумя ребрами не зависит от выбора ориентированного дерева T, мы видим, что граф T имеет минимальную стоимость с $\{c_e\}$ в том, и только в том случае, если он имеет минимальную стоимость с $\{c'_e\}$. ■

Теперь рассмотрим задачу в контексте стоимостей $\{c'_e\}$. Все ребра в множестве F^* имеют стоимость 0 с модифицированными стоимостями; таким образом, если (V, F^*) содержит цикл C, мы знаем, что стоимость всех ребер в C равна 0. Из этого следует, что мы можем использовать сколько угодно ребер из C (не нарушающих правила построения ориентированного дерева), поскольку включение ребер из C не повышает стоимость.

В продолжение работы алгоритма C свертывается в один суперузел с формированием меньшего графа $G' = (V', E')$. Здесь V' содержит узлы $V - C$ плюс один узел c^*, представляющий C. Каждое ребро $e \in E$ преобразуется в ребро $e' \in E'$ заменой каждого конца e, принадлежащего C, новым узлом c^*. В результате у G' могут появиться параллельные ребра (то есть ребра с одинаковыми концами), это нормально; однако из E' удаляются петли — ребра, у которых оба конца равны c^*. В этом уменьшенном графе G' со стоимостями $\{c'_e\}$ проводится рекурсивный поиск оптимального ориентированного дерева. Ориентированное дерево, возвращаемое рекурсивным вызовом, может быть преобразовано в ориентированное дерево графа G включением всех ребер цикла C, кроме одного.

Ниже приводится полный алгоритм.

```
Для каждого узла v ≠ r
   Пусть y_v — минимальная стоимость ребра, входящего в узел v
   Заменить стоимости всех ребер e, входящих в v, на c'_e=c_e − y_v
Выбрать одно ребро со стоимостью 0, входящее в каждый узел v ≠ r; полученное
множество обозначается F*
Если F* образует ориентированное дерево, вернуть его
Иначе существует направленный цикл C ⊆ F*
   Свернуть C в один суперузел; полученный граф обозначается G' = (V', E')
   Рекурсивно найти оптимальное ориентированное дерево (V', F') в G'
         со стоимостями {c'_e}
   Развернуть (V', F') в ориентированное дерево (V, F) в G,
      добавляя все ребра C, кроме одного
```

Анализ алгоритма

Этот алгоритм легко реализовать так, чтобы он выполнялся за полиномиальное время. Но приведет ли он к оптимальному ориентированному дереву? Прежде чем делать такой вывод, необходимо учесть одно обстоятельство: не каждое ориентированное дерево в G соответствует ориентированному дереву в свернутом графе G'. Не могли ли мы «упустить» настоящее оптимальное ориентированное дерево в G, сосредоточившись на G'?

Ориентированные деревья G' однозначно соответствуют ориентированным деревьям G, *которые имеют ровно одно ребро*, *входящее в цикл* C; и эти соответствующие ориентированные деревья имеют одинаковую стоимость в отношении $\{c'_e\}$, потому что C состоит из ребер стоимости 0. (Мы говорим, что ребро $e = (u, v)$ входит в C, если v принадлежит C, а u не принадлежит.) Итак, чтобы доказать, что наш алгоритм находит оптимальное ориентированное дерево в G, необходимо доказать, что G имеет оптимальное ориентированное дерево с ровно одним ребром, входящим в C. Сейчас мы это сделаем.

(4.38) Пусть C — цикл в G, состоящий из ребер со стоимостью 0, такой, что $r \notin C$. Тогда существует оптимальное ориентированное дерево с корнем r, в котором в C входит ровно одно ребро.

Доказательство. Рассмотрим оптимальное ориентированное дерево T в G. Так как в T от r существует путь к каждому узлу, существует как минимум одно ребро в T, которое входит в C. Если T входит в C ровно один раз, дело сделано. В противном случае предположим, что T входит в C многократно. Мы покажем, как изменить его для получения ориентированного дерева с не большей стоимостью, которое входит в C ровно один раз.

Пусть $e = (a, b)$ — ребро, входящее в C, которое лежит на самом коротком возможном пути от r. В частности, это означает, что никакие ребра на пути от r не могут входить в C. Удалим все ребра T, входящие в C, кроме ребра e. Добавим во все ребра C, кроме ребра, входящего в b, — конец ребра e. Полученный подграф G обозначим T'.

Утверждается, что T' также является ориентированным деревом. Если утверждение истинно, мы приходим к желаемому результату, поскольку стоимость T' очевидно не больше стоимости T: единственные ребра T', которые не

принадлежат также T, имеют стоимость 0. Почему же T' является ориентированным деревом?

Заметим, что у T' ровно одно ребро входит в каждый узел $v \neq r$, и никакое ребро не входит в r. Следовательно, T' содержит ровно $n - 1$ ребро; если нам удастся показать, что для каждого v в T' существует путь r-v, то граф T' должен быть связным в ненаправленном смысле, а следовательно, является деревом. Это означает, что он удовлетворяет исходному определению ориентированного дерева.

Итак, рассмотрим любой узел $v = r$; мы должны показать, что в T' существует путь r-v. Если $v \in C$, можно использовать тот факт, что путь в T от r к e был сохранен при конструировании T'; значит, чтобы достигнуть v, нужно сначала перейти к e, а потом следовать по ребрам цикла C. Теперь предположим, что $v \notin C$, и введем обозначение P для пути r-v в T. Если путь P не касается C, то он существует и в T'. В противном случае пусть w — последний узел в $P \cap C$, а P' — подпуть P от w к v. Заметьте, что все ребра в P' также существуют в T'. Ранее уже было показано, что узел w достижим из r в T', потому что он принадлежит C. Объединяя этот путь к w с подпутем P', мы получаем путь к v. ■

Теперь можно собрать все воедино и убедиться в правильности алгоритма.

(4.39) Алгоритм находит оптимальное ориентированное дерево с корнем r в графе G.

Доказательство. Воспользуемся индукцией по количеству узлов в G. Если ребра F образуют ориентированное дерево, то алгоритм вернет оптимальное ориентированное дерево согласно (4.36). В противном случае рассмотрим задачу с модифицированными стоимостями $\{c'e\}$, эквивалентную согласно (4.37). После свертки цикла C со стоимостью 0 для получения меньшего графа G' алгоритм создает оптимальное ориентированное дерево в G' согласно индукционной гипотезе. Наконец, согласно (4.38), в G существует оптимальное ориентированное дерево, соответствующее оптимальному ориентированному дереву, вычисленному для G'. ■

Упражнения с решениями

Упражнение с решением 1

Предположим, что трое ваших друзей, вдохновившись неоднократными просмотрами культового фильма «Ведьма из Блэр», решили отправиться по Аппалачской тропе. Они хотят проходить как можно большее расстояние в дневное время, но по очевидным причинам не после наступления темноты. На карте обозначено множество хороших точек для разбивки лагеря. Чтобы выбрать место для остановки, они используют следующее правило: подходя к очередной возможной точке, они определяют, удастся ли им добраться до следующей точки до наступления темноты. Если время позволяет, то они продолжают идти, а если нет — останавливаются.

Несмотря на множество серьезных недостатков, эта система, по мнению ваших друзей, обладает одной полезной особенностью. «Раз мы идем только днем, — говорят они, — необходимое количество стоянок сводится к минимуму».

Так ли это? Предлагаемая система выбора является жадным алгоритмом, и мы хотим определить, минимизирует ли он количество необходимых остановок.

Чтобы уточнить формулировку, сделаем несколько упрощающих предположений. Тропа будет моделироваться длинным отрезком прямой длины L; предполагается, что ваши друзья могут проходить d миль в день (независимо от рельефа, погоды и т. д.) Возможные точки остановок расположены на расстояниях $x_1, x_2, ..., x_n$ от начала тропы. Также будем считать (очень великодушно), что ваши друзья никогда не ошибаются в оценке того, успеют ли они добраться до следующей точки остановки до наступления темноты.

Множество точек остановки считается *действительным*, если расстояние между каждой смежной парой не превышает d (для первой точки считается расстояние от начала тропы, а от последней — до конца тропы). Таким образом, действительность множества точек означает, что турист будет останавливаться только в этих точках и при этом доберется до конца тропы. Естественно, предполагается, что полное множество из n точек действительно, иначе пройти весь путь не удастся.

Теперь исходный вопрос можно сформулировать следующим образом. Является ли жадный алгоритм ваших друзей (ежедневный переход на максимально возможное расстояние) *оптимальным* в том смысле, что он находит действительное множество с минимально возможным размером?

Решение

Жадный алгоритм часто на первый взгляд выглядит правильным. Но прежде чем поддаваться его интуитивной привлекательности, полезно спросить себя: почему он может не работать? Какие скрытые проблемы в нем могут присутствовать?

Описанный алгоритм создает естественный вопрос: а не получится ли так, что более ранняя остановка позволит лучше синхронизироваться с возможностями выбора будущих дней? Но если над этим задуматься, начинаешь интересоваться, действительно ли это возможно. Может ли существовать альтернативное решение, которое намеренно отстает от жадного решения, а потом резко «набирает скорость» и обгоняет его? И как оно может обогнать, если жадное решение каждый день перемещается на максимально возможное расстояние?

Кажется, последний вопрос может стать основой для рассуждений, основанных на принципе «опережения» из раздела 4.1. Возможно, удастся показать, что если жадная стратегия идет впереди в любой конкретный день, никакое другое решение не сможет догнать и обогнать его в будущем.

Чтобы доказать, что этот алгоритм действительно оптимален, следует определить естественный смысл, по которому выбираемые им точки «идут впереди» любого другого действительного множества точек остановки. И хотя доказательство строится по той же схеме, что и доказательство из раздела 4.1, стоит отметить

интересный контраст с задачей интервального планирования: в том случае доказывалось, что жадный алгоритм максимизирует нужную характеристику, а сейчас мы стремимся к минимизации некоторой характеристики.

Пусть $R = \{x_{p_1}, ..., x_{p_k}\}$ — множество точек остановки, выбранное жадным алгоритмом. Действуя от противного, предположим, что существует меньшее действительное множество точек остановки $S = \{x_{q_1}, ..., x_{q_m}\}$, такое что $m < k$.

Чтобы прийти к противоречию, сначала покажем, что точка остановки, достигаемая жадным алгоритмом в каждый день j, находится дальше точки остановки, достигаемой альтернативным решением, то есть

(4.40) Для всех $j = 1, 2, ..., m$ выполняется условие $x_{p_j} \geq x_{q_j}$.

Доказательство. Воспользуемся индукцией по j. Случай $j = 1$ очевиден из определения жадного алгоритма: в первый день туристы приходят максимально возможное расстояние, прежде чем остановиться. Теперь предположим, что $j > 1$, а предположение истинно для всех $i < j$. Тогда

$$x_{q_j} - x_{q_{j-1}} \leq d,$$

так как S является действительным множеством точек остановки, и

$$x_{q_j} - x_{p_{j-1}} \leq x_{q_j} - x_{q_{j-1}},$$

так как $x_{p_{j-1}} \geq x_{q_{j-1}}$ по индукционной гипотезе. Объединяя эти два неравенства, получаем

$$x_{q_j} - x_{p_{j-1}} \leq d.$$

Это означает, что у ваших друзей имеется возможность пройти весь путь от $x_{p_{j-1}}$ до x_{q_j} за один день; а значит, точка x_{p_j}, в которой они в конечном счете остановятся, может находиться только дальше x_{q_j}. (Обратите внимание на сходство с соответствующим доказательством для задачи интервального планирования: здесь жадный алгоритм «идет впереди», потому что на каждом шаге выбор, принимаемый альтернативным решением, является одним из его действительных вариантов.) ■

В частности, из (4.40) следует, что $x_{q_m} \leq x_{p_m}$. Теперь, если $m < k$, должно выполняться условие $x_{p_m} < L - d$, потому что в противном случае туристам не пришлось бы останавливаться в точке $x_{p_{m+1}}$. Объединяя эти два неравенства, можно сделать вывод, что $x_{q_m} < L - d$; но это противоречит предположению о том, что S является действительным множеством точек остановки.

Соответственно, предположение $m < k$ было ложным, и мы доказали, что жадный алгоритм создает действительное множество точек остановки минимального возможного размера.

Упражнение с решением 2

Ваши друзья открывают компанию, занимающуюся компьютерной безопасностью, и им необходимо получить лицензии на n разных криптографических продуктов. Согласно действующим требованиям, они могут получать не более одной лицензии в месяц.

Каждая лицензия в настоящее время стоит \$100, но со временем цена растет по экспоненциальному закону: в частности, лицензия j ежемесячно умножается на коэффициент $r_j > 1$ (где r_j — заданный параметр). Таким образом, если лицензия j приобретается через t месяцев, она будет стоить $100 \cdot r_j^t$. Будем считать, что все скорости роста цены различны, то есть $r_i \neq r_j$ для $i \neq j$ (хотя все они начинаются с одной исходной цены \$100).

Вопрос: если компания может покупать не более одной лицензии в месяц, в каком порядке следует совершать покупки, чтобы общая потраченная сумма была минимальной?

Приведите алгоритм, который получает n скоростей роста цены $r_1, r_2, ..., r_n$ и вычисляет порядок покупки лицензий, чтобы общая сумма потраченных денег была минимальной. Время выполнения алгоритма должно быть полиномиальным по n.

Решение

Два естественных варианта выбора последовательности покупки — сортировка r_i в порядке убывания или возрастания. Сталкиваясь с подобными альтернативами, бывает полезно проработать маленький пример и посмотреть, не снимает ли он по крайней мере один из этих вариантов. Проверим вариант $r_1 = 2$, $r_2 = 3$ и $r_3 = 4$. Покупка лицензий по возрастанию обеспечивает общую стоимость

$$100(2 + 3^2 + 4^3) = 7500,$$

тогда как при покупке в порядке убывания получается общая стоимость

$$100(4 + 3^2 + 2^3) = 2100.$$

Вариант с упорядочением по возрастанию можно сразу отбросить. (С другой стороны, это не значит, что вариант с упорядочением по убыванию правилен, но по крайней мере мы исключили один из двух варинатов.)

Попробуем доказать, что сортировка r_i по убыванию действительно всегда приводит к оптимальному решению. Когда жадный алгоритм применяется в подобных задачах, в которых требуется разместить группу значений в оптимальном порядке, вы уже видели, что для доказательства правильности часто бывает эффективно воспользоваться методом замены.

Предположим, имеется оптимальное решение O, отличное от решения S. (Другими словами, S состоит из лицензий, отсортированных по убыванию). Значит, оптимальное решение O должно содержать инверсию, то есть должны существовать два смежных месяца t и $t + 1$, для которых скорость роста цены лицензии, купленной в месяце t (обозначим ее r_t), меньше, чем купленной в месяце $t + 1$ (обозначается r_{t+1}). Следовательно, $r_t < r_{t+1}$.

Утверждается, что, поменяв местами эти две покупки, можно строго улучшить оптимальное решение; это противоречит исходному предположению об оптимальности *O*. Следовательно, если нам удастся это доказать, мы успешно покажем, что алгоритм правилен.

Если поменять местами эти две покупки, остальные покупки будут иметь одинаковые цены. В *O* сумма, уплаченная за два месяца, задействованных в замене, будет равна $100\left(r_t^t + r_{t+1}^{t+1}\right)$. С другой стороны, если поменять эти две покупки местами, мы заплатим $100\left(r_{t+1}^t + r_t^{t+1}\right)$. Поскольку константа 100 является общей для обоих выражений, нужно показать, что второй множитель больше первого. Итак, мы хотим показать, что

$$r_{t+1}^t + r_t^{t+1} < r_t^t + r_{t+1}^{t+1}$$

$$r_t^{t+1} - r_t^t < r_{t+1}^{t+1} - r_{t+1}^t$$

$$r_t^t\left(r_t - 1\right) < r_{t+1}^t\left(r_{t+1} - 1\right).$$

Но последнее неравенство истинно просто потому, что $r_i > 1$ для всех *i*, а значит, $r_t < r_{t+1}$.

На этом доказательство правильности можно считать завершенным. Время выполнения алгоритма равно $O(n \log n)$, поскольку сортировка занимает именно такое время, а остальные операции (вывод) выполняются за линейное время. Итак, общее время выполнения равно $O(n \log n)$.

Примечание: интересно, что даже при небольшом изменении вопроса ситуация существенно усложняется. Предположим, вместо покупки лицензий с возрастающей ценой вы пытаетесь продавать оборудование, цена которого последовательно снижается. Цена элемента *i* снижается с коэффициентом $r_i < 1$ в месяц, начиная с \$100, так что при продаже его через *t* месяцев вы получите $100 \cdot r_i^t$. (Иначе говоря, скорость роста теперь не больше, а меньше 1.) Если в месяц продается только один вид оборудования, в каком оптимальном порядке их следует продавать? Как выясняется, существуют ситуации, в которых оптимальное решение не упорядочивает коэффициенты по возрастанию или убыванию (как с входными данными $\frac{3}{4}, \frac{1}{2}, \frac{1}{100}$).

Упражнение с решением 3

Предположим, имеется связный граф *G*; будем считать, что стоимости всех ребер различны. Граф *G* содержит *n* вершин и *m* ребер. Задано конкретное ребро *e* графа *G*. Предложите алгоритм со временем выполнения $O(m + n)$, который будет решать, содержится ли *e* в минимальном остовном дереве *G*.

Решение

Мы знаем, что принадлежность ребра *e* минимальному остовному дереву проверяется по двум правилам: свойство сечения (4.17) утверждает, что *e* присутствует

в минимальном остовном дереве, если это ребро с наименьшей стоимостью, переходящее из некоторого множества S в дополнение $V - S$; а свойство цикла (4.20) утверждает, что e не включается ни в какое минимальное остовное дерево, если это ребро с наибольшей стоимостью в некотором цикле C. Удастся ли нам использовать эти два правила как часть алгоритма, который решает задачу за линейное время?

Оба свойства фактически определяют, в каких отношениях e находится с множеством ребер, обладающих *меньшей* стоимостью, чем у e. Свойство сечения можно рассматривать как вопрос: существует ли такое множество $S \subseteq V$, чтобы для перехода из S в $V - S$ без использования e пришлось бы использовать ребро с большей стоимостью, чем у e? И если взглянуть на цикл C в контексте свойства цикла, «длинный путь» по C (в обход e) может рассматриваться как альтернативный маршрут между конечными точками e, использующими только ребра с меньшей стоимостью.

Объединяя эти два наблюдения, мы приходим к следующему утверждению.

(4.41) Ребро $e = (v, w)$ не принадлежит минимальному остовному дереву графа G в том, и только в том случае, если v и w можно соединить путем, состоящим исключительно из ребер с меньшей стоимостью, чем у e.

Доказательство. Сначала предположим, что P — путь v-w, состоящий исключительно из ребер с меньшей стоимостью, чем у e. Если добавить e в P, мы получим цикл, для которого e является самым «дорогостоящим» ребром. Следовательно, по свойству цикла e не принадлежит минимальному остовному дереву G.

С другой стороны, предположим, что v и w невозможно соединить путем, состоящим исключительно из ребер со стоимостью меньшей, чем у e. Определим множество S, для которого e является ребром с наименьшей стоимостью, один конец которого находится в S, а другой в $V - S$; если это возможно, то из свойства сечения следует, что e принадлежит каждому минимальному остовному дереву. Наше множество S будет представлять собой множество всех узлов, к которым можно перейти из v по пути, состоящему только из ребер со стоимостью меньшей, чем у e. Согласно нашему предположению, $w \in V - S$. Кроме того, по определению S не может существовать ребро $f = (x, y)$ со стоимостью меньшей, чем у e, для которого один конец x принадлежит S, а другой конец y — $V - S$. В самом деле, если бы такое ребро f существовало, то раз к узлу x можно перейти из v по ребрам только с меньшей стоимостью, чем у e, узел y также был бы достижим. Следовательно, e — ребро с наименьшей стоимостью, один конец которого принадлежит S, а другой — $V - S$, как и требовалось. Доказательство закончено. ■

С учетом этого факта наш алгоритм выглядит так. Мы строим граф G', удаляя из G все ребра с весом, превышающим c_e (а также само ребро e). Затем при помощи одного из алгоритмов связности из главы 3 он определяет, существует ли путь от v к w в G'. Согласно (4.41), e принадлежит минимальному остовному дереву в том, и только в том случае, если такой путь не существует.

Время выполнения этого алгоритма включает $O(m + n)$ для построения G' и $O(m + n)$ для проверки пути от v к w.

Упражнения

1. Решите, истинно или ложно следующее утверждение. Если оно истинно, приведите краткое объяснение, а если ложно — контрпример.

   Пусть G — произвольный связный ненаправленный граф с различающимися стоимостями $c(e)$ всех ребер e. Предположим, e^* — ребро с минимальной стоимостью в G, то есть $c(e^*) < c(e)$ для каждого ребра $e \neq e^*$. Тогда существует минимальное остовное дерево T графа G, содержащее ребро e^*.

2. Решите, истинно или ложно каждое из следующих двух утверждений. Если утверждение истинно, приведите краткое объяснение, а если ложно — контрпример.

   (a) Предположим, имеется экземпляр задачи нахождения минимального остовного дерева для графа G, все стоимости ребер которого положительны и различны. Пусть T — минимальное остовное дерево для этого экземпляра.

   Стоимость каждого ребра c_e заменяется его квадратом c_e^2; так образуется новый экземпляр задачи с тем же графом, но другими стоимостями.

   Истинно или ложно? Дерево T должно остаться минимальным остовным деревом для нового экземпляра задачи.

   (b) Предположим, имеется экземпляр задачи нахождения кратчайшего пути s–t в направленном графе G. Все стоимости ребер положительны и различны. Пусть P — путь s–t для этого экземпляра с минимальной стоимостью.

   Стоимость каждого ребра c_e заменяется его квадратом c_e^2; так образуется новый экземпляр задачи с тем же графом, но другими стоимостями.

   Истинно или ложно? Путь P должен остаться путем s–t с минимальной стоимостью для нового экземпляра задачи.

3. Вы консультируете транспортную компанию, которая перевозит грузы между Нью-Йорком и Бостоном. Объем перевозок достаточно велик, чтобы ежедневно между двумя городами отправлялось несколько грузовиков. У каждого грузовика имеется предельный вес перевозимых грузов W. Грузы прибывают на склад в Нью-Йорке один за другим, каждый пакет i имеет вес w_i. Склад невелик, поэтому в любой момент времени на нем может находиться только один грузовик. По действующим в компании правилам грузы отправляются в порядке их поступления, чтобы покупатели не сердились, если груз, отправленный позднее их груза, доберется до Бостона раньше. Сейчас в компании используется простой жадный алгоритм упаковки: грузы упаковываются в порядке поступления, а если следующий груз не помещается, то грузовик отправляется без него.

   Но владельцы компании подозревают, что они используют слишком много грузовиков, и хотят повысить эффективность. Нельзя ли сократить количество машин, отправляя незаполненный грузовик, чтобы следующие грузовики заполнялись более эффективно?

   Докажите, что для заданного набора грузов с заданными весами жадный алгоритм, используемый в настоящее время, действительно минимизирует количество используемых грузовиков. **Доказательство** должно строиться по

тому же образцу, что и приведенное выше доказательство для задачи интервального планирования: оптимальность жадного алгоритма устанавливается определением метрики, в соответствии с которой он «опережает» все остальные решения.

4. Ваши друзья занялись расширяющейся областью анализа временных рядов. Целью анализа является поиск закономерностей в последовательностях событий, происходящих во времени. Одним из источников таких данных с естественным упорядочением по времени могут служить покупки на биржах (что и когда покупается). Ваши друзья хотят иметь эффективный механизм обнаружения некоторых «закономерностей» в заданной последовательности S таких событий: например, они хотят знать, происходят ли в последовательности S четыре события

   `купить Yahoo, купить eBay, купить Yahoo, купить Oracle`

   в такой очередности (но не обязательно подряд).

   Они начинают с набора всех возможных событий (например, возможных транзакций) и последовательности S из n таких событий. Заданное событие может встречаться в S несколько раз (например, акции Yahoo могут покупаться многократно в одной последовательности S). Последовательность S' называется *подпоследовательностью* S, если из S возможно удалить некоторые события так, чтобы оставшиеся события (с соблюдением порядка) были равны последовательности S'. Так, приведенная последовательность из четырех событий является подпоследовательностью для последовательности

   `купить Amazon, купить Yahoo, купить eBay, купить Yahoo, купить Yahoo, купить Oracle`

   Ваши друзья хотят вводить короткие последовательности и быстро узнавать, являются ли они подпоследовательностями S. Поставлена следующая задача: предложите алгоритм, который получает две последовательности событий — S' длины m и S длины n (в обеих последовательностях события могут повторяться) и за время $O(m + n)$ решает, является ли S' подпоследовательностью S.

5. Представьте длинную, спокойную проселочную дорогу, на которой редко встречаются отдельные дома. (Будем считать, что дорога представляет собой отрезок прямой с двумя концами, восточным и западным.) Также предположим, что несмотря на пасторальную обстановку, жители этих домов активно пользуются мобильными телефонами. Требуется разместить базовые станции сотовой связи вдоль дороги так, чтобы любой дом находился на расстоянии не более четырех миль от одной из базовых станций.

   Предложите эффективный алгоритм для достижения этой цели с минимальным количеством базовых станций.

6. Ваш друг работает в детском лагере и отвечает за проведение мероприятий в группе подростков. В его планы входит проведение мини-триатлона: каждый участник должен проплыть 20 дорожек в бассейне, проехать на велосипеде 10 миль, а потом пробежать 3 мили. Участники будут выходить на дистанцию

поочередно, по следующему правилу: в любой момент времени бассейн будет использоваться только одним участником. Другими словами, первый участник проплывает 20 дорожек, выходит и садится на велосипед. Как только он покидает бассейн, второй участник начинает проплывать свои 20 дорожек; когда он выходит из бассейна, заплыв начинает третий участник... и т. д.

У каждого участника имеется прогнозируемое время плавания (время, за которое он должен проплыть 20 дорожек), прогнозируемое время проезда (время, за которое он должен проехать 10 миль) и прогнозируемое время бега (время, за которое он пробежит 3 мили). Ваш друг хочет спланировать расписание триатлона: порядок, в котором участники будут выходить на дистанцию. *Завершающим временем* расписания будет называться самое меньшее время, за которое все участники пройдут все три стадии триатлона, при условии, что они точно выдержат спрогнозированное время во всех трех стадиях (еще раз: участники могут ехать на велосипеде и бежать одновременно, но в бассейне в любой момент времени может находиться только один человек). Как выглядит оптимальный порядок выхода участников, чтобы соревнования закончились как можно раньше? А точнее, предложите эффективный алгоритм, который строит расписание с минимальным завершающим временем.

7. Невероятно популярная испаноязычная поисковая система Goog проводит огромный объем вычислений при каждом пересчете индекса. К счастью, в распоряжении компании имеется один суперкомпьютер с практически неограниченным запасом мощных рабочих станций.

   Вычисления разбиты на n заданий $J_1, J_2, ..., J_n$, которые могут выполняться полностью независимо друг от друга. Каждое задание состоит из двух фаз: сначала оно проходит предварительную обработку на суперкомпьютере, а затем завершается на одной из рабочих станций. Допустим, обработка задания J_i требует p_i секунд на суперкомпьютере, а затем f_i секунд на рабочей станции.

   На площадке доступны как минимум n рабочих станций, поэтому завершающая фаза обработки всех заданий может проходить параллельно — все задания будут выполняться одновременно. Однако суперкомпьютер может работать только с одним заданием, поэтому администратор должен определить порядок передачи заданий суперкомпьютеру. Как только первое задание в этом порядке будет обработано на суперкомпьютере, оно передается на рабочую станцию для завершения; после обработки на суперкомпьютере второе задание передается на рабочую станцию независимо от того, завершилось первое задание или нет (так как рабочие станции работают параллельно), и т. д.

   Допустим, расписание представляет собой упорядоченный список заданий для суперкомпьютера, а время завершения расписания определяется самым ранним временем завершения всех заданий на рабочих станциях. Очень важно свести к минимуму эту характеристику, так как она определяет, насколько быстро El Goog сможет построить новый индекс.

   Предложите алгоритм с полиномиальным временем, который находит расписание с минимальным временем завершения.

8. Имеется связный граф G, стоимости всех ребер которого различны. Докажите, что в G существует уникальное минимальное остовное дерево.

9. Одна из практических целей для задачи нахождения минимального остовного дерева — построение остовной сети для множества узлов с минимальной общей стоимостью. Однако сейчас нас интересует другой тип целей: проектирование остовной сети, в которой максимальная стоимость ребра настолько низка, насколько это возможно.

   Говоря конкретнее, пусть $G = (V, E)$ — связный граф с n вершинами, m ребрами и положительными стоимостями ребер (предполагается, что все стоимости различны). Пусть $T = (V, E')$ — остовное дерево G; *критическим ребром* дерева T будет называться ребро из T с наибольшей стоимостью.

   Остовное дерево T графа G называется *минимально-критическим остовным деревом*, если не существует остовного дерева T' графа G с меньшей стоимостью критичного ребра.

   (a) Является ли каждое минимально-критичное дерево графа G минимальным остовным деревом G? Докажите или приведите контрпример.

   (b) Является ли каждое минимальное остовное дерево графа G минимально-критичным деревом графа G? Докажите или приведите контрпример.

10. Пусть $G = (V, E)$ — (ненаправленный) граф со стоимостями $c_e \geq 0$ ребер $e \in E$. Допустим, вам известно минимальное остовное дерево T для G. Предположим, что в G добавляется новое ребро, соединяющее два узла $v, w \in V$ со стоимостью c.

    (a) Предложите эффективный алгоритм проверки того, остается ли T остовным деревом минимальной стоимости при добавлении нового ребра в G (но не в дерево T). Алгоритм должен выполняться за время $O(|E|)$. Удастся ли вам заставить его выполняться за время $O(|V|)$? Отметьте все предположения, которые вам пришлось сделать относительно структуры данных, используемой для представления дерева T и графа G.

    (b) Допустим, T уже не является минимальным остовным деревом. Предложите алгоритм с линейным временем ($O(|E|)$) для замены T новым минимальным остовным деревом.

11. Предположим, имеется связный граф $G = (V, E)$, ребро e которого имело стоимость c_e. В предыдущей задаче было показано, что при различных стоимостях ребер G содержит уникальное минимальное остовное дерево. Но если в графе имеются ребра с одинаковой стоимостью, G может содержать несколько минимальных остовных деревьев. Вопрос: можно ли заставить алгоритм Крускала найти все минимальные остовные деревья графа G?

    Вспомните, что алгоритм Крускала сортирует ребра в порядке увеличения стоимости, а затем жадно обрабатывает ребра одно за другим, добавляя ребро e при условии, что оно не приводит к образованию цикла. Когда некоторые ребра имеют одинаковую стоимость, выражение «в порядке увеличения стоимости» необходимо задать более тщательно: упорядочение ребер называется *действительным*, если соответствующая последовательность стоимостей ребер

не уменьшается. *Действительным выполнением* алгоритма Крускала будет называться выполнение, начинающее с действительного упорядочения ребер G.

Для любого графа G и любого минимального остовного дерева T графа G существует ли действительное выполнение алгоритма Крускала для G, которое выдает T как результат? Приведите доказательство или контрпример.

12. Предположим, имеется n видеопотоков, которые необходимо передать один за другим по каналу связи. Поток i состоит из b_i битов, которые должны передаваться с постоянной скоростью за период t_i секунд. Два потока не могут передаваться одновременно, поэтому необходимо составить *расписание* потоков, то есть порядок их отправки. Какой бы порядок ни был выбран, задержек между концом одного и началом следующего потока быть не должно. Предположим, расписание начинается во время 0 (а следовательно, завершается во время $\sum_{i=1}^{n} t_i$ независимо от выбранного порядка). Предполагается, что все значения b_i и t_i — положительные целые числа.

Владелец канала не хочет, чтобы один пользователь создавал слишком большую нагрузку на канал, поэтому он устанавливает следующее ограничение с фиксированным параметром r:

(*) Для каждого натурального числа $t > 0$ общее количество битов, передаваемых за интервал времени от 0 до t, не может превышать r_t.

Обратите внимание: ограничения устанавливаются только для интервалов, начинающихся с 0 (но не для любой другой начальной точки).

Расписание называется *действительным*, если оно удовлетворяет ограничению (*), установленному для канала.

Задача. Для заданного множества из n потоков, каждый из которых задается количеством битов b_i и продолжительностью t_i, а также для параметра канала r определите, существует ли действительное расписание.

Пример. Предположим, имеются $n = 3$ потока с параметрами $(b_1, t_1) = (2000, 1)$, $(b_2, t_2) = (6000, 2)$, $(b_3, t_3) = (2000, 1)$, а параметр канала $r = 5000$. В этом случае расписание, передающее потоки в порядке 1, 2, 3, будет действительным, так как ограничение (*) соблюдается:

$t = 1$: весь первый поток отправлен, $2000 < 5000 \cdot 1$

$t = 2$: отправлена половина второго потока, $2000 + 3000 < 5000 \cdot 2$

Аналогичные вычисления проводятся для $t = 3$ и $t = 4$.

(a) Рассмотрим следующее утверждение:

Действительное расписание существует в том, и только в том случае, если каждый поток i удовлетворяет условию $b_i \leq rt_i$.

Решите, истинно или ложно это утверждение. Предоставьте доказательство или контрпример.

(b) Предложите алгоритм, который получает множество из n потоков, каждый из которых задается количеством битов b_i и продолжительностью t_i,

а также параметр канала r и определяет, существует ли для них действительное расписание. Время выполнения алгоритма должно быть полиномиальным по n.

13. Небольшая фирма — скажем, копировальный центр с одним большим копировальным аппаратом — сталкивается с проблемой планирования. Каждое утро она получает множество заказов от клиентов. Заказы должны быть выполнены на единственном аппарате в порядке, который устроит большинство клиентов. На выполнение заказа клиента i требуется время t_i. Пусть при заданном расписании (то есть порядке заказов) время завершения заказа i обозначается C_i. Например, если задание j должно быть выполнено первым, то $C_j = t_j$; а если задание j выполняется сразу же после задания i, то $C_j = C_i + t_j$. Каждому клиенту i также назначается вес w_i, представляющий его важность для фирмы. Настроение клиента i зависит от времени завершения его заказа. Таким образом, фирма стремится упорядочить заказы так, чтобы свести к минимуму взвешенную сумму времен завершения, $\sum_{i=1}^{n} w_i C_i$.

    Предложите эффективный алгоритм для решения этой задачи. Алгоритм получает множество из n заказов с временем обработки t_i и весом w_i для каждого заказа. Требуется упорядочить заказы так, чтобы минимизировать взвешенную сумму времен завершения $\sum_{i=1}^{n} w_i C_i$.

    Пример. Предположим, получены два заказа: первый с временем t_1=1 и весом w_1=10, и второй с временем $t_2 = 3$ и весом $w_2 = 2$. Если заказ 1 будет выполнен первым, то взвешенное время завершения составит $10 \cdot 1 + 2 \cdot 4 = 18$, а если начать со второго заказа, то взвешенное время увеличится до $10 \cdot 4 + 2 \cdot 3 = 46$.

14. Вы работаете с группой консультантов в области безопасности, которые наблюдают за работой большой компьютерной системы. Особый интерес вызывают процессы с пометкой «системный». Для каждого такого процесса задано начальное и конечное время; в промежутке между этими моментами процесс непрерывно работает; у консультантов имеется список начального и конечного времени для всех системных процессов, запланированных на сегодняшний день.

    Консультанты написали программу `status_check`, которая выполняется несколько секунд и сохраняет различную информацию о системных процессах, выполняемых в системе в настоящее время. (В нашей модели каждый вызов `status_check` относится только к одному конкретному моменту времени.) Консультанты хотели бы запускать программу `status_check` как можно меньше раз за день, но достаточно для того, чтобы на протяжении выполнения каждого системного процесса P программа была запущена хотя бы один раз.

    (а) Предложите эффективный алгоритм, который получает начальное и конечное время для всех системных процессов и находит минимальное множество временных точек для вызова `status_check`, с которым программа будет вызвана хотя бы один раз на протяжении работы каждого системного процесса P.

(b) Пока вы работали над алгоритмом, консультанты тоже пытались найти подход к решению задачи. «Допустим, нам удастся найти множество из k системных процессов, обладающих тем свойством, что никакие из них никогда не выполняются одновременно. Тогда, очевидно, алгоритм должен вызвать `status_check` минимум k раз; каждый отдельный вызов `status_check` сможет обработать только один из таких процессов».

Конечно, это правильно, и в дальнейшем обсуждении возник вопрос: а не действует ли более сильное правило, своего рода обратное приведенному? Предположим, k^* — наибольшее значение k, позволяющее найти множество из k системных процессов, никакие два из которых никогда не выполняются одновременно. Значит ли это, что должно существовать множество из k^* моментов времени для запуска `status_check`, чтобы хотя бы один вызов случился во время выполнения каждого системного процесса? (Другими словами, не является ли аргумент из предыдущего абзаца единственной причиной, по которой приходится много раз вызывать `status_check`?) Решите, истинно или ложно это утверждение, и приведите доказательство или контрпример.

15. К вам обращается менеджер крупного студенческого объединения. Она руководит группой из n студентов, каждый из которых по графику должен отработать одну *смену* за неделю. Со сменами связаны различные виды работ (помощь в доставке пакетов, перезагрузка зависших информационных терминалов и т. д.), но мы можем рассматривать каждую смену как один непрерывный интервал времени. Несколько смен могут проходить одновременно.

    Менеджер хочет выбрать подмножество из этих n студентов и сформировать наблюдательный комитет, с которым будет проводить встречи раз в неделю. Комитет будет считаться *полным*, если у каждого студента, не входящего в комитет, смена перекрывается (по крайней мере частично) со сменой студента, входящего в комитет. Таким образом, качество работы каждого студента будет видно по крайней мере одному участнику комитета.

    Приведите эффективный алгоритм, который получает расписание из n смен и выдает состав полного наблюдательного комитета с минимальным количеством участников.

    Пример. Предположим, $n = 3$ и смены проводятся в следующее время:

    понедельник 16:00 — понедельник 20:00

    понедельник 18:00 — понедельник 22:00

    понедельник 21:00 — понедельник 23:00

    В этом случае наименьший полный наблюдательный комитет будет состоять только из второго студента, потому что вторая смена перекрывается как с первой, так и с третьей.

16. Консультанты по вопросам финансовой безопасности в настоящее время работают с клиентом, исследующим потенциальную схему отмывания денег. Проведенные исследования показали, что за последнее время состоялось n сомнительных операций, каждая из которых была связана с переводом денег на

некий счет. К сожалению, отрывочный характер доказательств означает, что исследователям неизвестен сам счет, суммы переводов и точное время проведения операций. Известна лишь приблизительная временная метка каждой операции; из доказательств следует, что операция i состоялась во время $t_i \pm e_i$ с некоторой «погрешностью» e_i. (Иначе говоря, операция состоялась где-то в интервале от $t_i - e_i$ до $t_i + e_i$.) Не забывайте, что у разных операций могут быть разные погрешности.

Недавно был обнаружен подозрительный банковский счет, который, как предполагается, мог быть использован в преступлении. За последнее время со счетом выполнялись n операций, происходивших во время $x_1, x_2, ..., x_n$. Чтобы узнать, тот это счет или нет, нужно связать каждое из n событий с одной из n сомнительных операций так, чтобы если событие x_i связывается с сомнительной операцией t_j, то выполнялось условие $|t_j - x_i| \le e_j$. (Иначе говоря, клиент хочет знать, можно ли связать события с сомнительными операциями так, чтобы временные расхождения находились в пределах разумной погрешности; задача усложняется тем, что заранее неизвестно, какое событие с какой погрешностью должно связываться.)

Предложите эффективный алгоритм, который получает данные и решает, существует ли такая ассоциация. Если возможно, алгоритм должен выполняться за время не более $O(n^2)$.

17. Рассмотрим следующую вариацию на тему задачи интервального планирования. Имеется компьютер, который может работать ежедневно по 24 часа в сутки. Пользователи подают заявки на выполнение ежедневных заданий на компьютере. У каждого задания имеется начальное и конечное время; если задание принято на выполнение, оно выполняется каждый день в период между начальным и конечным временем. (Обратите внимание: задания могут начинаться до полуночи и завершаться после полуночи; из-за этого ситуация отличается от той, какую мы видели в задаче интервального планирования.)

    Получив список из n таких заданий, нужно принять как можно больше заданий (независимо от их длины) с учетом ограничения, согласно которому в любой момент времени на компьютере может выполняться только одно задание. Предложите алгоритм решения этой задачи с временем, полиномиальным по n. Для простоты можно считать, что не существует заданий с одинаковым начальным и конечным временем.

    Пример. Получены следующие четыре задания, заданных парами (*начало*, *конец*):

    (18:00, 6:00), (21:00, 4:00), (3:00, 14:00), (13:00, 19:00)

    В оптимальном варианте выбираются два задания (21:00, 4:00) и (13:00, 19:00), которые могут быть запланированы без перекрытия по времени.

18. Ваши друзья собрались в зимний поход к небольшому городку возле канадской границы. Они проанализировали все варианты перемещений и построили направленный граф, узлы которого представляют промежуточные пункты, а ребра — соединяющие их дороги.

Также известно, что из-за плохой погоды некоторые дороги в этой местности зимой заметает и перемещение по ним может существенно замедлиться. К счастью, имеется отличный сайт с точными прогнозами скорости перемещения по дорогам; однако эта скорость зависит от времени года. Точнее, сайт отвечает на запросы следующего вида: имеется ребро $e = (v, w)$, соединяющее два пункта v и w; для заданного начального времени t для пункта v сайт возвращает значение $f_e(t)$ прогнозируемое время прибытия в w. Сайт гарантирует, что $f_e(t) \geq t$ для всех ребер e и моментов времени t (обратные перемещения во времени невозможны), а $f_e(t)$ является монотонно возрастающей функцией t (то есть отправившись в путь позднее, вы не приедете раньше). В остальном функции $f_e(t)$ могут быть произвольными. Например, в местах, в которых скорость перемещения не зависит от времени года, $f_e(t) = t + \ell_e$, где ℓ_e — время, необходимое для перемещения от начала ребра e к его концу.

Ваши друзья хотят использовать сайт для определения самого быстрого способа перемещения по направленному графу от начальной точки до места назначения. (Предполагается, что они отправляются во время 0, а все прогнозы сайта полностью верны.) Предложите алгоритм с полиномиальным временем для решения этой задачи; отправка одного запроса на сайт (с заданным ребром e и временем t) рассматривается как отдельный шаг вычислений.

19. Группа планирования сети из коммуникационной компании CluNet столкнулась со следующей задачей. Имеется связный граф $G = (V, E)$; узлы графа представляют точки, с которыми требуется организовать обмен данными. Каждое ребро e представляет канал связи с заданной пропускной способностью b_e.

    Для каждой пары узлов $u, v \in V$ требуется выбрать один путь P из u в v, по которому эта пара будет обмениваться данными. *Критической пропускной способностью* $b(P)$ этого пути P называется минимальная пропускная способность всех входящих в него ребер; иначе говоря, $b(P) = \min_{e \in P} b_e$. *Лучшей достижимой пропускной способностью* для пары u, v в графе G называется максимум значения $b(P)$ по всем путям P для пар u-v в графе G.

    С какого-то момента отслеживать пути для всех пар узлов в сети становится слишком сложно, и один из проектировщиков сети выступает со смелым предложением: нельзя ли найти остовное дерево T для G, в котором для каждой пары узлов u, v уникальный путь u-v в дереве будет обеспечивать лучшую достижимую пропускную способность для маршрута u-v в G? (Иначе говоря, хотя путь u-v может выбираться по всему графу, улучшить результат по сравнению с путем u-v в T все равно не удастся.)

    В CluNet эта идея подверглась дружной критике, и у этого скептицизма имеется естественная причина: пути, которые будут использоваться для обеспечения пропускной способности между двумя узлами, могут сильно отличаться. С чего бы полагать, что одно дерево одновременно решит все проблемы? Но после нескольких неудачных попыток опровергнуть идею люди начинают подозревать, что это все же возможно.

Докажите, что такое дерево существует, и предложите эффективный алгоритм его построения. Иначе говоря, предложите алгоритм построения остовного дерева T, в котором для каждой пары $u, v \in V$ пропускная способность пути u-v в T равна лучшей достижимой пропускной способности для пары u, v в графе G.

20. Ежегодно осенью в отдаленной горной местности собирается группа представителей дорожных служб, которая выбирает, какие дороги следует постоянно расчищать предстоящей зимой. В округе n городов, а дорожная сеть может рассматриваться как (связный) граф (V, E) для этого множества городов; каждое ребро представляет дорогу, соединяющую два города. Зимой людей, живущих в горах, беспокоит не столько длина дороги, сколько высота — именно она начинает определять сложность поездки.

    Каждая дорога (каждое ребро e в графе) помечается числом a_e, определяющим высоту самой высокой точки дороги. Будем считать, что в графе нет двух ребер с одинаковым значением a_e. Высота пути P в графе определяется максимальным значением a_e для всех ребер e, входящих в P. Наконец, путь между городами i и j считается оптимальным, если он обеспечивает минимально возможную высоту по всем путям от i до j.

    Представители дорожных служб собираются выбрать множество $E' \subseteq E$ дорог, которые будут расчищаться зимой; снег на остальных дорогах убираться не будет, и они станут недоступными для путешественников. Все согласны с тем, что какое бы подмножество дорог E' ни было выбрано, оно должно обладать тем свойством, что (V, E') является связным подграфом; и что еще важнее, для каждой пары городов i и j высота оптимального пути в (V, E') не должна быть больше, чем в полном графе $G = (V, E)$. Связный подграф (V, E'), обладающий этим свойством, будет называться связным подграфом с минимальной высотой.

    Но если это ключевое свойство соблюдается, дорожные службы хотели бы свести к минимуму количество расчищаемых дорог. И вот однажды они выдвигают следующую гипотезу:

    > Минимальное остовное дерево графа G с весами ребер a_e является связным графом с минимальной высотой.

    (В одной из предшествующих задач утверждалось, что при различающихся весах ребер существует уникальное минимальное остовное дерево. Следовательно, благодаря предположению о том, что все значения a_e различны, можно говорить о минимальном остовном дереве.)

    Изначально эта гипотеза выглядит неочевидной, поскольку минимальное остовное дерево стремится минимизировать сумму значений a_e, а цель минимизации высоты кажется чем-то совершенно иным. Однако при отсутствии опровержений выдвигается еще более смелая вторая гипотеза:

    > Подграф (V, E') является связным подграфом с минимальной высотой в том, и только в том случае, если он содержит ребра минимального остовного дерева.

    Обратите внимание: из второй гипотезы немедленно следует первая, так как минимальное остовное дерево содержит свои ребра.

Вопросы:

(a) Является ли первая гипотеза истинной для всех вариантов выбора G и различающихся высот a_e? Представьте доказательство или контрпример с объяснением.

(b) Является ли вторая гипотеза истинной для всех вариантов выбора G и различающихся высот a_e? Представьте доказательство или контрпример с объяснением.

21. Будем называть граф $G = (V, E)$ *квазидеревом*, если он является связным и содержит не более $n + 8$ ребер, где $n = |V|$. Предложите алгоритм с временем выполнения $O(n)$, который получает квазидерево G со стоимостями всех ребер и возвращает минимальное остовное дерево графа G. Считайте, что все стоимости ребер различны.

22. Рассмотрим задачу нахождения минимального остовного дерева для ненаправленного графа $G = (V, E)$ со стоимостью $c_e \geq 0$ каждого ребра; при этом стоимости ребер могут совпадать. Если граф может содержать ребра с одинаковой стоимостью, то и решений с минимальной стоимостью может быть несколько. Предположим, имеется остовное дерево $T \subseteq E$ с гарантией того, что каждое ребро $e \in T$ принадлежит *некоторому* минимальному остовному дереву графа G. Можно ли заключить, что T является минимальным остовным деревом G? Представьте доказательство или контрпример с объяснением.

23. Вспомните задачу вычисления ориентированного дерева с минимальной стоимостью в направленном графе $G = (V, E)$ со стоимостями ребер $c_e \geq 0$. Здесь мы рассмотрим случай, в котором G является направленным ациклическим графом (DAG), то есть не содержит направленных циклов.

    Как и в случае с направленными графами вообще, разных решений с минимальной стоимостью может быть много. Допустим, имеется направленный ациклический граф $G = (V, E)$ и ориентированное дерево $A \subseteq E$ с гарантией того, что каждое ребро $e \in A$ принадлежит некоторому ориентированному дереву с минимальной стоимостью в G. Можно ли сделать вывод, что A является ориентированным деревом с минимальной стоимостью в G? Представьте доказательство или контрпример с объяснением.

24. Цепи синхронизации — один из важнейших компонентов больших интегральных схем. Ниже представлена простая модель такой цепи. Возьмем полное сбалансированное бинарное дерево с n листьями, где n — степень 2. С каждым ребром e дерева связана длина ℓ, которая является положительным числом. *Расстояние* от корня до заданного листа вычисляется суммированием длин всех ребер пути от корня к листу.

    Корень генерирует *синхросигнал*, который распространяется по ребрам к листьям. Будем считать, что время, необходимое сигналу для перехода к заданному листу, пропорционально расстоянию от корня до листа.

    Если расстояния от листьев до корня различны, то сигнал дойдет до листьев в разное время, и это создаст большие проблемы. Мы хотим, чтобы листья были полностью синхронизированы и получали сигнал одновременно. Для этого

необходимо увеличить длины некоторых ребер, чтобы все пути от корня до листьев имели одинаковую длину (уменьшить длины ребер нельзя). Если эта задача будет решена, то дерево (с новыми длинами ребер) имеет нулевой *сдвиг*. Наша цель — обеспечить нулевой сдвиг так, чтобы сумма длин всех ребер оставалась минимальной.

Предложите алгоритм, увеличивающий длину некоторых ребер так, чтобы полученное дерево имело нулевой сдвиг, а общая длина ребер была минимально возможной.

Пример. Рассмотрим дерево на рис. 4.20, на котором буквами обозначены узлы, а числами — длины ребер.

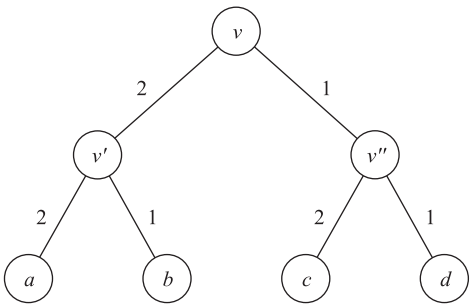


Рис. 4.20. Задача нулевого сдвига из упражнения 23

Уникальное оптимальное решение для этой задачи — взять три ребра длины 1 и увеличить каждую из этих длин до 2. Полученное дерево имеет нулевой сдвиг, а сумма длин ребер равна 12 (минимальная из возможных).

25. Имеется множество точек $P = \{p_1, p_2, ..., p_n\}$ и функция расстояния d для множества P; d — функция для пары точек в P, обладающая тем свойством, что $d(p_i, p_j) = d(p_j, p_i) > 0$, если $i \neq j$, а $d(p_i, p_i) = 0$ для каждого i.

Определим *иерархическую метрику* для P как произвольную функцию расстояния τ, которая строится следующим образом: строим корневое дерево T с n листьями и назначаем каждому узлу v дерева T (как листьям, так и дочерним узлам) высоту h_v. Высоты должны удовлетворять следующим условиям: $h(v) = 0$ для каждого листа v, и если u является родителем v в T, то $h(u) \geq h(v)$. Каждая точка из P ставится в соответствие отдельному листу T. Теперь для каждой пары точек p_i и p_j определяется расстояние между ними $\tau(p_i, p_j)$: мы находим наименьшего общего предка v в T для листьев, содержащих p_i и p_j, и определяем $\tau(p_i, p_j) = h_v$.

Иерархическая метрика τ называется *согласованной* с функцией расстояния d, если для всех пар i, j выполняется условие $\tau(p_i, p_j) \leq d(p_i, p_j)$.

Предложите алгоритм с полиномиальным временем, который получает функцию расстояния d и выдает иерархическую метрику τ со следующими свойствами:

(i) метрика τ согласована с d, и

(ii) если τ' — любая другая иерархическая метрика, согласованная с d, то $\tau'(p_i, p_j) \leq \tau(p_i, p_j)$ для каждой пары точек p_i и p_j.

26. Одна из первых задач, рассматриваемых в курсе математического анализа, — задача минимизации дифференцируемой функции (например, $y = ax^2 + bx + c$ для $a > 0$). С другой стороны, задача нахождения минимального остовного дерева относится к совершенно иной разновидности задач минимизации: количество вариантов для достижения минимума ограничеуно, но вычисления должны проводиться без перебора этого (огромного) конечного числа возможностей.

    Что произойдет, если свести воедино эти две разновидности минимизации? Ниже приведен пример задачи такого рода. Допустим, имеется связный граф $G = (V, E)$; *переменная стоимость* каждого ребра изменяется во времени в соответствии с функцией f_e: $\mathbf{R} \rightarrow \mathbf{R}$. Таким образом, в момент времени t ребро имеет стоимость $f_e(t)$. Предполагается, что все функции принимают положительные значения на всем интервале значений. Заметьте, что множество ребер, образующих минимальное остовное дерево графа G, может изменяться со временем. Конечно, стоимость минимального остовного дерева графа G также становится функцией времени t; обозначим эту функцию $c_G(t)$. Возникает естественная задача: найти значение t, при котором минимизируется $cG(t)$.

    Предположим, каждая функция f_e определяется многочленом степени 2: $f_e(t) = a_e t^2 + b_e t + c_e$, где $a_e > 0$. Предложите алгоритм, который получает граф G и набор значений $\{(a_e, b_e, c_e) : e \in E\}$ и возвращает значение времени t, при котором минимальное остовное дерево имеет минимальную стоимость. Алгоритм должен выполняться за полиномиальное время в зависимости от количества узлов и ребер графа G. Считайте, что арифметические операции с числами $\{(a_e, b_e, c_e)\}$ выполняются за постоянное время.

27. В ходе анализа комбинаторной структуры остовных деревьев можно рассмотреть пространство *всех* возможных остовных деревьев заданного графа и изучить свойства этого пространства. Эта стратегия также применяется во многих похожих задачах.

    Один из возможных подходов выглядит так. Пусть G — связный граф, а T и T' — два разных остовных дерева графа i. Деревья T и T' называются соседями, если T содержит ровно одно ребро, не входящее в T', а T' содержит ровно одно ребро, не входящее в T.

    Теперь на базе любого графа G строится (большой) граф H. Узлами H являются остовные деревья G, а ребро между двумя узлами H существует в том случае, если соответствующие остовные деревья являются соседями. Правда ли, что для любого связного графа G полученный граф H является связным? Приведите доказательство того, что граф H всегда связный, или представьте контрпример (с объяснением) связного графа G, для которого граф H не является связным.

28. Предположим, вы работаете на сетевую компанию CluNet, и вам предложена следующая задача. Сеть, над которой в настоящий момент работает компания, моделируется связным графом $G = (V, E)$ с n узлами. Каждое ребро e представляет оптоволоконный кабель, арендуемый CluNet у его владельцев — компаний X и Y. Ваши клиенты хотят выбрать остовное дерево T графа G и обновить каналы, соответствующие ребрам T. Специалисты по деловым отношениям уже заключили с компаниями X и Y договор, в котором указано число k. При выборе дерева T k из его ребер будут принадлежать X, а n-k-1 ребер будут принадлежать Y.

    Теперь руководство CluNet столкнулось с проблемой: ему совершенно неочевидно, существует ли остовное дерево T, удовлетворяющее таким критериям, и как найти его, если оно существует. Перед вами поставлена задача: найти алгоритм с полиномиальным временем, который получает граф G с ребрами, помеченными X или Y, и либо (i) возвращает остовное дерево, в котором ровно k ребер помечено X, либо (ii) обоснованно сообщает о том, что такое дерево не существует.

29. Имеется список из n натуральных чисел $d_1, d_2, ..., d_n$. Покажите, как за полиномиальное время решить, существует ли ненаправленный граф $G = (V, E)$, степени узлов которого равны $d_1, d_2, ..., d_n$. (То есть если $V = \{v_1, v_2, ..., v_n\}$, степень v_i должна быть равна d_i.) Граф G не может содержать нескольких ребер между одной парой узлов, или «циклических» ребер с одинаковыми конечными точками.

30. Граф $G = (V, E)$ содержит n узлов; каждая пара узлов соединяется ребром. Каждому ребру (i, j) присвоен положительный вес w_{ij}; значения весов соответствуют *неравенству треугольника* $w_{ik} \leq w_{ij} + w_{jk}$. Для подмножества $V' \subseteq V$ подграф (с весами ребер), определяемый множеством узлов V', обозначается $G[V']$.

    Имеется множество $X \subseteq V$ из k точек, которые должны быть соединены ребрами. *Деревом Штейнера* для X называется такое множество Z, что $X \subseteq Z \subseteq V$, в сочетании с остовным поддеревом T подграфа $G[Z]$. *Весом* дерева Штейнера называется вес дерева T.

    Покажите, что задача нахождения дерева Штейнера с минимальным весом для X может быть решена за время $O(n^{O(k)})$.

31. Вернемся к исходной формулировке задачи нахождения минимального остовного дерева: имеется связный ненаправленный граф $G = (V, E)$ с положительными длинами ребер $\{\ell_e\}$; нужно найти для него остовной подграф. Теперь допустим, что нас интересует подграф $H = (V, F)$ с большей «плотностью», чем у дерева, и с гарантиями того, что для каждой пары вершин $u, v \in V$ длина кратчайшего пути u-v в H не намного больше длины кратчайшего пути u-v в G. Под «*длиной* пути P» подразумевается сумма ℓ_e по всем ребрам e в P.

    Следующая разновидность алгоритма Крускала строит такой подграф:

    - Ребра сортируются в порядке возрастания длин. (Предполагается, что все длины ребер различны.)
    - Затем строится подграф $H = (V, F)$ с последовательным рассмотрением всех ребер.

- При переходе к ребру $e = (u, v)$ ребро e включается в подграф H, если в настоящее время в H не существует пути u-v. (Алгоритм Крускала делал бы то же самое.) С другой стороны, если путь u-v в H существует, обозначим d_{uv} кратчайший из таких путей (как и прежде, длина определяется значениями $\{\ell_e\}$). Ребро e добавляется в H, если $3\ell_e < d_{uv}$.

Иначе говоря, ребро добавляется даже в том случае, если u и v уже находятся в одной компоненте связности, — при условии, что добавление ребра в достаточной степени сокращает их расстояние кратчайшего пути.

Пусть $H = (V, F)$ — подграф G, возвращенный алгоритмом.

(a) Докажите, что для каждой пары узлов $u, v \in V$ длина кратчайшего пути u-v в H не более чем в три раза превышает длину кратчайшего пути u-v в G.

(b) Несмотря на приближенное сохранение расстояний кратчайших путей, подграф H, произведенный алгоритмом, не должен быть слишком плотным. Пусть $f(n)$ — максимальное количество ребер, которые могут быть получены на выходе алгоритма для всех входных графов из n узлов с длинами ребер. Докажите, что

$$\lim_{n\to\infty} \frac{f(n)}{n^2} = 0.$$

32. Рассмотрим направленный граф $G = (V, E)$ с корнем $r \in V$ и неотрицательными стоимостями ребер. В этой задаче рассматриваются разновидности алгоритма ориентированного дерева с минимальной стоимостью.

(a) Алгоритм, рассмотренный в разделе 4.9, работает следующим образом: мы изменяем стоимости, рассматриваем подграф с ребрами нулевой стоимости, ищем направленный цикл в этом подграфе и сжимаем его (если такой цикл существует). Возможно ли вместо циклов найти и сжать сильные компоненты подграфа? Приведите краткое обоснование.

(b) В алгоритме мы определяли величину y_v как минимальную стоимость ребра, входящего в v, и изменяли стоимости всех ребер e, входящих в узел v, по формуле $c'_e = c_e - y_v$. Допустим, вместо этого используется следующая модифицированная стоимость: $c''_e = \max(0, c_e - 2y_v)$. Вероятно, в результате этого изменения у большего количества ребер стоимость будет равна 0. Теперь предположим, что мы нашли ориентированное дерево T со стоимостью 0. Докажите, что стоимость T не более чем вдвое превышает стоимость ориентированного дерева с минимальной стоимостью в исходном графе.

(c) Предположим, что ориентированное дерево со стоимостью 0 не найдено. Сожмите все сильные компоненты со стоимостью 0 и рекурсивно примените ту же процедуру к полученному графу, пока не будет найдено ориентированное дерево. Докажите, что стоимость T не более чем вдвое превышает стоимость ориентированного дерева с минимальной стоимостью в исходном графе.

33. Имеется направленный граф $G = (V, E)$, в котором стоимость каждого ребра равна либо 0, либо 1. Предположим, в G имеется такой узел r, что существует путь от r к любому другому узлу G. Также дано целое число k. Приведите алгоритм с полиномиальным временем выполнения, который либо строит ориентированное дерево с корнем r, стоимость которого равна точно k, либо обоснованно сообщает о том, что такое дерево не существует.

Примечания и дополнительная литература

Благодаря своей концептуальной четкости и интуитивной привлекательности, жадные алгоритмы изучаются уже давно и нашли множество применений в компьютерных науках. В этой главе основное внимание уделялось ситуациям, для которых жадный алгоритм находит оптимальное решение. Жадные алгоритмы также часто используются в качестве простых эвристик даже тогда, когда оптимальность решения не гарантирована. В главе 11 рассматриваются жадные алгоритмы, которые находят решения, близкие к оптимальным.

Как упоминалось в главе 1, задача интервального планирования может рассматриваться как частный случай задачи поиска независимого множества для графа, представляющего перекрытия в наборе интервалов. Графы, которые встречаются в таких задачах, называются *интервальными графами* и были тщательно изучены; за информацией, например, обращайтесь к книге Голамбика (Golumbic, 1980). Не только задача поиска независимого множества, но и многие другие вычислительные задачи существенно упрощаются для особого случая интервальных графов.

Задача интервального планирования и задача планирования для минимизации максимальной задержки — два представителя категории базовых задач планирования, для которых простой жадный алгоритм выдает оптимальное решение. Описания многих других взаимосвязанных задач можно найти в работе Лоулера, Ленстры, Ринноя Кана и Шмойса (Lawler, Lenstra, Rinnoy Kan and Shmoys, 1993).

Автором оптимального алгоритма кэширования и его анализа является Белади (Belady, 1966). Как упоминалось в тексте, в реальных условиях алгоритмы кэширования должны применять решения по вытеснению в реальном времени, без информации о будущих запросах. Такие стратегии кэширования будут рассматриваться в главе 13.

Алгоритм нахождения кратчайших путей в графе с неотрицательными длинами ребер предложил Дейкстра (Dijkstra, 1959). Обзоры методов решения задачи минимального остовного дерева (вместе с историческим обзором) можно найти в статьях Грэма и Хелла (Graham, Hell, 1985) и Несетрила (Nesetril, 1997).

Алгоритм одиночной связи является одним из самых распространенных подходов к общей задаче кластеризации; в книгах Андерберга (Anderberg, 1973), Дуды, Харта и Сторка (Duda, Hart, Stork, 2001), а также Джейна и Дьюбса (Jain, Dubes, 1981) представлены различные методы кластеризации.

Алгоритм построения оптимальных префиксных кодов разработан Хаффманом (Huffman, 1952); более ранние методы, упоминаемые здесь, встречаются в книгах

Фано (Fano, 1949), а также Шеннона и Уивера (Shannon, Weaver, 1949). Общий обзор методов сжатия данных приведен в книге Белла, Клири и Уиттен (Bell, Cleary, Witten, 1990), а также в статье Лелевера и Хиршберга (Lelewer, Hirschberg, 1987). В более общем смысле эта тема относится к *теории передачи информации*, занимающейся методами представления и кодирования цифровых данных. Одной из фундаментальных работ в этой области является книга Шеннона и Уивера (Shannon, Weaver, 1949); в более позднем учебнике Кавера и Томаса (Cover, Thomas, 1991) приводится подробная информация по теме.

Авторами алгоритма нахождения ориентированного дерева с минимальной стоимостью обычно называют Чу и Лю (Chu, Liu, 1965) и Эдмондсу (Edmonds, 1967), разработавших его независимо друг от друга. Как упоминалось в главе, этот многофазный метод расширяет наши представления о том, что собой представляет жадный алгоритм. Он также важен с точки зрения линейного программирования, поскольку в этом контексте может рассматриваться как фундаментальное практическое применение метода ценообразования, или *прямо-двойственного метода* (primal-dual technique), при разработке алгоритмов. Книга Немхаузера и Волси (Nemhauser, Wolsey, 1988) развивает эти связи с линейным программированием. Метод будет описан в главе 11 в контексте аппроксимирующих алгоритмов.

Как было сказано в начале главы, трудно дать точное определение того, что же следует считать жадным алгоритмом. В поисках такого определения даже непонятно, можно ли провести аналогию со знаменитым критерием непристойности судьи Стюарта из Верховного суда США («Когда я её вижу, я знаю, что это такое»), поскольку в научном сообществе нет единой точки зрения (и даже интуитивных представлений) относительно границы между жадными и нежадными алгоритмами. Проводились исследования, направленные на формализацию классов жадных алгоритмов: из примеров можно упомянуть влиятельную теорию матроидов (Edmonds, 1971; Lawer, 2001). В статье Бородина, Нильсена и Ракоффа (Borodin, Nielsen, Rackoff, 2002) формализуются понятия жадных алгоритмов и алгоритмов «жадного типа», а также приводятся сравнения с другими теоретическими работами по данному вопросу.

Примечания к упражнениям

Упражнение 24 основано на результатах, авторами которых были М. Эдахиро, Т. Чао, И. Сюй, Дж. Хо, К. Безе и А. Канг; упражнение 31 основано на результатах, авторами которых были Инго Альтхофер, Гаутам Дас, Дэвид Добкин и Дебора Джозеф.

Глава 5
Разделяй и властвуй

Термином «разделяй и властвуй» обозначается класс алгоритмических методов, которые разбивают входные данные на несколько частей, рекурсивно решают задачу для каждой части, а затем объединяют решения подзадач в одно общее решение. Во многих случаях такие решения оказываются весьма простыми и эффективными.

Анализ времени выполнения алгоритма категории «разделяй и властвуй» обычно подразумевает вычисление *рекуррентного соотношения*, которое устанавливает рекурсивную границу времени выполнения в контексте времени выполнения меньших экземпляров. Глава открывается общим обсуждением рекуррентных отношений; вы увидите, как они встречаются в процессе анализа и какие методы используются для установления верхних границ на их основе.

Затем будет описано использование принципа «разделяй и властвуй» в разных предметных областях: вычисление функции расстояния для разных вариантов ранжирования множества объектов; поиск ближайшей пары точек на плоскости; умножение двух целых чисел; и сглаживание сигнала с шумом. Принцип «разделяй и властвуй» также встречается в последующих главах, поскольку этот метод часто хорошо работает в сочетании с другими методами разработки алгоритмов. Например, в главе 6 он будет применен в сочетании с динамическим программированием для создания эффективного по затратам памяти решения фундаментальной задачи сравнения последовательностей, а в главе 13 он используется в сочетании с рандомизацией для создания простого и эффективного алгоритма вычисления медианы для множества чисел.

У многих ситуаций, в которых применяется принцип «разделяй и властвуй» (включая перечисленные), есть нечто общее: естественный алгоритм «грубой силы» может выполняться за полиномиальное время, а стратегия «разделяй и властвуй» способна сократить время выполнения до многочлена меньшей степени. В этом отношении он отличается от многих задач из предыдущих глав, например от задач, в которых алгоритм «грубой силы» работал с экспоненциальным временем, а целью разработки более сложного алгоритма был переход к полиномиальному времени. Например, как упоминалось в главе 2, естественный алгоритм «грубой силы» для поиска ближайшей пары среди n точек на плоскости просто проверяет все $\Theta(n^2)$ расстояний с (полиномиальным) временем выполнения $\Theta(n^2)$. Принцип «разделяй и властвуй» позволит улучшить время выполнения до $O(n \log n)$. Таким образом, на высоком уровне общая тема этой главы не отличается от того, что мы

уже видели ранее: улучшение результатов поиска «грубой силой» становится фундаментальной концептуальной проблемой на пути эффективного решения задачи, и проектирование сложного алгоритма помогает с ней справиться. Просто различия между поиском методом «грубой силы» и улучшенным решением не всегда означают различия между экспоненциальным и полиномиальным временем.

5.1. Первое рекуррентное отношение: алгоритм сортировки слиянием

Чтобы понять общий подход к анализу алгоритмов «разделяй и властвуй», начнем с алгоритма сортировки слиянием. Этот алгоритм уже упоминался в главе 2, когда мы рассматривали типичное время выполнения некоторых алгоритмов. Алгоритм сортировки слиянием сортирует список чисел: сначала список делится на две половины, каждая половина рекурсивно сортируется по отдельности, после чего результаты рекурсивных вызовов (в форме двух отсортированных половин) объединяются алгоритмом слияния отсортированных списков, представленным в главе 2.

Чтобы проанализировать время выполнения сортировки слиянием, мы абстрагируем поведение алгоритма в следующий шаблон, описывающий многие типичные алгоритмы «разделяй и властвуй».

(†) Разбить входные данные на два блока равного размера; рекурсивно решить две подзадачи для этих блоков по отдельности; объединить два результата в одно решение, с линейными затратами времени для исходного деления и итогового объединения.

В алгоритме сортировки слиянием, как и в любом алгоритме этого типа, также необходим базовый случай рекурсии, который обычно завершает выполнение для входных данных некоторого постоянного размера. В случае сортировки слиянием предполагается, что при достижении размера 2 рекурсия останавливается, а два элемента сортируются простым сравнением.

Рассмотрим любой алгоритм, построенный по шаблону (†); обозначим $T(n)$ худшее время выполнения для входных данных с размером n. Если предположить, что n четно, алгоритм за время $O(n)$ делит входные данные на два блока с размером $n/2$, а затем тратит время $T(n/2)$ на решение каждой производной задачи ($T(n/2)$ — худшее время выполнения для входных данных с размером $n/2$); и наконец, время $O(n)$ тратится на объединение решений из двух рекурсивных вызовов. Это время выполнения $T(n)$ удовлетворяет следующему *рекуррентному отношению*.

(5.1) Для некоторой константы c,

$$T(n) \leq 2T(n/2) + cn$$

для $n > 2$, и

$$T(2) \leq c.$$

Структура (5.1) типична для рекуррентных отношений: имеется неравенство или равенство, ограничивающее $T(n)$ в выражении, использующем $T(k)$ для меньших значений k, и существует базовый случай, который фактически сообщает, что $T(n)$ является константой для константы n. Также заметьте, что (5.1) можно записать неформально в виде $T(n) \leq 2T(n/2) + O(n)$, исключив константу c. Тем не менее явное включение c обычно бывает полезно при анализе рекуррентных отношений.

Чтобы упростить описание, мы будем считать, что параметры (такие, как n) четные. Такое предположение создает некоторую неточность; без него два рекурсивных вызова будут применяться к задачам размера $\lceil n/2 \rceil$, а рекуррентное отношение должно иметь вид

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + cn$$

для $n \geq 2$. Однако для всех рекуррентных отношений, которые мы будем рассматривать (и которые встречаются на практике), игнорирование верхней или нижней границы чисел обычно не влияет на асимптотические границы, а формулы заметно упрощаются.

Учтите, что (5.1) не дает явной асимптотической границы для скорости роста функции T, скорее $T(n)$ задается неявно в контексте его значений для входных данных меньшего размера. Для получения явной границы необходимо разрешить рекуррентное отношение так, чтобы обозначения T присутствовали только в левой части неравенства.

Задача разрешения рекуррентности была встроена в некоторые стандартные компьютерные алгебраические системы, а многие стандартные рекуррентные отношения успешно разрешаются автоматизированными средствами. Тем не менее будет полезно разобраться в процессе разрешения рекуррентности и научиться распознавать рекуррентные отношения, приводящие к хорошему времени выполнения, — разработка эффективных алгоритмов «разделяй и властвуй» тесно связана с пониманием того, как рекуррентное отношение определяет время выполнения.

Методы разрешения рекуррентности

Существуют два основных подхода к разрешению рекуррентности.

Самый естественный и понятный подход к поиску решения — «раскрутка» рекурсии с отслеживанием времени выполнения на нескольких начальных уровнях и выявлением закономерности, которая сохраняется в ходе рекурсии. Затем время выполнения суммируется по всем уровням рекурсии (то есть до достижения нижнего предела для подзадач постоянного размера) с получением общего времени выполнения.

Во втором варианте строится гипотеза, которая подставляется в рекуррентное отношение и проверяется на жизнеспособность. Для формальной проверки таких подстановок используется индукция по n. Существует полезная разновидность

этого метода, в которой известна общая форма решения, но неизвестны точные значения всех параметров. Оставляя эти параметры неопределенными в подстановке, часто удается подобрать их по мере необходимости.

А теперь мы обсудим каждый из этих методов на примере рекуррентного отношения (5.1).

Раскрутка рекуррентности в алгоритме сортировки слиянием

Начнем с первого способа разрешения рекуррентности в (5.1). Основная идея представлена на рис. 5.1.

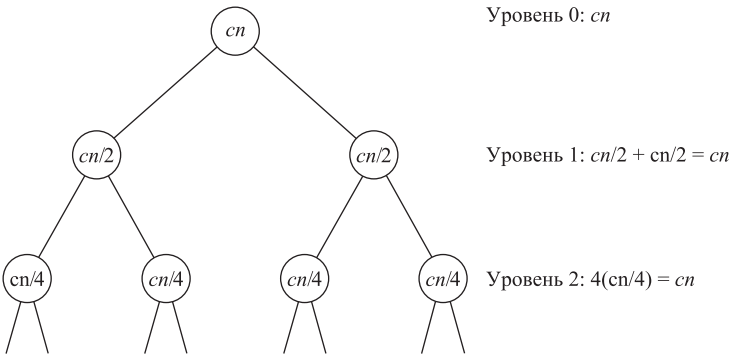


Рис. 5.1. Раскрутка рекуррентности $T(n) \le 2T(n/2) + O(n)$

- *Анализ нескольких начальных уровней*: на первом уровне рекурсии имеется одна задача с размером n, которая выполняется за время cn плюс время, потраченное на все последующие рекурсивные вызовы. На следующем уровне имеются две задачи с размером $n/2$. Каждая задача выполняется за время максимум $cn/2$, что дает в сумме cn плюс время последующих рекурсивных вызовов. На третьем уровне используются четыре задачи с размером $n/4$, каждая выполняется за время $cn/4$, итого максимум cn.
- *Выявление закономерности*: что вообще происходит? На уровне j рекурсии количество подзадач удваивается j раз и поэтому равно 2^j. Каждая задача соответственно уменьшается вдвое j раз; следовательно, размер каждой подзадачи равен $n/2^j$, и она выполняется за время максимум $cn/2^j$. Таким образом, вклад уровня j в общее время выполнения составляет максимум $2^j(cn/2^j) = cn$.
- *Суммирование по всем уровням рекурсии*: выясняется, что у рекурсии из (5.1) общий объем работы, выполняемой на каждом уровне, имеет одну и ту же верхнюю границу cn. Количество делений входных данных для сокращения их размера с n до 2 составляет $\log_2 n$. Суммируя объем работы cn по $\log n$ уровням рекурсии, получаем общее время выполнения $O(n \log n)$.

Следующее утверждение подводит итог анализа.

(5.2) Любая функция $T(\cdot)$, удовлетворяющая условию (5.1), имеет границу $O(n \log n)$ при $n > 1$.

Подстановка решения в рекуррентное отношение сортировки слиянием

Аргументы, которые позволили прийти к выводу (5.2), могут использоваться для определения того, что функция $T(n)$ ограничивается $O(n \log n)$. С другой стороны, если у вас имеется гипотеза относительно времени выполнения, ее можно проверить, подставив ее в рекуррентное отношение.

Допустим, вы предполагаете, что $T(n) \leq cn \log_2 n$ для всех $n \geq 2$, и хотите проверить, так ли это. Для $n = 2$ оценка очевидно верна, так как в этом случае $cn \log_2 n = 2c$, а в (5.1) явно указано, что $T(2) \leq c$. Теперь предположим по индукции, что $T(m) \leq cm \log_2 m$ для всех значений m, меньших n; теперь нужно доказать, что гипотеза выполняется для $T(n)$. Для этого запишем рекуррентное отношение для $T(n)$ и подставим его в неравенство $T(n/2) \leq c(n/2) \log_2(n/2)$. Полученное выражение можно упростить, если заметить, что $\log_2(n/2) = (\log_2 n) - 1$. Ниже приведены полные вычисления.

$$T(n) \leq 2T(n/2) + cn$$

$$\leq 2c(n/2) \log_2(n/2) + cn$$

$$= cn[(\log_{2\,n}) - 1] + cn$$

$$= (cn \log_{2\,n}) - cn + cn$$

$$= cn \log_{2\,n}.$$

Искомая граница для $T(n)$ доказана в предположении, что она действительна для меньших значений $m < n$; рассуждение по индукции завершено.

Использование частичной подстановки

Также существует более слабый вид подстановки, в котором предполагается общая форма решения без точных значений всех констант и других параметров.

Предположим, вы считаете, что $T(n) = O(n \log n)$, но не уверены в константе в обозначении $O(\cdot)$. Метод подстановки может использоваться даже без точного значения этой константы. Сначала записываем $T(n) \leq kn \log_b n$ для некоторой константы k и основания b, которые будут определены позднее.

(Вообще говоря, основание и константа связаны друг с другом, так как основание логарифма можно изменить простым умножением на константу, — см. главу 2.)

Теперь хотелось бы знать, существуют ли значения k и b, которые будут работать в индукции. Для начала проверим один уровень индукции.

$$T(n) \le 2T(n/2) + cn \le 2k(n/2)\log_b(n/2) + cn$$

Появляется естественная мысль выбрать для логарифма основание $b = 2$, поскольку мы видим, что это позволит применить упрощение $\log_2(n/2) = (\log_2 n) - 1$. Продолжая с выбранным значением, получаем

$$T(n) \le 2k(n/2)\log_2(n/2) + cn$$

$$= 2k(n/2)[(\log_2 n) - 1] + cn$$

$$= kn[(\log_2 n) - 1] + cn$$

$$= (kn\log_2 n) - kn + cn.$$

Наконец, спросим себя: можно ли подобрать значение k, при котором последнее выражение будет ограничиваться $kn\log_2 n$? Очевидно, что ответ на этот вопрос положителен; просто нужно выбрать любое значение k, не меньшее c, и мы получаем

$$T(n) \le (kn\log_2 n) - kn + cn \le kn\log_2 n.$$

Индукция завершена.

Итак, метод подстановки может пригодиться для определения точных значений констант, если у вас уже имеется некоторое представление об общей форме решения.

5.2. Другие рекуррентные отношения

В предыдущем разделе в ходе решения задачи было получено рекуррентное отношение (5.1), которое еще встретится при разработки нескольких алгоритмов типа «разделяй и властвуй» позднее в этой главе. Для дальнейшего исследования этой темы рассмотрим класс рекуррентных отношений, который обобщает (5.1), и посмотрим, как разрешаются рекуррентности из этого класса. Другие представители класса еще встретятся при разработке алгоритмов и в этой, и в последующих главах.

Этот более общий класс алгоритмов образуется при рассмотрении алгоритмов «разделяй и властвуй», которые создают рекурсивные вызовы для q подзадач с размером $n/2$ каждая, а затем объединяют результаты за время $O(n)$. Это соответствует рекуррентному отношению сортировки слиянием (5.1) с $q = 2$ рекурсивных вызовов, или всего одним ($q = 1$) рекурсивным вызовом. Случай $q > 2$ встретится позднее в этой главе, когда мы займемся разработкой алгоритмов для целочисленного умножения; разновидность с $q = 1$ будет представлена позднее, когда мы будем разрабатывать рандомизированный алгоритм нахождения медианы в главе 13.

Если $T(n)$ обозначает время выполнения алгоритма, построенного по такому принципу, то $T(n)$ подчиняется следующему рекуррентному отношению, которое напрямую обобщает (5.1) с заменой 2 на q:

(5.3) Для некоторой константы c,

$$T(n) \le qT(n/2) + cn$$

для $n>2$, и

$$T(2) \le c$$

В следующем разделе описано, как решить (5.3) с использованием уже применявшихся методов: раскрутки, подстановки и частичной подстановки. Случаи $q > 2$ и $q = 1$ рассматриваются по отдельности, так как они качественно отличаются друг от друга, а также от случая $q = 2$.

Случай $q > 2$ подзадач

Начнем с раскрутки (5.3) для случая $q > 2$ по схеме, использованной ранее для (5.1). Как вы увидите, результат сильно отличается.

- *Анализ нескольких начальных уровней*: пример приведен для случая $q = 3$ на рис. 5.2. На первом уровне рекурсии имеется одна задача с размером n, которая выполняется за время cn плюс время, потраченное на все последующие рекурсивные вызовы. На следующем уровне имеются q задач с размером $n/2$. Каждая задача выполняется за время максимум $cn/2$, что дает в сумме $(q/2)$ cn плюс время последующих рекурсивных вызовов. На следующем уровне используются q^2 задач с размером $n/4$ с общим временем $(q^2/4)cn$. Мы видим, что при $q > 2$ общий объем работы на уровень *возрастает* с увеличением уровня.

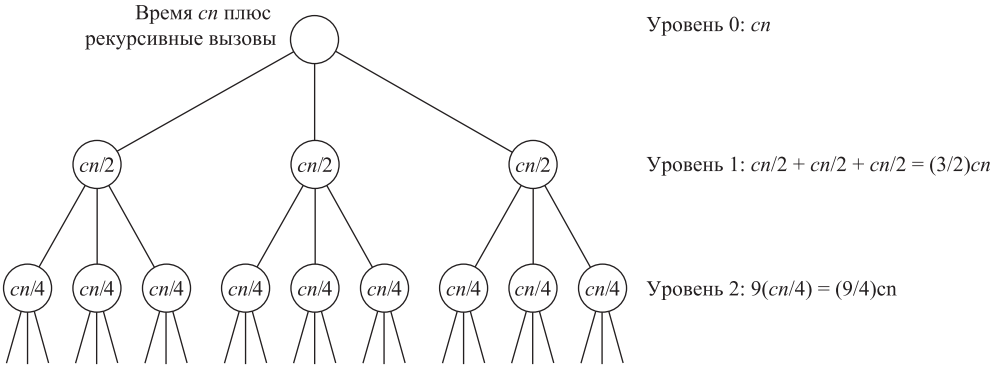


Рис. 5.2. Раскрутка рекуррентности $T(n) \le 3T(n/2) + O(n)$

- *Выявление закономерности*: на произвольном уровне j рекурсии задействованы q^j разных экземпляров, каждый из которых имеет размер $n/2^j$. Следовательно, общий объем работы, выполняемой на уровне j, составляет $q^j(cn/2^j) = (q/2)^j cn$.
- *Суммирование по всем уровням рекурсии*: как и в предыдущем случае, существуют $\log_2 n$ уровней рекурсии, а общий объем выполняемой работы определяется следующей суммой:

$$T(n) \le \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right) cn = cn \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j .$$

Это геометрическая сумма, состоящая из степеней $r = q/2$. Воспользовавшись формулой геометрической суммы при $r > 1$, получаем

$$T(n) \le cn\left(\frac{r\log_2 n - 1}{r-1}\right) \le cn\left(\frac{r\log_2 n}{r-1}\right).$$

Так как нас интересует асимптотическая верхняя граница, будет полезно выделить то, что является константой: последнее выражение может быть записано в виде

$$T(n) \le cn\left(\frac{c}{r-1}\right) nr^{\log_2 n}.$$

Наконец, нужно разобраться в том, что же собой представляет $r^{\log_2 n}$. Воспользуемся очень удобной формулой: для всех $a > 1$ и $b > 1$ справедливо $a^{\log b} = b^{\log a}$. Следовательно,

$$r^{\log_2 n} = n^{\log_2 r} = n^{\log_2 (q/2)} = n^{(\log_2 q)-1}.$$

В итоге получаем

$$T(n) \le \left(\frac{c}{r-1}\right) n \cdot n^{(\log_2 q)-1} \le \left(\frac{c}{r-1}\right) n^{\log_2 q} = O\left(n^{\log_2 q}\right).$$

Результат можно обобщить в следующей форме:

(5.4) Любая функция $T(\cdot)$, удовлетворяющая условию (5.3) с $q > 2$, имеет границу $O(n^{\log_2 q})$.

Получается, что время выполнения больше линейного, так как $\log_2 q > 1$, но все равно полиномиальное по n. При подстановке конкретных значений q время выполнения составит $O(n^{\log_2 3}) = O(n^{1,59})$ для $q = 3$; а для $q = 4$ оно равно $O(n^{\log_2 4}) = O(n^2)$. Увеличение времени выполнения с ростом q выглядит логично, поскольку рекурсивные вызовы порождают больший объем работы с увеличением q.

Частичная подстановка

Появление $\log_2 q$ в показателе степени естественно следовало из решения (5.3), но такое выражение далеко не всегда можно предположить в исходной ситуации. Посмотрим, как подход, основанный на частичной подстановке, позволяет обнаружить эту экспоненту другим способом.

Допустим, мы предполагаем, что решение (5.3) при $q > 2$ имеет форму $T(n) \le kn^d$ для некоторых констант $k > 0$ и $d > 1$. Это достаточно общее предположение, так как мы даже не пытаемся задать показатель степени d. Теперь начнем рассуждения методом индукции и посмотрим, какие ограничения потребуются для k и d. Имеем

$$T(n) \le qT(n/2).$$

В результате применения индукционной гипотезы к $T(n/2)$ это выражение развертывается в

$$T(n) \le qk\left(\frac{n}{2}\right)^d + cn = \frac{q}{2^d} kn^d + cn.$$

Результат получился поразительно близким: если выбрать d так, чтобы $q/2^d = 1$, то получится $T(n) \le kn^d + cn$ — почти то, что требуется, не считая лишнего члена cn. Итак, нужно разобраться с двумя проблемами: как выбрать d, чтобы $q/2^d = 1$, и как избавиться от cn.

С выбором d все просто: нужно, чтобы $2^d = q$, поэтому $d = \log_2 q$. Как видите, показатель степени $\log_2 q$ совершенно естественно появляется, когда мы решаем определить, какое значение d будет работать при подстановке в рекуррентное отношение.

Но еще нужно избавиться от члена cn. Для этого мы изменим форму предположения для $T(n)$ так, чтобы явно устранить его вычитанием. Попробуем проверить форму $T(n) \le kn^d - \ell n$, в которой мы решили, что $d = \log_2 q$, но не зафиксировали константы k или ℓ. При применении новой формулы к $T(n/2)$ получаем

$$\begin{aligned} T(n) &\le qk\left(\frac{n}{2}\right)^d - q\ell\frac{n}{2} + cn \\ &= \frac{q}{2^d} kn^d - \frac{q\ell}{2} n + cn \\ &= kn^d - \frac{q\ell}{2} n + cn \\ &= kn^d - \left(\frac{q\ell}{2} - c\right) n. \end{aligned}$$

И теперь все полностью работает, если выбрать ℓ так, чтобы $\left(\frac{q\ell}{2} - c\right) = \ell$: другими словами, $\ell = 2c/(q-2)$. На этом шаг индукции для n завершается. Также необходимо обработать базовый случай $n = 2$, а для этого можно воспользоваться тем фактом, что значение k еще не зафиксировано: выберем k достаточно большим, чтобы формула определяла действительную верхнюю границу для случая $n = 2$.

Случай одной подзадачи

Рассмотрим случай $q = 1$ в (5.3), который хорошо демонстрирует еще одну разновидность алгоритма. Хотя в этой главе пример применения рекуррентности для $q = 1$ не встречается, он будет представлен в главе 13, как упоминалось ранее.

Для начала попробуем построить решение методом раскрутки рекуррентного отношения.

- *Анализ нескольких начальных уровней*: начальные уровни рекурсии изображены на рис. 5.3. На первом уровне рекурсии имеется одна задача с размером n, которая выполняется за время cn плюс время, потраченное на все последующие рекурсивные вызовы. На следующем уровне имеется одна задача с размером $n/2$, которая вносит вклад $cn/2$; далее идет уровень с одной задачей с размером $n/4$, добавляющая время $cn/4$. Как видите, в отличие от предыдущего случая общий объем работы при $q = 1$ сокращается с ростом уровня рекурсии.

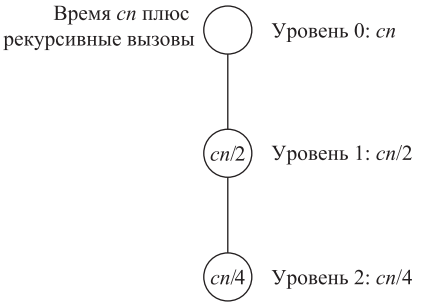


Рис. 5.3. Раскрутка рекуррентности $T(n) \le T(n/2) + O(n)$

- *Выявление закономерности*: на произвольном уровне j по-прежнему остается одна подзадача с размером $n/2^j$. Ее вклад в общее время выполнения составляет $cn/2^j$.
- *Суммирование по всем уровням рекурсии*: рекурсия имеет $\log_2 n$ уровней, а общий объем выполняемой работы определяется следующей суммой:

$$T(n) \le \sum_{j=0}^{\log_2 n - 1} \frac{cn}{2^j} = cn \sum_{j=0}^{\log_2 n - 1} \left(\frac{1}{2^j}\right).$$

Эта геометрическая сумма очень легко вычисляется; даже если продолжить ее до бесконечности, она будет стремиться к 2. Следовательно, выполняется отношение

$$T(n) \le 2cn = O(n).$$

Подведем итог:

(5.5) Любая функция T(·), удовлетворяющая условию (5.3) с $q = 1$, имеет границу $O(n)$.

На первый взгляд этот результат выглядит противоестественно. Алгоритм выполняет $\log n$ уровней рекурсии, но общее время выполнения остается линейным по n. Суть в том, что геометрическая прогрессия с убывающей экспонентой весьма эффективна: половина работы, выполняемой алгоритмом, выполняется на верхнем уровне рекурсии.

Также стоит заметить, что частичная подстановка в рекурсии очень хорошо работает в данном случае. Предположим, как и ранее, что решение задается в фор-

ме $T(n) \le kn^d$. Попробуем установить это посредством индукции с использованием (5.3), предполагая, что решение работает для меньшего значения $n/2$:

$$\begin{aligned} T(n) &\le T(n/2) + cn \\ &\le k\left(\frac{n}{2}\right)^d + cn \\ &= \frac{k}{2^d} n^d + cn. \end{aligned}$$

Если теперь просто выбрать $d = 1$ и $k = 2c$, получаем

$$T(n) \le \frac{k}{2} n + cn = \left(\frac{k}{2} + c\right) n = kn,$$

с завершением индукции.

Действие параметра *q*

Стоит подробнее изучить роль параметра q в классе рекуррентных отношений $T(n) \le qT(n/2) + O(n)$, определяемых (5.3). С $q = 1$ время выполнения получается линейным; с $q = 2$ оно равно $O(n \log n)$; при $q > 2$ достигается полиномиальная граница с показателем степени больше 1, возрастающим с увеличением q. Причина таких различий во времени выполнения связана с тем, на какой стадии выполняется большая часть работы в этой рекурсии: при $q = 1$ в общем времени выполнения преобладает верхний уровень, тогда как при $q > 2$ основная работа выполняется подзадачами постоянного размера на нижних уровнях. Рассматривая происходящее с этой точки зрения, мы видим, что рекуррентное отношение для $q = 2$ в действительности представляет «баланс»: объем работы, выполняемой на каждом уровне, одинаков для всех уровней, и именно этот фактор обеспечивает время выполнения $O(n \log n)$.

Похожее рекуррентное отношение: $T(n) \le 2T(n/2) + O(n^2)$

В завершение темы рассмотрим последнее рекуррентное отношение; оно поучительно и как пример убывающей геометрической суммы, и как интересный контраст с рекуррентным отношением (5.1), которым характеризовалась сортировка слиянием. Более того, его разновидность будет рассматриваться в главе 6, когда мы займемся анализом алгоритма «разделяй и властвуй» для решения задачи выравнивания последовательностей с малыми затратами памяти.

Рекуррентное отношение базируется на следующей структуре «разделяй и властвуй»: входные данные разбиваются на два блока равного размера; две подзадачи для этих блоков рекурсивно решаются по отдельности; два результата объединя-

ются в одно решение, с квадратичными затратами времени для исходного деления и итогового объединения.

Для наших целей важно то, что алгоритмы такого рода имеют время выполнения $T(n)$, удовлетворяющее следующему рекуррентному отношению:

(5.6) Для некоторой константы c

$T(n) \leq 2T(n/2) + cn^2$
при $n > 2$, и
$T(2) \leq c$.

Инстинктивно хочется предположить, что решение будет иметь вид $T(n) = O(n^2 \log n)$, поскольку оно почти идентично (5.1), если не считать увеличения объема работы на уровень с коэффициентом, равным размеру входных данных. На самом деле эта верхняя граница верна (хотя для обоснования потребуются более содержательные обоснования, чем предыдущее предложение), но также выясняется, что верхнюю границу можно усилить. Для этого мы выполним раскрутку рекуррентности по стандартному шаблону.

- *Анализ нескольких начальных уровней*: на первом уровне рекурсии имеется одна задача с размером n, которая выполняется за время cn^2 плюс время, потраченное на все последующие рекурсивные вызовы. На следующем уровне имеются две задачи с размером $n/2$. Каждая из них вносит вклад $(cn/2)^2 = cn^2/4$, что в сумме дает максимум $cn^2/2$, и снова плюс время, потраченное на последующие рекурсивные вызовы. На третьем уровне имеются четыре задачи, каждая из которых имеет размер $n/4$ и занимает время максимум $c(n/4)^2 = cn^2/16$, в сумме не более $cn^2/4$. Уже видно, что ситуация отличается от решения для аналогичного рекуррентного отношения (5.1); тогда общий объем работы на уровень оставался неизменным, а здесь он сокращается.
- *Выявление закономерности*: на произвольном уровне j находятся 2^j подзадач, каждая из которых имеет размер $n/2^j$. Следовательно, общий объем работы на этом уровне ограничивается $2^j c\left(\frac{n}{2^j}\right)^2 = cn^2/2^j$.
- *Суммирование по всем уровням рекурсии*: после всех вычислений мы пришли почти к такой же сумме, которая использовалась для случая $q = 1$. Получаем

$$T(n) \leq \sum_{j=0}^{\log_2 n-1} \frac{cn^2}{2^j} = cn^2 \sum_{j=0}^{\log_2 n-1} \left(\frac{1}{2^j}\right) \leq 2cn^2 = O\left(n^2\right).$$

Второе неравенство следует из того факта, что мы имеем дело со сходящейся геометрической суммой.

Оглядываясь назад, видим, что наше исходное предположение $T(n) = O(n^2 \log n)$, базирующееся на аналогии с (5.1), было завышенным из-за скорости убывания составляющей n^2 при замене ее $\left(\frac{n}{2}\right)^2$, $\left(\frac{n}{4}\right)^2$, $\left(\frac{n}{8}\right)^2$ и т. д. в процессе раскрутки

рекуррентного отношения. Из-за этого мы получаем геометрическую сумму вместо суммы, растущей на фиксированную величину по всем n уровням (как в решении (5.1)).

5.3. Подсчет инверсий

Глава началась с обсуждения методов разрешения некоторых распространенных рекуррентностей. В оставшейся части главы будет продемонстрировано применение принципа «разделяй и властвуй» к задачам из разных областей; информация, представленная в предыдущих разделах, будет использоваться для ограничения времени выполнения этих алгоритмов. Сначала мы увидим, как разновидность метода сортировки слиянием используется для решения задачи, не связанной напрямую с сортировкой чисел.

Задача

Следующая задача встречается при анализе ранжирования, которое начинает играть важную роль во многих современных областях. Например, некоторые сайты применяют метод, называемый *совместной фильтрацией*, чтобы сопоставить ваши предпочтения (книги, кино, рестораны) с предпочтениями других людей в Интернете. Обнаружив других людей с «похожими» вкусами (которые определяются сравнением оценок, присвоенных ими и вами разным вещам), сайт рекомендует новые ресурсы, которые понравились вашим «единомышленникам». Другое практическое применение встречается в инструментах *метапоиска*, которые выполняют один запрос в разных поисковых системах, а потом пытаются синтезировать результаты на основании сходств и различий между рангами, полученными от разных поисковых систем.

Центральное место в таких приложениях занимает задача сравнения двух ранжировок. Вы оцениваете множество из n фильмов, а система совместной фильтрации по своим базам данных ищет других людей с «похожими» оценками. Но как измерить сходство оценок двух людей на числовом уровне? Очевидно, идентичные оценки очень похожи, а полностью противоположные сильно различаются; нужна некая метрика, способная выполнять интерполяцию на середине шкалы.

Рассмотрим задачу сравнения двух ранжировок набора фильмов: вашей и чьей-то еще. Естественный метод — пометить фильмы от 1 до n в соответствии с вашей ранжировкой, затем упорядочить эти метки в соответствии с ранжировкой другого человека и посмотреть, сколько пар «нарушает порядок». Более конкретная формулировка задачи выглядит так: имеется последовательность из n чисел $a_1, ..., a_n$; предполагается, что все числа различны. Требуется определить метрику, которая сообщает, насколько список отклоняется от упорядочения по возрастанию; значение метрики должно быть равно 0, если $a_1 < a_2 < ... < a_n$, и должно быть тем выше, чем сильнее нарушен порядок чисел.

Естественное количественное выражение этого понятия основано на подсчете *инверсий*. Мы будем говорить, что два индекса $i < j$ образуют инверсию, если $a_i > a_j$, то есть если два элемента a_i и a_j идут «не по порядку». Нужно определить количество инверсий в последовательности $a_1, ..., a_n$.

Чтобы вы лучше поняли смысл определения, рассмотрим пример с последовательностью 2, 4, 1, 3, 5. В нем присутствуют три инверсии: (2, 1), (4, 1) и (4, 3). Также существует элегантный геометрический способ наглядного представления инверсий, изображенный на рис. 5.4: сверху записывается последовательность входных чисел в порядке, в котором они заданы, а ниже — эти же числа, упорядоченные по возрастанию. Затем каждое число соединяется отрезком со своей копией в нижнем списке. Каждая пересекающаяся пара отрезков соответствует одной паре, находящейся в обратном порядке в двух списках, — иначе говоря, инверсии.

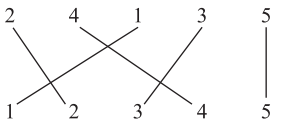


Рис. 5.4. Подсчет инверсий в последовательности 2, 4, 1, 3, 5. Каждая пересекающаяся пара отрезков соответствует паре, находящейся в обратном порядке во входном и упорядоченном списках, — иначе говоря, инверсии

Обратите внимание, как количество инверсий плавно интерполируется между полным совпадением (когда последовательность упорядочена по возрастанию, инверсий нет) и полным несовпадением (если последовательность упорядочена по убыванию, каждая пара образует инверсию и общее количество инверсий равно $\binom{n}{2}$).

Разработка и анализ алгоритма

Как проще всего посчитать инверсии? Разумеется, можно проверить каждую пару чисел (a_i, a_j) и определить, образуют ли они инверсию; проверка займет время $O(n^2)$.

Сейчас мы покажем, как подсчитать инверсии намного быстрее, за время $O(n \log n)$. Так как возможно квадратичное количество инверсий, такой алгоритм должен быть способен подсчитать инверсии без проверки отдельных инверсий. Общая идея состоит в применении стратегии (†) из раздела 5.1. Мы назначаем $m = \lceil n/2 \rceil$ и делим список на две части: $a_1, ..., a_m$ и $a_{m+1}, ..., a_n$. Сначала количество инверсий подсчитывается в каждой половине по отдельности, а затем считаются инверсии (a_i, a_j) для двух чисел, принадлежащих разным половинам; хитрость в том, что это нужно сделать за время $O(n)$, если мы хотим применить (5.2). Обратите внимание: инверсии «первая половина/вторая половина» имеют очень удобную форму: они представляют собой пары (a_i, a_j), в которых a_i находится в первой половине, a_j находится во второй половине и $a_i > a_j$.

Чтобы упростить подсчет инверсий между половинами, мы заставим алгоритм также рекурсивно сортировать числа в двух половинах. Некоторое возрастание объема работы на шаге рекурсии (сортировка и подсчет инверсий) упростит «объединяющую» часть алгоритма.

Итак, главной частью процесса станет процедура «слияния с подсчетом». Предположим, мы рекурсивно отсортировали первую и вторую половины списка и подсчитали инверсии в каждой половине. Теперь имеются два отсортированных списка A и B, содержащих первую и вторую половины соответственно. Мы хотим объединить их с построением одного отсортированного списка C, одновременно подсчитав пары (a, b), у которых $a \in A$, $b \in B$, и $a > b$. По приведенному выше определению это именно то, что необходимо для «объединяющего» шага, вычисляющего количество инверсий между половинами.

Эта задача тесно связана с более простой задачей из главы 2, в которой был сформирован соответствующий «объединяющий» шаг для сортировки слиянием: тогда у нас были два отсортированных списка A и B, которые нужно было объединить в один отсортированный список за время $O(n)$. На этот раз нужно сделать кое-что еще: не только построить один отсортированный список из A и B, но и подсчитать количество «инвертированных пар» (a, b), для которых $a \in A$, $b \in B$, и $a > b$.

Как выясняется, это делается практически так же, как для слияния. Процедура «слияния с подсчетом» проходит по отсортированным спискам A и B, удаляя элементы от начала и присоединяя их к отсортированному списку C. На каждом конкретном шаге для каждого списка доступен указатель *Current*, обозначающий текущую позицию. Предположим, эти указатели в настоящее время указывают на элементы a_i и b_j. За один шаг мы сравниваем элементы a_i и b_j, удаляем меньший элемент из списка и присоединяем его в конец списка C.

Со слиянием понятно, но как подсчитать количество инверсий? Так как списки A и B отсортированы, отслеживать количество обнаруженных инверсий на самом деле очень просто. Каждый раз, когда элемент a_i добавляется к C, новые инверсии не возникают, так как a_i меньше всех элементов, оставшихся в списке B, и должен предшествовать им всем. С другой стороны, если элемент b_j присоединяется к списку C, значит, он меньше всех оставшихся элементов A и должен следовать после них всех, поэтому счетчик инверсий увеличивается на количество элементов, оставшихся в A. Это центральная идея алгоритма: за постоянное время учитывается потенциально высокое количество инверсий. Процесс проиллюстрирован на рис. 5.5.

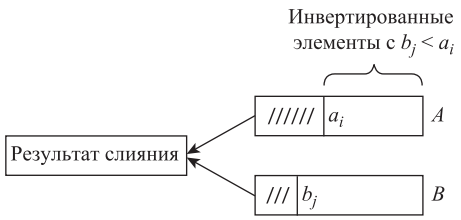


Рис. 5.5. Слияние двух отсортированных списков с одновременным подсчетом инверсий между ними

Ниже приведено описание алгоритма на псевдокоде.

```
Merge-and-Count(A,B)
   Хранить для каждого списка указатель Current, инициализируемый
      указателем на начальный элемент
   Хранить количество инверсий в переменной Count, инициализируемой 0
   Пока оба списка не пусты:
      Пусть a_i и b_j – элементы, на которые указывают указатели Current
      Присоединить меньший из них к выходному списку
      Если меньшим является элемент b_j, то
         Увеличить Count на количество элементов, оставшихся в A
      Конец Если
      Переместить указатель Current в списке, из которого был выбран
         меньший элемент.
   Конец Пока
   Когда один из списков опустеет, присоединить остаток другого списка
     к выходному списку
   Вернуть Count и объединенный список
```

Время выполнения алгоритма слияния с подсчетом может быть ограничено аргументом, аналогичным тому, который использовался для исходного алгоритма слияния в сортировке слиянием: каждая итерация цикла выполняется за постоянное время, и при каждой итерации в выходной список добавляется элемент, который исключается из рассмотрения. Следовательно, количество итераций не может превышать сумму исходных длин A и B, а значит, общее время выполнения составляет $O(n)$.

Процедура используется в рекурсивном алгоритме, который одновременно сортирует и подсчитывает количество инверсий в списке L.

```
Sort-and-Count(L)
   Если список содержит один элемент, инверсий нет
   Иначе
      Разделить список на две половины:
         A содержит первые n/2 элементов
         B содержит остальные n/2 элементов
      (r_A, A) = Sort-and-Count(A)
      (r_B, B) = Sort-and-Count(B)
      (r, L) = Merge-and-Count(A, B)
   Конец Если
   Вернуть r = r_A + r_B + r и отсортированный список L
```

Так как слияние с подсчетом выполняется за время $O(n)$, время выполнения $T(n)$ полной процедуры сортировки с подсчетом удовлетворяет рекуррентному отношению (5.1). Из (5.2) следует (5.7).

(5.7) Алгоритм сортировки с подсчетом i правильно сортирует входной список и подсчитывает количество инверсий; для списка с n элементами он выполняется за время $O(n \log n)$.

5.4. Поиск ближайшей пары точек

В этом разделе описана другая задача, которая может быть решена алгоритмом рассматриваемого типа; но чтобы найти правильный способ «слияния» решений двух порождаемых подзадач, придется хорошенько подумать.

Задача

Формулировка задачи очень проста: для заданных n точек на плоскости найти пару точек, расположенных ближе друг к другу.

В начале 1970-х годов эту задачу рассматривали М. И. Шамос и Д. Хоуи в ходе проекта по поиску эффективных алгоритмов для базовых вычислительных примитивов для геометрических задач. Эти алгоритмы заложили основу зарождающейся в то время области вычислительной геометрии и проникли в такие области, как компьютерная графика, обработка изображений, географические информационные системы и молекулярное моделирование. И хотя задача нахождения ближайшей пары точек принадлежит к числу самых естественных алгоритмических задач в геометрии, найти для нее эффективный алгоритм оказывается на удивление сложно. Очевидно, существует решение $O(n^2)$ — вычислить расстояния между каждой парой точек и выбрать минимум; Шамос и Хоуи задались вопросом, существует ли алгоритм, который был бы асимптотически более быстрым, чем квадратичный. Прошло довольно много времени, прежде чем был получен ответ на этот вопрос. Приведенный ниже алгоритм $O(n \log n)$ по сути не отличается от найденного ими. Более того, когда мы вернемся к этой задаче в главе 13, то увидим, что время выполнения можно дополнительно улучшить до $O(n)$ за счет применения рандомизации.

Разработка алгоритма

Начнем с определения некоторых обозначений. Имеется множество точек $P = \{p_1, \ldots, p_n\}$, в котором точка p_i имеет координаты (x_i, y_i); для двух точек $p_i, p_j \in P$ стандартное евклидово расстояние между ними будет обозначаться $d(p_i, p_j)$. Наша цель — найти пару точек p_i, p_j, минимизирующую $d(p_i, p_j)$.

Будем считать, что никакие две точки из P не имеют одинаковых значений координаты x или y. Такое предположение упрощает дальнейшее обсуждение; чтобы снять его, достаточно применить к точкам поворот, в результате которого предположение становится истинным, или слегка расширить описанный алгоритм.

Будет полезно рассмотреть одномерную версию этой задачи — она намного проще, а результаты показательны. Как найти ближайшую пару точек на прямой? Нужно сначала отсортировать точки за время $O(n \log n)$, а затем перебрать отсортированный список с вычислением расстояния от каждой точки до следующей. Понятно, что одно из этих расстояний должно быть минимальным.

На плоскости можно попытаться отсортировать точки по координате y (или x) и надеяться, что две ближайшие точки будут находиться поблизости в порядке сортировки. Но легко построить пример, в котором они будут находиться очень далеко, так что приспособить одномерное решение к двум измерениям не удастся.

Вместо этого будет применена стратегия в стиле «разделяй и властвуй», использованная в сортировке слиянием: мы находим ближайшую пару среди точек в «левой половине» P и ближайшую пару среди точек в «правой половине» P, после чего эта информация используется для получения общего решения за линейное время. Если разработать алгоритм с такой структурой, то решение базового рекуррентного отношения из (5.1) обеспечит время выполнения $O(n \log n)$.

Сложности возникают в последней, «объединяющей» фазе алгоритма: ни один из рекурсивных вызовов не рассматривает расстояния между точкой из левой и точкой из правой половины; всего таких расстояний $\Omega(n^2)$, но мы должны найти наименьшее из них за время $O(n)$ после завершения рекурсивных вызовов. Если это удастся сделать, то решение будет завершено: результатом становится меньшее из значений, вычисленных в ходе рекурсивных вызовов, и минимального расстояния между точками из левой и правой половин.

Подготовка рекурсии

Начнем с нескольких простых подготовительных операций. Было бы очень удобно, если бы каждый рекурсивный вызов для множества $P' \subseteq P$ начинался с двух списков: списка P'_x, в котором все точки P' отсортированы по возрастанию координаты x, и списка P'_y, в котором все точки P' отсортированы по возрастанию координаты y. Для этого можно воспользоваться описанным ниже алгоритмом.

Еще перед началом рекурсии все точки P сортируются по координате x, а потом по координате y; так получаются списки P_x и P_y. С каждым элементом списка связана запись с информацией о позиции точки в обоих списках.

Первый уровень рекурсии работает так, как описано ниже; другие уровни работают полностью аналогично. Q определяется как множество точек в первых $\lceil n/2 \rceil$ позициях списка P_x («левая половина»), а R — как множество точек в других $\lfloor n/2 \rfloor$ позициях списка P_x («правая половина»); схема разбиения представлена на рис. 5.6. За один проход по P_x и P_y за время $O(n)$ можно создать четыре списка: Q_x с точками Q, отсортированными по возрастанию координаты x; Q_y с точками Q, отсортированными по возрастанию координаты y; и два аналогичных списка R_x и R_y. Для каждого элемента каждого из этих списков также хранится его позиция в обоих исходных списках.

Затем рекурсивно определяется ближайшая пара точек в Q (с обращениями к спискам Q_x и Q_y). Предположим, q_0^* и q_1^* (правильно) возвращены как ближайшая пара точек в Q. Аналогичным образом определяется ближайшая пара точек в R, обозначим их r_0^* и r_1^*.

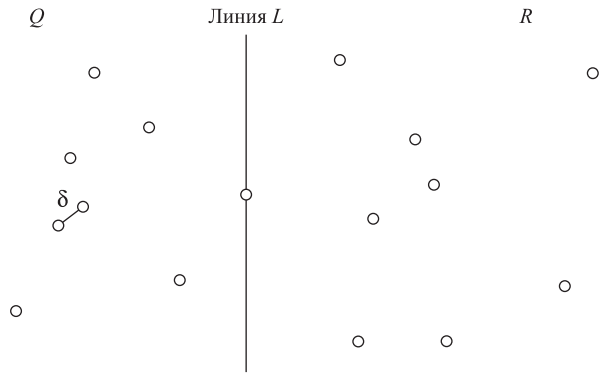


Рис. 5.6. Первый уровень рекурсии: множество точек P делится на равные половины Q и R по линии L и в каждой половине выполняется рекурсивный поиск ближайшей пары

Объединение решений

Общая схема «разделяй и властвуй» позволила нам дойти до этой точки, не углубляясь в структуру задачи нахождения ближайшей пары. Но при этом мы не решили исходную проблему: как использовать решения двух подзадач в «объединяющей» операции с линейным временем?

Введем обозначение δ для минимума из $d(q_0^*, q_1^*)$ и $d(r_0^*, r_1^*)$. Настоящий вопрос выглядит так: существуют ли точки $q \in Q$ и $r \in R$, для которых $d(q, r) < \delta$? Если нет, то ближайшая пара уже была найдена одним из предшествующих рекурсивных вызовов. Если же такие точки существуют, то ближайшие q и r образуют ближайшую пару в P.

Пусть x^* обозначает координату x крайней правой точки в Q, а L — вертикальную линию, описываемую уравнением $x = x^*$. Линия L «отделяет» Q от R. А теперь простой факт:

(5.8) Если существуют $q \in Q$ и $r \in R$, для которых $d(q, r) < \delta$, то каждая из точек q и r располагается в пределах расстояния δ от L.

Доказательство. Предположим, такие q и r существуют; обозначим $q = (q_x, q_y)$ и $r = (r_x, r_y)$. Из определения x^* следует, что $q_x \le x^* \le r_x$. Тогда

$$x^* - q_x \le r_x - q_x \le d(q, r) < \delta$$

и

$$r_x - x^* \le r_x - q_x \le d(q, r) < \delta,$$

а это значит, что у каждой из точек q и r координата x находится в пределах δ от x^* — и следовательно, ее расстояние от линии L не превышает δ. ■

Итак, чтобы найти ближайшие точки q и r, поиск можно ограничить узкой полосой, содержащей только точки P, находящиеся в пределах δ от L. Допустим, это множество обозначается $S \subseteq P$, а S_y — список, состоящий из точек S, отсортированных по возрастанию координаты y. Один проход по списку P_y строит S_y за время $O(n)$.

В контексте множества S утверждение (5.8) переформулируется следующим образом.

(5.9) Существуют $q \in Q$ и $r \in R$, для которых $d(q, r) < \delta$ в том и только в том случае, если существуют $s, s' \in S$, для которых $d(s, s') < \delta$.

На этой стадии стоит заметить, что S может быть тождественно всему множеству P; в этом случае (5.8) и (5.9) вроде бы ничего не дают. Но это впечатление обманчиво, как показывает следующий удивительный факт.

(5.10) Если $s, s' \in S$ обладают тем свойством, что $d(s, s') < \delta$, то s и s' находятся на расстоянии не более 15 позиций друг от друга в отсортированном списке S_y.

Доказательство. Рассмотрим подмножество Z плоскости, состоящее из всех точек, удаленных от L на расстояние не более δ. Разделим Z на *поля* — квадраты со стороной $\delta/2$. Один ряд Z будет состоять из четырех полей, горизонтальные стороны которых имеют одинаковые координаты y. Это множество полей изображено на рис. 5.7.

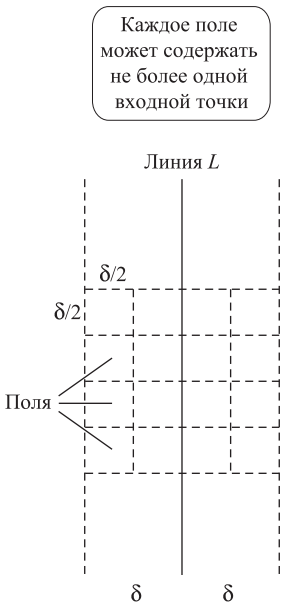


Рис. 5.7. Часть плоскости возле разделительной линии L — см. доказательство (5.10).

Допустим, две точки S лежат в одном поле. Так как все точки этого поля находятся по одну сторону от L, эти две точки либо обе принадлежат Q, либо обе принадлежат R. Но любые две точки в одном поле находятся на расстоянии не более $\delta \cdot \sqrt{2}/2 < \delta$, а это противоречит нашему определению δ как минимального расстояния между любой парой точек Q или R. Следовательно, каждое поле содержит не более одной точки S.

Теперь допустим, что $s, s' \in S$ обладают свойством $d(s, s') < \delta$ и находятся на расстоянии 16 и более позиций в S_y. Предположим без потери общности, что

s имеет меньшую координату y. Так как поле может содержать не более одной точки, между s и s' лежат как минимум три ряда Z. Но любые две точки в Z, разделенные минимум тремя рядами, должны находиться на расстоянии минимум $3\delta/2$ — противоречие. ■

Заметим, что значение 15 можно сократить; но для нас сейчас важно то, что это абсолютная константа.

В свете (5.10) алгоритм можно завершить так: мы делаем один проход по S_y и для каждого $s \in S_y$ вычисляем расстояние до каждой из следующих 15 точек в S_y. Из утверждения (5.10) следует, что при этом будет вычислено расстояние между каждой парой точек в S, находящихся на расстоянии менее δ друг от друга (если они есть). После этого можно сравнить наименьшее из этих расстояний с δ и выдать один из двух результатов: (i) ближайшую пару точек в S, если расстояние между ними меньше δ; или (ii) (обоснованное) заключение о том, что в S не существует пар точек, разделенных расстоянием не более δ. В случае (i) эта пара является ближайшей парой в P; в случае (ii) ближайшей парой в P будет ближайшая пара, найденная рекурсивными вызовами.

Обратите внимание на сходство этой процедуры с алгоритмом, отвергнутым в самом начале, в котором мы попытались ограничиться одним перебором P по координате y. Новый метод работает благодаря дополнительной информации (значение δ), полученной в результате рекурсивных вызовов, и специальной структуре множества S.

На этом описание «объединяющей» части алгоритма завершается, поскольку, согласно (5.9), мы определили, что минимальное расстояние между точкой из Q и точкой из R меньше δ, а это означает, что мы нашли ближайшую из таких пар.

Полное описание алгоритма и доказательство его правильности неявно содержатся в приведенном выше описании, но ради конкретности приведем краткие описания по обоим пунктам.

Краткое описание алгоритма

Ниже приводится высокоуровневое описание алгоритма с использованием введенных ранее обозначений.

Closest-Pair(P)
  Построить P_x и P_y (время $O(n \log n)$)
  $(p_0^*, p_1^*) =$ *Closest-Pair-Rec*(P_x, P_y)

Closest-Pair-Rec(P_x, P_y)
  Если $|P| \leq 3$
    Найти ближайшую пару вычислением всех попарных расстояний
  Конец Если

  Построить Q_x, Q_y, R_x, R_y (время $O(n)$)
  $(q_0^*, q_1^*) =$ *Closest-Pair-Rec*(Q_x, Q_y)
  $(r_0^*, r_1^*) =$ *Closest-Pair-Rec*(R_x, R_y)

```
δ = min(d(q0*, q1*), d(r0*, r1*))
  x* = максимальная координата x точки в множестве Q
  L = {(x,y) : x = x*}
  S = точки в P, находящиеся от L на расстоянии не более δ.

  Построить Sy (время O(n))
  Для каждой точки s ∈ Sy вычислить расстояние от s
     до каждой из следующих 15 точек в Sy
     Пусть s, s' — пара с минимальным расстоянием
       (время O(n))

  Если d(s, s') < δ
     Вернуть (s, s')
  Иначе Если d(q0*, q1*) < d(r0*, r1*)
     Вернуть (q0*, q1*)
  Иначе
     Вернуть (r0*, r1*)
  Конец Если
```

Анализ алгоритма

Сначала мы докажем, что алгоритм выдает правильный ответ, используя факты, установленные в ходе его разработки.

(5.11) Алгоритм правильно находит ближайшую пару точек в P.

Доказательство. Как упоминалось ранее, все компоненты доказательства уже были представлены ранее, так что осталось лишь собрать их все воедино.

Для доказательства правильности будет использована индукция по размеру P; случай $|P| \leq 3$ очевиден. Для заданного P ближайшая пара рекурсивных вызовов правильно вычисляется индукцией. Из (5.10) и (5.9) оставшаяся часть алгоритма правильно определяет, находится ли любая пара точек S на расстоянии менее δ, и если находится — возвращает такую пару с минимальным расстоянием. Теперь у ближайшей пары P либо обе точки принадлежат одному из множеств Q или R, либо они принадлежат разным множествам. В первом случае ближайшая пара находится рекурсивным вызовом; во втором она находится на расстоянии менее δ и ищется оставшейся частью алгоритма. ■

Также определим границу времени выполнения, используя (5.2).

(5.12) Время выполнения алгоритма равно $O(n \log n)$.

Доказательство. Исходная сортировка P по x и y выполняется за время $O(n \log n)$. Время выполнения оставшейся части алгоритма удовлетворяет рекуррентному отношению (5.1), а следовательно, равно $O(n \log n)$ согласно (5.2). ■

5.5. Целочисленное умножение

А теперь рассмотрим совершенно иное применение принципа «разделяй и властвуй», в котором «стандартный» квадратичный алгоритм улучшается при помощи другой рекуррентности. В анализе более быстрого алгоритма используется одно из рекуррентных отношений, рассмотренных в разделе 5.2, в котором на каждом уровне порождается более двух рекурсивных вызовов.

Задача

В каком-то смысле задача умножения двух целых чисел настолько фундаментальна, что на первый взгляд может показаться, что это вообще не алгоритмический вопрос. Но в действительности на уроках математики в младших классах объясняют конкретный (и довольно эффективный) алгоритм умножения двух целых чисел x и y, каждое из которых состоит из n цифр. Сначала вычисляется «частичное произведение», для чего каждая цифра y отдельно умножается на x, а полученные частичные произведения суммируются. (Рисунок 5.8 поможет вам вспомнить этот алгоритм. В начальной школе умножение всегда выполняется в десятичной системе, но по основанию 2 оно работает точно так же.) Если взять отдельную операцию с парой битов за один шаг в этом вычислении, для вычисления каждого частичного произведения потребуется время $O(n)$, а также время $O(n)$ для объединения его с накапливаемой суммой всех частичных произведений. С n частичными суммами общее время выполнения составляет $O(n^2)$.

```
  12          1100
× 13        × 1101
  36          1100
  12          0000
 156         1100
             1100
         10011100
```

a *b*

Рис. 5.8. Алгоритм умножения двух целых чисел в десятичном (*a*) и двоичном (*b*) представлении

Если вы особенно не задумывались над этим алгоритмом с начальной школы, перспективы его улучшения на первый взгляд кажутся сомнительными. Разве все эти частичные произведения не являются в каком-то смысле необходимыми? Но оказывается, время $O(n^2)$ можно улучшить при использовании другого, рекурсивного способа выполнения умножения.

Разработка алгоритма

В основу улучшенного алгоритма заложен более умный способ разбиения произведения на частичные суммы. Предположим, вычисления ведутся по основанию 2

(на самом деле это не важно), и запишем x в форме $x_1 \cdot 2^{n/2} + x_0$. Иначе говоря, x_1 соответствует «старшим» $n/2$ битам, а x_0 соответствует «младшим» $n/2$ битам. Аналогичным образом записывается $y = y_1 \cdot 2^{n/2} + y_0$. Получаем

$$xy = (x_1 \cdot 2^{n/2} + x_0)(y_1 \cdot 2^{n/2} + y_0) = x_{1y1} \cdot 2^n + (x_{1y0} + x_{0y1}) \cdot 2^{n/2} + x_{0y0}. \qquad (5.1)$$

Уравнение (5.1) сводит проблему решения одной n-разрядной задачи (умножение двух n-разрядных чисел x и y) к проблеме решения четырех $n/2$-разрядных задач (вычисление произведений x_1y_1, x_1y_0, x_0y_1 и x_0y_0). Таким образом, появляется первый кандидат на решение методом «разделяй и властвуй»: рекурсивное вычисление результатов для этих четырех $n/2$-разрядных экземпляров с последующим объединением их с использованием уравнения (5.1). Объединение решений требует постоянного количества сложений $O(n)$-разрядных чисел, поэтому оно выполняется за время $O(n)$; следовательно, время выполнения $T(n)$ ограничивается рекуррентным отношением

$$T(n) \le 4T(n/2) + cn$$

для константы c. Хватит ли этого, чтобы обеспечить субквадратичное время выполнения? Чтобы получить ответ, достаточно заметить, что это просто случай $q = 4$ класса рекуррентных отношений в (5.3). Как было показано ранее в этой главе, решение имеет вид $T(n) \le O\left(n^{\log_2 q}\right) = O\left(n^2\right)$.

Получается, что, применяя алгоритм «разделяй и властвуй» с разбиением на четыре части, мы всего лишь возвращаемся к тому же квадратичному времени более сложным способом! И чтобы добиться лучших результатов со стратегией, сводящей задачу к *n/2*-разрядным экземплярам, нужно по возможности обойтись только тремя рекурсивными вызовами. Это приведет к случаю $q = 3$ из (5.3), который, как было показано, имеет решение $T(n) \le O\left(n^{\log_2 q}\right) = O\left(n^{1.59}\right)$.

Вспомните, что нашей целью является вычисление выражения $x_1y_1 \cdot 2^n + (x_1y_0 + x_0y_1) \cdot 2^{n/2} + x_0y_0$ в уравнении (5.1). Как выясняется, существует простой прием, который позволит определить все члены этого выражения с использованием всего трех рекурсивных вызовов. Для этого следует проанализировать результат одного умножения $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$. Четыре произведения суммируются с затратами одного рекурсивного умножения. Если x_1y_1 и x_0y_0 определяются при рекурсии, то внешние члены будут определены явно, а средний член можно получить вычитанием x_1y_1 и x_0y_0 из $(x_1 + x_0)(y_1 + y_0)$. Итак, в развернутом виде наш алгоритм выглядит так:

Recursive-Multiply(x,y):
  Записать $x = x_1 \cdot 2^{n/2} + x_0$
  $y = y_1 \cdot 2^{n/2} + y_0$
  Вычислить $x_1 + x_0$ и $y_1 + y_0$
  p = *Recursive-Multiply*$(x_1 + x_0,\ y_1 + y_0)$
  x_1y_1 = *Recursive-Multiply*$(x_1,\ y_1)$
  x_0y_0 = *Recursive-Multiply*$(x_0,\ y_0)$
  Вернуть $x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$

Анализ алгоритма

Время выполнения алгоритма определяется следующим образом. Для двух n-разрядных чисел выполняется постоянное количество сложений $O(n)$-разрядных чисел помимо трех рекурсивных вызовов. Пока проигнорируем тот факт, что $x_1 + x_0$ и $y_1 + y_0$ могут содержать $n/2 + 1$ битов (вместо $n/2$), так как это не влияет на асимптотические результаты; итак, каждый из рекурсивных вызовов работает с экземпляром размера $n/2$. Вместо рекурсии с четверным ветвлением мы получаем тройное ветвление с временем выполнения, удовлетворяющим отношению

$$T(n) \leq 3T(n/2) + cn$$

для константы c.

Перед нами тот самый случай $q = 3$ из (5.3), к которому мы стремились. Используя разрешение рекуррентности, приведенное ранее в этой главе, имеем:

(5.13) Время выполнения рекурсивного умножения для двух n-разрядных множителей равно $O\left(n^{\log_2 3}\right) = O\left(n^{1.59}\right)$.

5.6. Свертки и быстрое преобразование Фурье

В последнем примере этой главы мы покажем, как базовое рекуррентное отношение из (5.1) применяется при разработке алгоритма быстрого преобразования Фурье — алгоритма с широким диапазоном практических применений.

Задача

Даны два вектора $a = (a_0, a_1, ..., a_{n-1})$ и $b = (b_0, b_1, ..., b_{n-1})$. Существует много стандартных способов их объединения: например, вычисление суммы с получением вектора $a + b = (a_0 + b_0, a_1 + b_1, ..., a_{n-1} + b_{n-1})$ или вычисление скалярного произведения в форме вещественного числа $a \cdot b = a_0b_0 + a_1b_1 + ... + a_{n-1}b_{n-1}$. (По причинам, которые вскоре станут ясны, векторы в этом разделе удобнее записывать с координатами, которые индексируются с 0, а не с 1.)

Один из способов объединения векторов, находящий очень важные практические применения (хотя и не всегда упоминаемый во вводном курсе линейной алгебры), — *свертка* $a * b$. Свертка двух векторов длины n (таких, как a и b) представляет собой вектор с $2n - 1$ координатами, в котором координата k равна

$$\sum_{\substack{(i,j):i+j=k \\ i,j<n}} a_i b_j.$$

Другими словами,

$$a * b = (a_0b_0, a_0b_1 + a_1b_0, a_0b_2 + a_1b_1 + a_2b_0, \ldots, a_{n-2}b_{n-1} + a_{n-1}b_{n-2}, a_{n-1}b_{n-1}).$$

Когда это определение видишь впервые, понять его смысл нелегко. Представьте себе матрицу $n \times n$, в которой элемент (i, j) равен $a_i b_j$:

$$a_0b_0\ a_0b_1\ \ldots\ a_0b_{n-2}\ a_0b_{n-1}$$

$$a_1b_0\ a_1b_1\ \ldots\ a_1b_{n-2}\ a_1b_{n-1}$$

$$a_2b_0\ a_2b_1\ \ldots\ a_2b_{n-2}\ a_2b_{n-1}$$

$$\ldots\ \ldots\ \ldots\ \ldots\ \ldots$$

$$a_{n-1}b_0\ a_{n-1}b_1\ \ldots\ a_{n-1}b_{n-2}\ a_{n-1}b_{n-1}$$

Координаты вектора свертки вычисляются суммированием по диагоналям.

Стоит отметить, что, в отличие от векторной суммы и скалярного произведения, свертка легко обобщается на векторы разной длины $a = (a_0, a_1, \ldots, a_{m-1})$ и $b = (b_0, b_1, \ldots, b_{n-1})$. В этом более общем случае $a * b$ определяется как вектор с $m + n - 1$ координатами, в котором координата k равна

$$\sum_{\substack{(i,j)|i+j=k \\ i<m,\, j<n}} a_i b_j.$$

И снова можно представить происходящее с помощью матрицы произведений $a_i b_j$; матрица стала прямоугольной, но координаты по-прежнему могут вычисляться суммированием по диагоналям. (В дальнейшем условие $i < m, j < n$ в сверточном суммировании явно упоминаться не будут, поскольку из контекста понятно, что сумма может вычисляться только по определенным элементам.)

Определение свертки понятно не сразу, но это не все: также не очень понятно, для чего нужна такая операция. В каких обстоятельствах может потребоваться вычисление свертки двух векторов? Однако свертка встречается в множестве самых разнообразных контекстов. Приведем хотя бы несколько примеров.

- Первый пример (который также доказывает, что свертка уже встречалась нам в старших классах школы, хотя и неявно) — умножение многочленов. Любой многочлен $A(x) = a_0 + a_1x + a_2x^2 + \ldots a_{m-1}x^{m-1}$ может быть естественно представлен в виде вектора коэффициентов $a = (a_0, a_1, \ldots, a_{m-1})$. Теперь для двух многочленов $A(x) = a_0 + a_1x + a_2x^2 + \ldots a_{m-1}x^{m-1}$ и $B(x) = b_0 + b_1x + b_2x^2 + \ldots b_{n-1}x^{n-1}$ рассмотрим многочлен $C(x) = A(x)B(x)$, равный их произведению. В многочлене $C(x)$ коэффициент при члене x^k равен

$$c_k = \sum_{(i,j)|i+j=k} a_i b_j.$$

  Иначе говоря, вектор коэффициентов $C(x)$ вычисляется как свертка векторов коэффициентов $A(x)$ и $B(x)$.

- Пожалуй, самое важное практическое применение сверток лежит в области *обработки сигналов*. Эта тема сама по себе заслуживает отдельного учебного

курса, поэтому мы ограничимся простым примером, который дает общее представление о том, где возникают свертки.

Допустим, имеется вектор $a = (a_0, a_1, ..., a_{m-1})$, который представляет серию измерений (температура, данные биржевых котировок и т. д.), зафиксированных в m последовательных временных точках. Такие серии часто содержат высокий уровень шума из-за ошибок измерения или случайных отклонений, поэтому к ним часто применяется операция «сглаживания»: каждое значение a_i усредняется со взвешенной суммой соседей на k позиций слева и справа в последовательности; весовые коэффициенты быстро убывают при увеличении расстояния от a_i. Например, при гауссовом сглаживании a_i заменяется на

$$a_i' = \frac{1}{z}\sum_{j=i-k}^{i+k} a_j e^{-(j-i)^2}$$

с некоторым параметром «ширины» k и значением Z, выбранным просто для нормализации суммы весов в среднем до 1. Существуют некоторые проблемы с граничными условиями (например, что делать, если $i - k < 0$ или $i + k > m$?), но они могут решаться игнорированием первых и последних k элементов из сглаженного сигнала или масштабированием по другой схеме, компенсирующей недостающие элементы.

Чтобы понять, как это связано с операцией свертки, можно рассматривать операцию сглаживания следующим образом. Сначала определяется «маска»

$$w = (w_{-k}, w_{-(k-1)}, ..., w_{-1}, w_0, w_{1,..}, w_{k-1}, w_k),$$

состоящая из весов, которые должны использоваться при усреднении каждой точки. (Например, $w = \frac{1}{z}\left(e^{-k^2}, e^{-(k-1)^2}, ..., e^{-1}, 1, e^{-1}, ..., e^{-(k-1)^2}, e^{-k^2}\right)$ для случая гауссова сглаживания). Затем маска последовательно позиционируется так, чтобы она была совмещена по центру с каждой возможной точкой в серии a; для каждого варианта позиционирования вычисляется взвешенное среднее. Иначе говоря, a_i заменяется на $a_i' = \sum_{s=-k}^{k} w_s a_{i+s}$.

Последнее выражение фактически является сверткой; чтобы это стало очевидно, нужно лишь немного изменить запись. Определим $b = (b_0, b_1, ..., b_{2k})$, где $b_\ell = w_{k-\ell}$. Нетрудно убедиться в том, что с этим определением сглаженное значение вычисляется в виде

$$a_i' = \sum_{(j,\ell)\,|\,j+\ell=i+k} a_j b_\ell .$$

Другими словами, сглаженная серия представляет собой результат свертки исходного сигнала и обращенной маски (с некоторыми несуществующими координатами в начале и в конце).

- Последний пример — задача объединения гистограмм. Предположим, по результатам социологического опроса были получены две гистограммы: с информа-

цией о ежегодном доходе всех мужчин и женщин в выборке. Нужно построить новую гистограмму, где для каждого k будет отображаться количество пар (M, W), для которых у мужчины M и женщины W суммарный доход составляет k.

Задача представляет собой свертку. Первую гистограмму можно записать в виде вектора $a = (a_0, ..., a_{m-1})$, который показывает, что существуют a_i мужчин с ежегодным доходом, равным i. Аналогичным образом вторая гистограмма записывается в виде вектора $b = (b_0, ..., b_{n-1})$. Введем обозначение c_k для количества пар (m, w) с суммарным доходом k; это количество способов, которыми можно выбрать мужчину с доходом a_i и женщину с доходом b_j для любой пары (i, j) с $i + j = k$. Другими словами,

$$c_k = \sum_{(i,j)|i+j=k} a_i b_j ,$$

так что объединенная гистограмма $c = (c_0, ..., c_{m+n-2})$ представляет собой простую свертку a и b. (С использованием вероятностной терминологии, которая будет раскрыта в главе 13, этот пример может рассматриваться как применение свертки в качестве способа вычисления распределения суммы двух независимых случайных переменных.)

Вычисление свертки

Разобравшись с тем, для чего нужна свертка, перейдем к задаче ее эффективного вычисления. Для простоты будем рассматривать случай векторов равной длины (то есть $m = n$), хотя все сказанное может быть напрямую применено к случаю векторов разной длины.

Вычисление свертки — тема более тонкая, чем кажется на первый взгляд. В конце концов, из определения свертки следует абсолютно законный способ ее вычисления: для каждого k вычисляется сумма

$$\sum_{(i,j)|i+j=k} a_i b_j ,$$

а результат используется как значение координаты k. Проблема в том, что при таком прямолинейном способе вычисления свертки приходится вычислять произведение $a_i b_j$ для каждой пары (i,j), а это означает $\Theta(n^2)$ арифметических операций. Время выполнения $O(n^2)$ для вычисления свертки выглядит естественно, так как в определении задействованы $O(n^2)$ умножений $a_i b_j$. Однако неочевидно, что вычисление свертки обязательно должно выполняться за квадратичное время, поскольку и ввод и вывод имеют только размер $O(n)$.

Возможно ли разработать алгоритм, который обходит квадратичный размер определения свертки и вычисляет его другим, более умным способом?

Как ни удивительно, это возможно. Ниже описан метод, который вычисляет свертку двух векторов с использованием только $O(n \log n)$ арифметических операций. В его основу заложен эффективный прием, называемый *быстрым преобразованием Фурье* (Fast Fourier Transform).

Быстрое преобразование Фурье широко применяется для анализа последовательностей числовых значений; быстрое вычисление свертки, которым мы будем заниматься здесь, — всего лишь одно из таких применений.

Разработка и анализ алгоритма

Чтобы преодолеть ограничение квадратичного времени для свертки, мы воспользуемся связью между сверткой и умножением двух многочленов, как в первом из рассмотренных выше примеров. Но вместо того, чтобы использовать свертку как примитив при умножении многочленов, мы будем использовать эту связь в обратном направлении.

Предположим, даны векторы $a = (a_0, a_1, ..., a_{n-1})$ и $b = (b_0, b_1, ..., b_{n-1})$. Будем рассматривать их как многочлены $A(x) = a_0 + a_1x + a_2x^2 + ... a_{n-1}x^{n-1}$ и $B(x) = b_0 + b_1x + b_2x^2 + ... b_{n-1}x^{n-1}$ и посмотрим, как вычислить их произведение $C(x) = A(x)B(x)$ за время $O(n \log n)$. Если $c = (c_0, c_1, ..., c_{2n-2})$ — вектор коэффициентов C, то, как было сказано выше, c в точности представляет собой свертку $a * b$, поэтому нужный ответ можно напрямую получить из коэффициентов $C(x)$.

Вместо того чтобы перемножать A и B в алгебраическом виде, мы можем рассматривать их как функции переменной x и перемножим так, как описано ниже.

(i) Сначала выбираем $2n$ значений $x_1, x_2, ..., x_{2n}$ и вычисляем $A(x_j)$ и $B(x_j)$ для каждого $j = 1, 2, ..., 2n$.

(ii) Теперь $C(x_j)$ легко вычисляется для каждого j как произведение двух чисел $A(x_j)$ и $B(x_j)$.

(iii) Остается восстановить C по значениям $x_1, x_2, ..., x_{2n}$. Для этого мы воспользуемся одним фундаментальным свойством многочленов: любой многочлен степени d может быть восстановлен по своим значениям для произвольного набора из $d + 1$ и более точек. Этот механизм, называемый *полиномиальной интерполяцией*, более подробно рассматривается ниже. А пока заметим, что для многочленов A и B, степень которых не превышает $n - 1$, степень произведения C не превышает $2n - 2$, что позволяет реконструировать многочлен по значениям $C(x_1), C(x_2), ..., C(x_{2n})$, вычисленным на шаге (ii).

Этот метод умножения многочленов имеет как положительные аспекты, так и недостатки. Сначала о хорошем: шаг (ii) требует только $O(n)$ арифметических операций, поскольку в нем задействовано умножение $O(n)$ чисел. Но на шагах (i) и (iii) ситуация выглядит уже не столь радужно. В частности, вычисление многочленов A и B для одного значения требует $\Omega(n)$ операций, а наш план требует выполнения $2n$ таких вычислений. Казалось бы, мы снова возвращаемся к квадратичному времени.

Чтобы эта идея заработала, необходимо найти множество из $2n$ значений $x_1, x_2, ..., x_{2n}$, которые связаны определенным образом — например, для которых работа по вычислению A и B может быть повторно использована в разных вычислениях. Множеством, для которого этот способ отлично работает, является множество *комплексных корней из единицы*.

Комплексные корни из единицы

На этой стадии необходимо вспомнить несколько фактов, касающихся комплексных чисел и их роли в решениях полиномиальных уравнений.

Нам известно, что комплексные числа могут рассматриваться как точки «комплексной плоскости», оси которой соответствуют действительной и мнимой части числа. Комплексное число записывается в полярных координатах по отношению к этой плоскости в виде $re^{\theta i}$, где $e^{\omega i} = -1$ (и $e^{2\omega i} = 1$). Для положительного целого числа k полиномиальное уравнение $x^k = 1$ имеет k разных комплексных корней, которые легко определяются. Каждое из комплексных чисел $\omega_{j,k} = e^{2\omega ji/k}$ (для $j = 0, 1, 2, ..., k - 1$) удовлетворяет этому уравнению, поскольку

$$(e^{2\pi ji/k})^k = e^{2\pi ji} = (e^{2\pi j})^j = 1^j = 1.$$

А так как все эти числа различны, то и корни тоже различны. Эти числа называются *корнями k-й степени из единицы*. Их можно представить как множество из k точек, разделенных одинаковыми расстояниями на единичном круге комплексной плоскости, как показано на рис. 5.9 для $k = 8$.

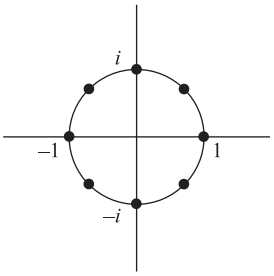


Рис. 5.9. Корни 8-й степени из единицы на комплексной плоскости

Для наших чисел $x_1, ..., x_{2n}$, для которых должны вычисляться A и B, мы выберем корни $(2n)$-й степени из единицы. Стоит запомнить (хотя это и не обязательно для понимания алгоритма), что использование комплексных корней из единицы лежит в основе самого названия быстрого преобразования Фурье: представление многочлена P степени d его значениями для корней $(d + 1)$-й степени без единицы иногда называется *дискретным преобразованием Фурье* для P; «сердцем» нашей процедуры является метод ускорения этих вычислений.

Рекурсивная процедура вычисления многочлена

Мы хотим создать алгоритм рекурсивного вычисления A для каждого из корней $(2n)$-й степени из единицы, чтобы воспользоваться знакомым рекуррентным отношением из (5.1), а именно $T(n) \le 2T(n/2) + O(n)$, где $T(n)$ в данном случае обозначает количество операций, необходимых для вычисления многочлена степени $n - 1$ для всех корней (2n)-й степени из единицы. Для простоты при описании алгоритма будем считать, что n является степенью 2.

Как разбить вычисление многочлена на две подзадачи равного размера? Воспользуемся полезным приемом: определим два многочлена $A_{even}(x)$ и $A_{odd}(x)$, состоящих из четных и нечетных коэффициентов A соответственно. То есть

$$A_{even}(x) = a_0 + a_{2x} + a_{4x}{}^2 + \cdots + a_{n-2x}{}^{(n-1)/2}$$

и

$$A_{odd}(x) = a_1 + a_{3x} + a_{5x}{}^2 + \cdots + a_{(n-1)x}{}^{(n-2)/2}.$$

Простые алгебраические выкладки показывают, что

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2).$$

Итак, в нашем распоряжении появился способ вычисления $A(x)$ с постоянным количеством операций через вычисление двух многочленов, степень каждого из которых равна половине степени A.

Теперь предположим, что каждый из многочленов A_{even} и A_{odd} вычисляется для корней n-й степени из единицы. Эта задача в точности соответствует той, с которой мы сталкиваемся для A и корней $(2n)$-й степени из единицы, за исключением того, что входные данные уменьшились вдвое: степень равна $(n - 2)/2$ вместо $n - 1$, а вместо $2n$ используются n корней. Следовательно, эти вычисления могут выполняться за время $T(n/2)$ для каждого из многочленов A_{even} и A_{odd}, с суммарным временем $2T(n/2)$.

Мы очень близко подошли к созданию рекурсивного алгоритма, который удовлетворяет условию (5.1) и обеспечивает желаемое время выполнения; остается лишь произвести вычисления A для корней $(2n)$-й степени из единицы с использованием $O(n)$ дополнительных операций. Но с учетом результатов рекурсивных вызовов для A_{even} и A_{odd} это несложно. Рассмотрим один из этих корней из единицы $\omega_{j,2n} = e^{2\pi ji/2n}$. Величина $\omega_{j,2n}^2$ равна $(e^{2\pi ji/2n})^2 = e^{2\pi ji/n}$, а следовательно, $\omega_{j,2n}^2$ является корнем n-й степени из единицы. Таким образом, когда мы переходим к вычислениям

$$A(\omega_{j,2n}) = A_{even}(\omega_{j,2n}^2) + \omega_{j,2n} A_{odd}(\omega_{j,2n}^2),$$

выясняется, что оба вычисления в правой части были выполнены на шаге рекурсии и $A(\omega_{j,2n})$ можно определить с постоянным количеством операций. Таким образом, выполнение этих операций для всех $2n$ корней из единицы означает $O(n)$ дополнительных операций после двух рекурсивных вызовов, а следовательно, граница $T(n)$ количества операций в самом деле удовлетворяет отношению $T(n) \leq 2T(n/2) + O(n)$. Та же процедура применяется для вычисления многочлена B для корней $(2n)$-й степени из единицы, и это дает желаемую границу $O(n \log n)$ для шага (i) нашей структуры алгоритма.

Полиномиальная интерполяция

Мы уже видели, как вычислить A и B для множества всех корней $(2n)$-й степени из единицы с использованием $O(n \log n)$ операций; и как объясняется выше, произведения $C(\omega_{j,n}) = A(\omega_{j,2n})B(\omega_{j,2n})$ вычисляются за $O(n)$ дополнительных операций.

Таким образом, для завершения алгоритма умножения A и B необходимо выполнить шаг (iii) приведенной выше схемы с использованием $O(n \log n)$ операций для реконструкции C из значений корней $(2n)$-й степени из единицы.

В описании этой части алгоритма следует иметь в виду один важный момент: как выясняется, задача реконструкции C решается простым определением соответствующего многочлена (см. ниже D) и его вычислением для корней $(2n)$-й степени из единицы. Только что было показано, как это делается с использованием $O(n \log n)$ операций, поэтому мы делаем это снова, выполняя дополнительные $O(n \log n)$ операций; это приводит к завершению алгоритма.

Рассмотрим многочлен $C(x) = \sum_{s=0}^{2n-1} c_s x^s$, который нужно реконструировать по значениям $C(\omega_{s,2n})$ в корнях $(2n)$-й степени из единицы. Определим новый многочлен $D(x) = \sum_{s=0}^{2n-1} d_s x^s$, где $d_s = C(\omega_{s,2n})$. Теперь рассмотрим значения $D(x)$ для корней $(2n)$-й степени из единицы.

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{s=0}^{2n-1} C(\omega_{s,2n})\omega_{j,2n}^s \\ &= \sum_{s=0}^{2n-1}\left(\sum_{t=0}^{2n-1} c_t \omega_{s,2n}^t\right)\omega_{j,2n}^s \\ &= \sum_{t=0}^{2n-1} c_t\left(\sum_{s=0}^{2n-1} \omega_{s,2n}^t \omega_{j,2n}^s\right) \end{aligned}$$

по определению. Теперь вспомните, что $\omega_{s,2n} = \left(e^{2\pi i/2n}\right)^s$. Используя этот факт и расширяя запись до $\omega_{s,2n} = \left(e^{2\pi i/2n}\right)^s$, даже когда $s \geq 2n$, мы получаем:

$$\begin{aligned} D(\omega_{j,2n}) &= \sum_{t=0}^{2n-1} c_t\left(\sum_{s=0}^{2n-1} e^{(2\pi i)(st+js)/2n}\right) \\ &= \sum_{t=0}^{2n-1} c_t\left(\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s\right). \end{aligned}$$

Чтобы проанализировать последнюю строку, мы воспользуемся тем фактом, что для каждого корня $(2n)$-й степени из единицы $\omega \neq 1$, имеем $\sum_{s=0}^{2n-1} \omega^s = 0$. Это следует из того, что ω по определению является корнем $x^{2n} - 1 = 0$; так как $x^{2n} - 1 = (x-1)\left(\sum_{t=0}^{2n-1} x^t\right)$ и $\omega \neq 1$, из этого следует, что ω также является корнем $\sum_{t=0}^{2n-1} x^t$.

Следовательно, единственная составляющая внешней суммы последней строки, отличная от 0, относится к c_t, для которого $\omega_{t+j,2n} = 1$; а это происходит, если $t + j$ кратно $2n$, то есть если $t = 2n - j$. Для этого значения $\sum_{s=0}^{2n-1} \omega_{t+j,2n}^s = \sum_{s=0}^{2n-1} 1 = 2n$. От-

сюда следует, что $D\left(\omega_{j,2n}\right) = 2nc_{2n-j}$. Таким образом, вычисление многочлена $D(x)$ в корнях $(2n)$-й степени из единицы дает коэффициенты многочлена $C(x)$ в обратном порядке (и умноженные на $2n$). Подведем итог:

(5.14) Для любого многочлена $C(x) = \sum_{s=0}^{2n-1} c_s x^s$ и соответствующего многочлена $D(x) = \sum_{s=0}^{2n-1} C\left(\omega_{s,2n}\right) x^s$ выполняется условие $c_s = \frac{1}{2n} D\left(\omega_{2n-s,2n}\right)$.

Все вычисления значений $D\left(\omega_{2n-s,2n}\right)$ выполняются за $O(n \log n)$ операций с использованием метода «разделяй и властвуй», разработанного для шага (i).

На этом работа подходит к концу: мы реконструировали многочлен C по его значениям для корней $(2n)$-й степени из единицы, и коэффициенты C определяют координаты искомого вектора свертки $c = a * b$.

Итак, мы успешно продемонстрировали следующее:

(5.15) Использование быстрого преобразования Фурье для определения произведения многочленов $C(x)$ позволяет вычислить свертку исходных векторов a и b за время $O(n \log n)$.

Упражнения с решениями

Упражнение с решением 1

Имеется массив A из n элементов, в которых хранятся несовпадающие числа. Известно, что последовательность значений $A[1], A[2], ..., A[n]$ унимодальна: для некоторого индекса p в диапазоне от 1 до n значения в элементах массива возрастают до позиции P, а затем убывают до позиции n. (Если нарисовать график, на котором ось x представляет позицию элемента в массиве j, а ось y — значение элемента $A[j]$, график будет возрастать до точки p на оси x, где достигается максимум, а затем будет непрерывно убывать.)

Требуется найти «пиковый элемент» p без чтения всего массива, более того, нужно ограничиться чтением минимально возможного количества элементов A. Покажите, как найти p чтением не более $O(\log n)$ элементов A.

Решение

Сначала обсудим, как вообще добиться времени выполнения $O(\log n)$, а затем вернемся к конкретной задаче. Чтобы выполнить некоторые вычисления так, чтобы количество операций не превышало $O(\log n)$, можно воспользоваться полезной стратегией, упоминавшейся в главе 2: выполнить постоянный объем работы, отбросить половину входных данных и продолжить рекурсивную обработку остатка. Например, именно эта идея обеспечила время выполнения $O(\log n)$ для бинарного поиска.

Происходящее можно рассматривать как проявление принципа «разделяй и властвуй»: для некоторой константы $c > 0$ выполняются не более c операций, после чего выполнение продолжается рекурсивно для входных данных, размер которых не превышает $n/2$. Мы будем считать, что рекурсия завершается при $n = 2$, выполняя максимум c операций для завершения вычислений. Если обозначить $T(n)$ время выполнения для входных данных с размером n, то мы имеем рекуррентное отношение

(5.16)

$$T(n) \le T(n/2) + c$$

при $n > 2$ и

$$T(2) \le c.$$

Это рекуррентное отношение нетрудно разрешить методом раскрутки, как показано ниже.

- *Анализ нескольких начальных уровней*: на первом уровне рекурсии имеется одна задача с размером n, которая выполняется за время c плюс время, потраченное на все последующие рекурсивные вызовы. На следующем уровне имеется одна задача с размером не более $n/2$, которая добавляет еще c, а на следующем уровне — одна задача с размером не более $n/4$, добавляющая еще c.
- *Выявление закономерности*: на сколько бы уровней мы ни продолжали анализ, на каждом уровне будет только одна задача: на уровне j будет одна задача с размером не более $n/2^j$, которая вносит вклад c во время выполнения независимо от j.
- *Суммирование по всем уровням рекурсии*: каждый уровень рекурсии добавляет максимум c операций, и для сокращения n до 2 потребуются $\log_2 n$ уровней рекурсии. Следовательно, общее время выполнения не превышает c, умноженного на количество уровней рекурсии, то есть максимум $c \log_2 n = O(\log n)$.

К тому же результату также можно прийти частичной подстановкой. Предположим, $T(n) \le k \log_b n$, причем k и b неизвестны. Если предположить, что это условие выполняется для меньших значений n в индукции, получим

$$\begin{aligned} T(n) &\le T(n/2) + c \\ &\le k \log_b (n/2) + c \\ &= k \log_b n - k \log_b 2 + c. \end{aligned}$$

Первое слагаемое в правой части — то, что нам нужно, остается выбрать k и b для компенсации добавления c в конце. Это можно сделать, выбрав $b = 2$ и $k = c$, так что $k \log_b 2 = c \log_2 2 = c$. Приходим к решению $T(n) \le c \log n$, которое полностью совпадает с тем, которое было получено при раскрутке рекуррентного отношения.

Наконец, стоит упомянуть, что ко времени выполнения $O(\log n)$ можно прийти при помощи практически таких же рассуждений в более общем случае, когда на каждом уровне рекурсии отбрасывается любая постоянная часть входных данных,

а экземпляр с размером n преобразуется в экземпляр с размером не более an для некоторой константы $a < 1$. Теперь для сокращения n до постоянного размера потребуется не более $\log_{1/a} n$ уровней рекурсии, а каждый уровень рекурсии требует не более c операций.

Вернемся к нашей задаче. Чтобы использовать (5.16), можно было бы проверить среднюю точку массива и попытаться определить, лежит «пиковый элемент» до или после середины.

Допустим, мы обращаемся к значению $A[n/2]$. По одному этому значению невозможно определить, лежит p до или после $n/2$, поскольку еще нужно определить, к какой половине относится $n/2$ — «восходящей» или «нисходящей». Соответственно мы проверяем значения $A[n/2 - 1]$ и $A[n/2 + 1]$. Возможны три варианта.

- Если $A[n/2 - 1] < A[n/2] < A[n/2 + 1]$, значит, $n/2$ следует строго перед p, и выполнение можно рекурсивно продолжить для элементов с $n/2 + 1$ до n.
- Если $A[n/2 - 1] > A[n/2] > A[n/2 + 1]$, значит, $n/2$ следует строго после p, и выполнение можно рекурсивно продолжить для элементов с 1 до $n/2 - 1$.
- Наконец, если $A[n/2]$ больше $A[n/2 - 1]$ и $A[n/2+ 1]$, выполнение завершено: «пиковым элементом» в данном случае является $n/2$.

Во всех перечисленных случаях выполняется не более трех обращений к массиву A, а задача сводится к задаче с размером не более половинного. Это позволяет применить (5.16) и сделать вывод, что время выполнения составляет $O(\log n)$.

Упражнение с решением 2

Вы работаете на небольшую инвестиционную компанию, проводящую интенсивную обработку данных; в вашей работе приходится снова и снова решать задачу определенного типа.

Типичная ситуация выглядит так. Проводится моделирование, при котором рассматриваются курс акций за n последовательных дней где-то в прошлом. Пронумеруем дни $i = 1, 2, ..., n$; для каждого дня i известна биржевая котировка этих акций $p(i)$ в этот день. (Для простоты будем считать, что в течение дня котировка не изменяется.) Предположим, где-то в этот период времени нужно купить 1000 акций в один день и продать все эти акции в другой (более поздний) день. Вопрос: когда покупать (и когда продавать) акции для получения максимальной прибыли? (Если за эти n дней перепродажа с прибылью невозможна, алгоритм должен сообщить об этом.)

Предположим, $n = 3, p(1) = 9, p(2) = 1, p(3) = 5$. Алгоритм должен вернуть «Покупать в день 2, продавать в день 3» (в этом случае прибыль составит \$4 на акцию — максимально возможная прибыль за этот период).

Очевидно, существует простой алгоритм с временем $O(n^2)$: опробовать все возможные комбинации дней покупки/продажи и посмотреть, какой из них обеспечивает максимальную прибыль. Ваши работодатели надеются иметь более эффективное решение.

Покажите, как найти правильные числа i и j за время $O(n \log n)$.

Решение

В этой главе уже были рассмотрены примеры, в которых перебор пар элементов методом «грубой силы» сокращается до $O(n \log n)$ методом «разделяй и властвуй». Здесь мы сталкиваемся с аналогичной проблемой — нельзя ли и в этом случае применить стратегию «разделяй и властвуй»?

Было бы естественно рассмотреть первые и последние $n/2$ дней по отдельности, рекурсивно решить проблему для каждого из этих двух множеств, а затем сформировать из полученных результатов общее решение за время $O(n)$. Тогда можно будет прийти к обычному рекуррентному отношению $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$, а значит, и ко времени $O(n \log n)$ согласно (5.1).

Кроме того, для упрощения задачи мы сделаем обычное предположение, что n является степенью 2. Такое предположение не ведет к потере общности: если n' — следующая степень 2 больше n, можно назначить $p(i) = p(n)$ для всех i между n и n'. Ответ при этом не изменится, а размер входных данных в худшем случае увеличится вдвое (что не повлияет на обозначение $O()$).

Пусть S — множество дней 1, ..., $n/2$, а S' — множество дней $n/2 + 1$, ..., n. Наш алгоритм «разделяй и властвуй» базируется на следующем наблюдении: либо существует оптимальное решение, при котором инвесторы удерживают купленные ранее акции в конце дня $n/2$, либо его не существует. Если решения не существует, то оптимальное решение является лучшим из оптимальных решений множеств S и S'. Если существует оптимальное решение, при котором акции удерживаются в конце дня $n/2$, то значение этого решения равно $p(j) - p(i)$, где $i \in S$ и $j \in S'$. Но это значение максимизируется простым выбором $i \in S$, при котором минимизируется $p(i)$, и выбором $j \in S'$, при котором максимизируется $p(j)$.

Следовательно, наш алгоритм должен взять лучшее из трех возможных решений:

- Оптимальное решение для S.
- Оптимальное решение для S'.
- Максимум $p(j) - p(i)$ для $i \in S$ и $j \in S'$.

Первые две альтернативы вычисляются за время $T(n/2)$ (каждая посредством рекурсии), а третья альтернатива вычисляется нахождением минимума в S и максимума в S', что выполняется за время $O(n)$. Следовательно, общее время выполнения $T(n)$ удовлетворяет условию $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$, как и требовалось.

Стоит заметить, что это не лучшее время выполнения, которое может быть обеспечено в этой задаче. Оптимальную пару дней можно найти за время $O(n)$ средствами динамического программирования (тема следующей главы); а в конце этой главы этот вопрос будет поставлен в упражнении 7.

Упражнения

1. Требуется проанализировать информацию из двух разных баз данных. Каждая база данных содержит n числовых значений (так что общее количество значений равно $2n$); будем считать, что одинаковых значений нет. Требуется вычислить медиану этого множества из $2n$ значений, которую мы определим как n-е значение в порядке возрастания.

   Однако доступ к информации затруднен — ее можно получить только при помощи запросов к базам данных. В одном запросе указывается значение k для одной из двух баз данных, и выбранная база данных возвращает k-е значение в порядке возрастания, содержащееся в этой базе. Так как обращения к базам занимают много времени, медиану хотелось бы вычислить с минимальным количеством запросов.

   Приведите алгоритм вычисления медианы с количеством запросов не более $O(\log n)$.

2. Вспомните задачу нахождения количества инверсий. Как и в тексте главы, дана последовательность чисел $a_1, ..., a_n$; предполагается, что все числа различны. Определим инверсию как пару $i < j$, для которой $a_i > a_j$.

   Также напомним, что задача подсчета инверсий была представлена как способ оценки различий между вариантами упорядочения. Однако кому-то эта метрика может показаться излишне чувствительной. Назовем пару *значимой инверсией*, если $i < j$, а $a_i > 2a_j$. Приведите алгоритм $O(n \log n)$ для подсчета количества значимых инверсий между двумя ранжировками.

3. Вы работаете на банк, который занимается выявлением попыток мошенничества. Поставлена следующая задача: имеется набор из n банковских карт, конфискованных по подозрению в мошенничестве. На каждой банковской карте имеется магнитная полоса с зашифрованными данными, и каждая карта соответствует определенному счету в банке. Две карты называются *эквивалентными*, если они соответствуют одному счету.

   Прочитать номер счета с банковской карты напрямую практически невозможно, но в банке имеется современный аппарат, который берет две банковские карты, и после некоторых вычислений определяет, являются ли они эквивалентными.

   Вопрос заключается в следующем: можно ли найти в множестве из n карт подмножество с размером более $n/2$, эквивалентных друг другу? Считается, что единственной возможной операцией с картами является проверка их на эквивалентность. Покажите, как найти ответ на вопрос так, чтобы количество проверок эквивалентности не превышало $O(n \log n)$.

4. Вы работаете в группе физиков, которым при планировании эксперимента требуется проанализировать взаимодействия между очень большим количеством заряженных частиц. По сути, эксперимент проходит так: имеется инертная молекулярная решетка, которая используется для размещения заряженных частиц с равными интервалами по прямой линии. Структура может моделиро-

ваться последовательностью точек {1, 2, 3, ..., n} на прямой; в каждой из точек j находится частица с зарядом q_j. (Каждый заряд может быть как положительным, так и отрицательным.)

Физики хотят проанализировать суммарную силу для каждой частицы, измерив ее и сравнив с вычислительным прогнозом. Собственно, именно в вычислительном прогнозе им нужна ваша помощь. Суммарная сила для каждой частицы j по закону Кулона равна

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}.$$

Для вычисления F_j по всем j написана следующая простая программа:

```
For j = 1, 2, ..., n
   Инициализировать F_j значением 0
   For i = 1, 2, ..., n
      If i < j then
         Прибавить Cq_iq_j / (j-i)^2 к F_j

      Иначе Если i > j
         Прибавить -Cq_iq_j / (j-i)^2 к F_j

      Конец Если
   Конец For
   Вывести F_j
Конец For
```

Нетрудно проанализировать время выполнения этой программы: каждое выполнение внутреннего цикла по i занимает время $O(n)$; этот внутренний цикл вызывается $O(n)$ раз, так что общее время выполнения составляет $O(n^2)$.

К сожалению, для больших значений n, с которыми работают физики, эта программа выполняется несколько минут. С другой стороны, экспериментальная установка оптимизирована так, чтобы в нее можно было запустить n частиц, выполнить измерения и приготовиться к обработке новых n частиц за несколько секунд. Физики хотят узнать, нельзя ли вычислить все силы F_j намного быстрее, чтобы не отставать от темпа эксперимента.

Помогите им и разработайте алгоритм, который вычисляет все силы F_j за время $O(n \log n)$.

5. *Удаление скрытых поверхностей* — задача из области компьютерной графики, которая вряд ли нуждается в долгих объяснениях: если Вуди стоит перед Баззом, то вы видите Вуди, а не Базза; если Базз стоит перед Вуди... ну, в общем, вы поняли.

Особенность удаления скрытых поверхностей заключается в том, что часто вычисления производятся намного чаще, чем подсказывает интуиция. Простой геометрический пример демонстрирует суть ускорения. Имеются n невертикальных линий на плоскости L_1, ..., L_n, i-я линия определяется уравнением $y = a_i x + b_i$. Предполагается, что никакие три линии не пересекаются в одной точке. Линия L_i будет называться *верхней* в заданной координате x_0, если координата y для точки x_0 больше координат y всех остальных линий в точке x_0: $a_i x_0 + b_i > a_j x_0 + b_j$ для всех $j = i$. Линия L_i называется *видимой*, если существует координата x, в которой она является верхней (то есть какая-то часть линии видна, если смотреть на нее «сверху вниз» из $y = \infty$).

Предложите алгоритм, который получает n линий на входе, а за время $O(n \log n)$ возвращает все видимые линии. Пример приведен на рис. 5.10.

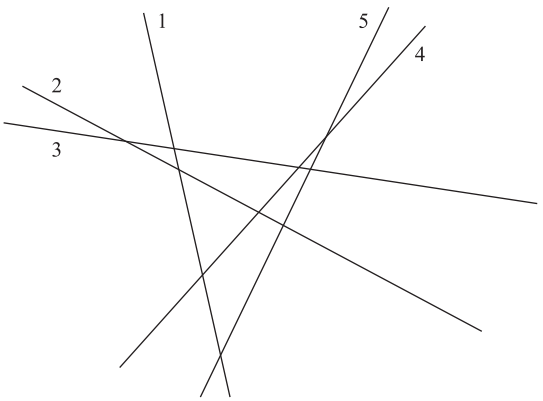


Рис. 5.10. Пример удаления скрытых поверхностей с пятью линиями (помеченными 1–5). Все линии, кроме 2, являются видимыми

6. Рассмотрим полное бинарное дерево T из n узлов, где $n = 2^d - 1$ для некоторого d. Каждый узел v дерева T помечен вещественным числом x_v. Считайте, что все вещественные числа различны. Узел v дерева T является *локальным минимумом*, если метка x_v меньше метки x_w для всех узлов w, соединенных с v ребром.

   Вы получаете полное бинарное дерево T, но метки заданы косвенно: для каждого узла v значение x_v определяется *проверкой* узла v. Покажите, как найти локальный минимум T с использованием только $O(\log n)$ проверок узлов T.

7. Предположим, имеется решетчатый граф $n \times n$ G. (Решетчатый граф $n \times n$ представляет собой граф смежности шахматной доски $n \times n$. Или для полной точности — граф, множество узлов которого является множеством всех упорядоченных пар натуральных чисел (i, j), где $1 \le i \le n$ и $1 \le j \le n$; узлы (i, j) и (k, ℓ) соединены ребром в том и только в том случае, если $|i - k| + |j - \ell| = 1$.)

   Воспользуемся терминологией из предыдущего вопроса. Снова каждый узел v помечается вещественным числом x_v; считается, что все метки различны.

Покажите, как найти локальный минимум G с использованием только $O(n)$ проверок узлов G. (Обратите внимание: G состоит из n^2 узлов.)

Примечания и дополнительная литература

Воинственное выражение «разделяй и властвуй» появилось вскоре после создания самого метода. Кнут (Knuth, 1998) приписывает Джону фон Нейману одно из первых явных выражений этого принципа — алгоритм сортировки слиянием в 1945 году. Также у Кнута (Knuth, 1997b) приведена дополнительная информация о разрешении рекуррентности.

Алгоритм вычисления ближайшей пары точек на плоскости был создан Майклом Шамосом и является одним из первых нетривиальных алгоритмов в области вычислительной геометрии. В обзорной статье Смида (Smid, 1999) обсуждается широкий спектр результатов для задач поиска ближайших точек. Более быстрый рандомизированный алгоритм для этой задачи будет рассмотрен в главе 13. (Смид также делает интересное замечание по поводу неочевидности представленного здесь алгоритма «разделяй и властвуй»: он говорит, что создатели алгоритма изначально считали, что квадратичное время является лучшим возможным для поиска ближайшей пары точек на плоскости.) В более общем плане метод «разделяй и властвуй» оказался очень полезным в вычислительной геометрии, и в книгах Препараты и Шамоса (Preparata, Shamos, 1985) и де Берга и др. (de Berg, 1997) приводится множество других примеров использования этого метода при проектировании геометрических алгоритмов.

Алгоритм умножения двух n-разрядных целых чисел за субквадратичное время разработали Карацуба и Офман (Karatsuba, Ofman, 1962). Дополнительная информация об асимптотически быстрых алгоритмах умножения приводится у Кнута (Knuth, 1997b). Конечно, чтобы субквадратичные методы обеспечили выигрыш по сравнению со стандартным алгоритмом, количество разрядов во входных данных должно быть достаточно большим.

У Пресса и др. (Press et al., 1988) приводится информация о быстром преобразовании Фурье, включая его практическое применение в обработке сигналов и других областях.

Примечания к упражнениям

Упражнение 7 основано на результатах, авторами которых были Донна Луэллин, Крейг Тови и Майкл Трик.

Глава 6

Динамическое программирование

Наше изучение алгоритмических методов началось с жадных алгоритмов, которые в определенном смысле представляют наиболее естественный подход к разработке алгоритма. Как вы видели, столкнувшись с новой вычислительной задачей, иногда можно легко предложить для нее несколько возможных жадных алгоритмов; проблема в том, чтобы определить, дают ли какие-либо из этих алгоритмов верное решение задачи во всех случаях.

Все задачи из главы 4 объединял тот факт, что в конечном итоге действительно находился работающий жадный алгоритм. К сожалению, так бывает далеко не всегда; для большинства задач настоящие трудности возникают не с выбором правильной жадной стратегии из несколько вариантов, а с тем, что естественного жадного алгоритма для задачи вообще не существует. В таких случаях важно иметь наготове другие методы. Принцип «разделяй и властвуй» иногда выручает, однако реализации этого принципа, которые мы видели в предыдущей главе, часто недостаточно эффективны для сокращения экспоненциального поиска «грубой силой» до полиномиального времени. Как было замечено в главе 5, его применения чаще позволяют сократить излишне высокое, но уже полиномиальное время выполнения до более быстрого.

В этой главе мы обратимся к более мощному и нетривиальному методу разработки алгоритмов — *динамическому программированию*. Охарактеризовать динамическое программирование будет проще после того, как вы увидите его в действии, но основная идея базируется на интуитивных представлениях, лежащих в основе принципа «разделяй и властвуй», и по сути противоположна жадной стратегии: алгоритм неявно исследует пространство всех возможных решений, раскладывает задачу на серии *подзадач*, а затем строит правильные решения подзадач все большего размера. В некотором смысле динамическое программирование может рассматриваться как метод, работающий в опасной близости от границ перебора «грубой силой»: хотя оно систематически прорабатывает большой набор возможных решений задачи, это делается без явной проверки всех вариантов. Именно из-за этой необходимости тщательного выдерживания баланса динамическое программирование бывает трудно освоить; как правило, вы начинаете чувствовать себя уверенно только после накопления немалого практического опыта.

Помня об этом, мы рассмотрим первый пример динамического программирования: задачу взвешенного интервального планирования, определение которой приводилось в разделе 1.2. Разработка алгоритма динамического программирования для этой задачи будет проводиться в два этапа: сначала как рекурсивная процедура, которая сильно напоминает поиск «грубой силой», а затем, после переосмысления этой процедуры, — как итеративный алгоритм, который строит решения последовательно увеличивающихся подзадач.

6.1. Взвешенное интервальное планирование: рекурсивная процедура

Мы уже видели, что жадный алгоритм обеспечивает оптимальное решение задачи интервального планирования, целью которой является получение множества неперекрывающихся интервалов максимально возможного размера. Задача взвешенного интервального планирования имеет более общий характер: с каждым интервалом связывается определенное значение (*вес*), а решением задачи считается множество с максимальным суммарным весом.

Разработка рекурсивного алгоритма

Поскольку исходная задача интервального планирования представляет собой частный случай, в котором все веса равны 1, мы уже знаем, что большинство жадных алгоритмов не обеспечивает оптимального решения. Но даже алгоритм, который работал ранее (многократный выбор интервала, завершающегося раньше всех), в этой ситуации уже не является оптимальным, как показывает простой пример на рис. 6.1.

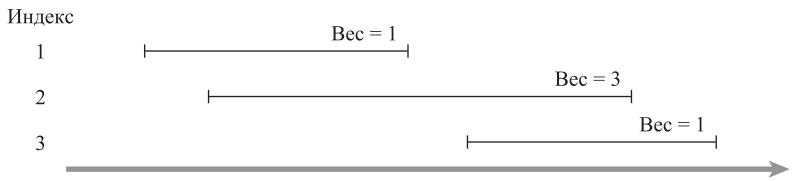


Рис. 6.1. Простой пример задачи взвешенного интервального планирования

Действительно, естественный жадный алгоритм для этой задачи неизвестен, что заставляет нас переключиться на динамическое программирование. Как упоминалось ранее, знакомство с динамическим программированием начнется с рекурсивного алгоритма, а затем в следующем разделе мы перейдем на итеративный метод, более близкий по духу к алгоритмам оставшейся части этой главы.

Воспользуемся определениями, приведенными в формулировке задачи интервального планирования из раздела 1.2. Имеются n заявок с пометками 1, ..., n; для

каждой заявки i указывается начальное время s_i и конечное время f_i. С каждым интервалом i связывается некоторое значение — вес v_i. Два интервала называются *совместимыми*, если они не перекрываются. Целью текущей задачи является нахождение подмножества $S \subseteq \{1, ..., n\}$ взаимно совместимых интервалов, максимизирующего сумму весов выбранных интервалов $\sum_{i \in S} v_i$.

Предположим, заявки отсортированы в порядке неубывания конечного времени: $f_1 \le f_2 \le ... \le f_n$. Заявка i называется предшествующей заявке j, если $i < j$. Таким образом определяется естественный порядок «слева направо», в котором будут рассматриваться интервалы. Чтобы было удобнее обсуждать этот порядок, определим $p(j)$ для интервала j как наибольший индекс $i < j$, при котором интервалы i и j не перекрываются. Другими словами, i — крайний левый интервал, который завершается до начала j. Определим $p(j) = 0$, если не существует заявки $i < j$, не перекрывающейся с j. Пример определения $p(j)$ изображен на рис. 6.2.

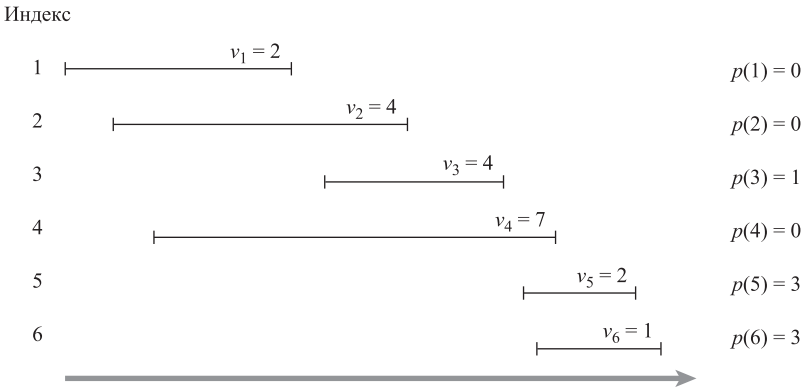


Рис. 6.2. Задача взвешенного интервального планирования с функциями $p(j)$, определенными для каждого интервала j

Допустим, у нас имеется экземпляр задачи взвешенного интервального планирования; рассмотрим оптимальное решение O, временно забыв о том, что нам о нем ничего не известно. Есть нечто совершенно очевидное, что можно утверждать о решении O: либо интервал n (последний) принадлежит O, либо нет. Попробуем исследовать обе стороны этой дихотомии. Если $n \in O$, то очевидно, ни один интервал, индексируемый строго между $p(n)$ и n, не может принадлежать O, потому что по определению $p(n)$ мы знаем, что интервалы $p(n) + 1, p(n) + 2, ..., n - 1$ — все они перекрывают интервал n. Более того, если $n \in O$, то решение O должно включать *оптимальное* решение задачи, состоящей из заявок $\{1, ..., p(n)\}$, потому что в противном случае выбор заявок O из $\{1, ..., p(n)\}$ можно было бы заменить лучшим выбором без риска перекрытия заявки n.

С другой стороны, если $n \notin O$, то O просто совпадает с оптимальным решением задачи, состоящей из заявок $\{1, ..., n - 1\}$. Рассуждения полностью аналогичны: предположим, что O не включает заявку n; если оно не выбирает оптимальное множество заявок из $\{1, ..., n - 1\}$, то его можно заменить лучшим выбором.

Все это наводит на мысль, что в процессе поиска оптимального решения для интервалов $\{1, 2, ..., n\}$ будет производиться поиск оптимальных решений меньших задач в форме $\{1, 2, ..., j\}$. Пусть для каждого значения j от 1 до n оптимальное решение задачи, состоящей из заявок $\{1, ..., j\}$, обозначается O_j, а значение (суммарный вес) этого решения — $\texttt{OPT}(j)$. (Также мы определим $\texttt{OPT}(0) = 0$ как оптимум по пустому множеству интервалов.) Искомое оптимальное решение представляет собой O_n со значением $\texttt{OPT}(n)$. Для оптимального решения O_j по интервалам $\{1, 2, ..., j\}$ из приведенных выше рассуждений (обобщенных для $j = n$) следует, что либо $j \in O_j$, и тогда $\texttt{OPT}(j) = v_j + \texttt{OPT}(p(j))$, либо $j \notin O_j$, и тогда $OPT(j) = \texttt{OPT}(j - 1)$. Так как других вариантов быть не может ($j \in O_j$ или $j \in O_j$), можно утверждать, что

(6.1) $\texttt{OPT}(j) = \max(v_j + \texttt{OPT}(p(j)), \texttt{OPT}(j - 1))$.

Как решить, принадлежит ли n оптимальному решению O_j? Это тоже несложно: n принадлежит оптимальному решению в том и только в том случае, если первый из упомянутых выше вариантов по крайней мере не хуже второго; другими словами,

(6.2) Заявка j принадлежит оптимальному решению для множества $\{1, 2, ..., j\}$ в том и только в том случае, если

$$v + \texttt{OPT}(p(j)) \geq \texttt{OPT}(j - 1).$$

Эти факты образуют первый важнейший компонент, на котором базируется решение методом динамического программирования: рекуррентное уравнение, которое выражает оптимальное решение (или его значение) в контексте оптимальных решений меньших подзадач.

Несмотря на простые рассуждения, которые привели нас к этому заключению, утверждение (6.1) само по себе является существенным достижением. Оно напрямую предоставляет рекурсивный алгоритм для вычисления $\texttt{OPT}(n)$ при условии, что заявки уже отсортированы по времени завершения и для каждого j вычислены значения $p(j)$.

```
Compute-Opt(j)
   Если j = 0
      Вернуть 0
   Иначе
      Вернуть max(v_j + Compute-Opt(p(j)), Compute-Opt(j – 1))
   Конец Если
```

Правильность алгоритма напрямую доказывается индукцией по j:

(6.3) $\texttt{Compute-Opt}(j)$ правильно вычисляет $\texttt{OPT}(j)$ для всех $j = 1, 2, ..., n$.

Доказательство. По определению $\texttt{OPT}(0) = 0$. Теперь возьмем некоторое значение $j > 0$ и предположим, что $\texttt{Compute-Opt}(i)$ правильно вычисляет $\texttt{OPT}(i)$ для всех $i < j$. Из индукционной гипотезы следует, что $\texttt{Compute-Opt}(p(j)) = \texttt{OPT}(p(j))$ и $\texttt{Compute-Opt}(j - 1) = \texttt{OPT}(j - 1)$; следовательно, из (6.1)

$\texttt{OPT}(j) = \max(v_j + \texttt{Compute-Opt}(p(j)), \texttt{Compute-Opt}(j - 1))$
$= \texttt{Compute-Opt}(j)$ ■

К сожалению, если реализовать алгоритм Compute-Opt в этом виде, его выполнение в худшем случае займет экспоненциальное время. Например, на рис. 6.3 изображено дерево вызовов для экземпляра задачи на рис. 6.2: оно очень быстро расширяется из-за рекурсивного ветвления. Или более экстремальный пример: в системе с частым наслоением наподобие изображенной на рис. 6.4, где $p(j) = j - 2$ для всех $j = 2, 3, 4, ..., n$, мы видим, что Compute-Opt(j) генерирует раздельные рекурсивные вызовы для задач с размерами $j - 1$ и $j - 2$. Другими словами, общее количество вызовов Compute-Opt в этой задаче будет расти в темпе чисел Фибоначчи, что означает экспоненциальный рост. Таким образом, решение с полиномиальным временем так и остается нереализованным.

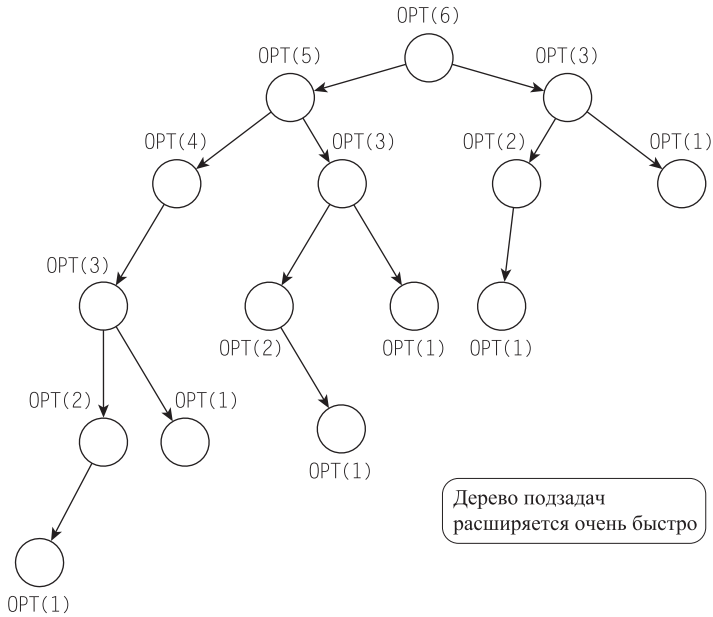


Рис. 6.3. Дерево подзадач, вызываемых Compute-Opt, для экземпляра задачи на рис. 6.2

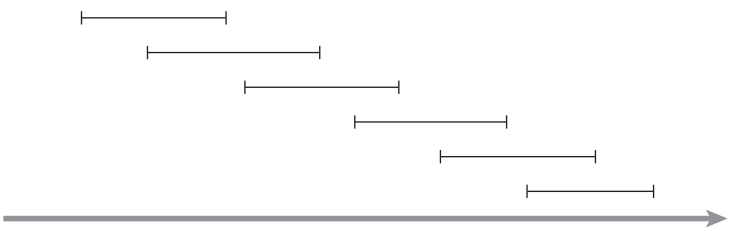


Рис. 6.4. Экземпляр задачи взвешенного интервального планирования, для которого простая рекурсия Compute-Opt занимает экспоненциальное время. Веса интервалов в этом примере равны 1

Мемоизация рекурсии

Вообще говоря, до алгоритма с полиномиальным временем не так уж далеко. Фундаментальный факт, который становится вторым критическим компонентом решения методом динамического программирования, заключается в том, что наш рекурсивный алгоритм Compute-Opt в действительности решает $n + 1$ разных подзадач: Compute-Opt(0), Compute-Opt(1), ..., Compute-Opt(n). Тот факт, что он выполняется за экспоненциальное время, приведен просто из-за впечатляющей избыточности количества выдач каждого из этих вызовов.

Как устранить всю эту избыточность? Можно сохранить значение Compute-Opt в глобально-доступном месте при первом вычислении и затем просто использовать заранее вычисленное значение вместо всех будущих рекурсивных вызовов. Метод сохранения ранее вычисленных значений называется *мемоизацией* (memoization).

Описанная стратегия будет реализована в более «умной» процедуре M-Compute-Opt. Процедура использует массив $M[0 \ldots n]$; элемент $M[j]$ изначально содержит «пустое» значение, но принимает значение Compute-Opt(j) после первого вычисления. Чтобы определить OPT(n), мы вызываем M-Compute-Opt(n).

```
M-Compute-Opt(j)
   Если j = 0
      Вернуть 0
   Иначе Если M[j] не пусто
      Вернуть M[j]
   Иначе
      Определить M[j] = max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
      Вернуть M[j]
   Конец Если
```

Анализ мемоизированной версии

Конечно, этот алгоритм очень похож на предыдущую реализацию; однако мемоизация позволяет сократить время выполнения.

(6.4) Время выполнения M-Compute-Opt(n) равно $O(n)$ (предполагается, что входные интервалы отсортированы по конечному времени).

Доказательство. Время, потраченное на один вызов M-Compute-Opt, равно $O(1)$ без учета времени, потраченного в порожденных им рекурсивных вызовах. Таким образом, время выполнения ограничивается константой, умноженной на количество вызовов M-Compute-Opt. Так как сама реализация не предоставляет явной верхней границы для количества вызовов, для получения границы мы попытаемся найти хорошую метрику «прогресса».

Самой полезной метрикой прогресса является количество «непустых» элементов M. В исходном состоянии оно равно 0; но при каждом переходе на новый уровень с выдачей двух рекурсивных вызовов M-Compute-Opt заполняется новый элемент, а следовательно, количество заполненных элементов увеличивается на 1. Так как M содержит только $n + 1$ элемент, из этого следует, что количество вызовов

M-Compute-Opt не превышает $O(n)$, а следовательно, время выполнения M-Compute-Opt(n) равно $O(n)$, как и требовалось. ■

Вычисление решения помимо его значения

Пока что мы вычислили только значение оптимального решения; вероятно, также желательно получить полный оптимальный набор интервалов. Алгоритм M-Compute-Opt легко расширяется для отслеживания оптимального решения: можно создать дополнительный массив S, в элементе $S[i]$ которого хранится оптимальное множество интервалов из $\{1, 2, ..., i\}$. Однако наивное расширение кода для сохранения решений в массиве S увеличит время выполнения с дополнительным множителем $O(n)$: хотя позиция в массиве M может обновляться за время $O(1)$, запись множества в массив S занимает время $O(n)$. Чтобы избежать возрастания $O(n)$, вместо явного хранения S можно восстановить оптимальное решение по значениям, хранящимся в массиве M после вычисления оптимального значения.

Из (6.2) известно, что j принадлежит оптимальному решению для множества интервалов $\{1, ..., j\}$ в том, и только в том случае, если $v_j + \text{OPT}(p(j)) \geq \text{OPT}(j-1)$. Воспользовавшись этим фактом, получаем простую процедуру, которая «проходит в обратном направлении» по массиву M, чтобы найти множество интервалов оптимального решения.

```
Find-Solution(j)
Если j = 0
   Ничего не выводить
Иначе
   Если v_j + M[p(j)] ≥ M[j − 1]
      Вывести j вместе с результатом Find-Solution(p(j))
   Иначе
      Вывести результат Find-Solution(j – 1)
   Конец Если
Конец Если
```

Поскольку Find-Solution рекурсивно вызывает себя только для строго меньших значений, всего делается $O(n)$ рекурсивных вызовов; а так как на один вызов тратится постоянное время, имеем

(6.5) Для массива M с оптимальными значениями подзадач Find-Solution возвращает оптимальное решение за время $O(n)$.

6.2. Принципы динамического программирования: мемоизация или итерации с подзадачами

Теперь мы воспользуемся алгоритмом задачи взвешенного интервального планирования, разработанным в предыдущем разделе, для обобщения базовых принци-

пов динамического программирования. Кроме того, он поможет понять принцип, который играет ключевую роль в оставшейся части главы: итерации с подзадачами вместо рекурсивного вычисления решений.

В предыдущем разделе для построения решения задачи взвешенного интервального планирования с полиномиальным временем мы сначала разработали рекурсивный алгоритм с экспоненциальным временем, а затем преобразовали его (посредством мемоизации) в эффективный рекурсивный алгоритм, который обращался к глобальному массиву M за оптимальными решениями подзадач. Но чтобы понять, что здесь на самом деле происходит, желательно сформулировать практически эквивалентную версию алгоритма. Именно эта новая формулировка наиболее явно отражает суть метода динамического программирования и служит общим шаблоном для алгоритмов, которые будут разработаны в последующих разделах.

Разработка алгоритма

Ключом к построению эффективного алгоритма является массив M. В нем закодирована идея о том, что мы используем значение оптимальных решений подзадач для интервалов $\{1,2, ..., j\}$ по всем j, и он использует (6.1) для определения значения $M[j]$ на основании значений предшествующих элементов массива. Как только мы получаем массив M, задача решена: $M[n]$ содержит значение оптимального решения для всего экземпляра, а процедура `Find-Solution` может использоваться для эффективного обратного отслеживания по M и возвращения оптимального решения.

Важно понять, что элементы M можно напрямую вычислять итеративным алгоритмом вместо рекурсии с мемоизацией. Просто начнем с $M[0] = 0$ и будем увеличивать j; каждый раз, когда потребуется определить значение $M[j]$, ответ предоставляется (6.1). Алгоритм выглядит так:

Iterative-Compute-Opt

```
 M[0] = 0
   For j = 1,2, ..., n
   M[j] = max(v_j + M[p(j)], M[j − 1] )
Конец For
```

Анализ алгоритма

Следуя точной аналогии с доказательством (6.3), можно доказать посредством индукции по j, что этот алгоритм записывает `OPT`(j) в элемент массива $M[j]$; (6.1) предоставляет шаг индукции. Кроме того, как и ранее, `Find-Solution` можно передать заполненный массив M для получения оптимального решения помимо значения. Наконец, время выполнения `Iterative-Compute-Opt` очевидно равно $O(n)$, так как алгоритм явно выполняется в течение n итераций и проводит постоянное время в каждой из них.

Пример выполнения `Iterative-Compute-Opt` представлен на рис. 6.5. При каждой итерации алгоритм заполняет один дополнительный элемент массива M, сравнивая значение $v_j + M[p(j)]$ со значением $M[j-1]$.

Основная схема динамического программирования

Итак, описанная процедура дает второй эффективный алгоритм решения задачи взвешенного интервального планирования. Разумеется, на концептуальном уровне между двумя решениями существует много общего, поскольку оба решения обусловлены информацией, содержащейся в рекуррентном отношении (6.1). В оставшейся части этой главы мы будем разрабатывать алгоритмы динамического программирования с применением второго метода — итеративного построения подзадач, потому что алгоритмы часто проще выражаются таким образом. Но в каждом из рассматриваемых случаев существует эквивалентная формулировка алгоритма с мемоизированной рекурсией.

Что особенно важно, большую часть нашего обсуждения конкретной задачи выбора интервалов можно преобразовать в заготовку шаблона для разработки алгоритмов динамического программирования. Чтобы взяться за разработку алгоритма, основанного на методе динамического программирования, необходимо иметь набор подзадач, производных от исходной задачи, который обладает некоторыми базовыми свойствами.

(i) Количество подзадач ограничено полиномиальной зависимостью.

(ii) Решение исходной задачи легко вычисляется по решениям подзадач. (Например, исходная задача может быть одной из подзадач.)

(iii) Существует естественное упорядочение подзадач от «меньшей» к «большей» в совокупности с легко вычисляемой рекуррентностью (как в (6.1) и (6.2)), которая позволяет определить решение подзадачи по решениям некоторого количества меньших подзадач.

Естественно, это всего лишь неформальные ориентиры. В частности, понятие «меньшего» из части (iii) зависит от типа рекуррентности.

Вы увидите, что иногда процесс разработки такого алгоритма проще начать с формулировки естественного множества подзадач, а затем найти рекуррентное отношение, которое свяжет их воедино; но часто (как в случае со взвешенным интервальным планированием) полезнее сначала определить рекуррентное отношение на основе анализа структуры оптимального решения, а затем определить, какие подзадачи потребуются для его раскрутки. Связь между подзадачами и рекуррентными отношениями, в которой трудно отделить первопричину от следствия, является одной из тонких особенностей, лежащих в основе динамического программирования. Никогда нельзя быть уверенным в том, что набор подзадач будет полезным, пока не будет найдено рекуррентное отношение, связывающее подзадачи воедино; но о рекуррентном отношении трудно думать в отсутствие «меньших» подзадач, с которыми оно работает. В после-

дующих разделах будут описаны другие приемы, которые помогут справиться с этой неопределенностью.

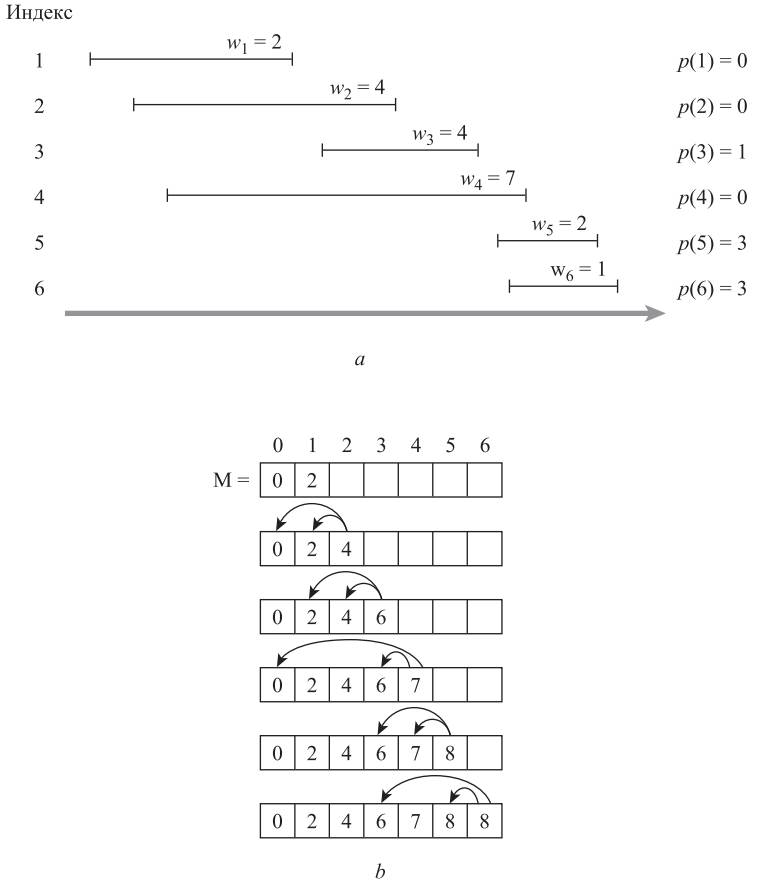


Рис. 6.5. В части *b* изображены итерации Iterative-Compute-Opt для экземпляра задачи взвешенного интервального планирования (*a*)

6.3. Сегментированные наименьшие квадраты: многовариантный выбор

Перейдем к другой категории задач, которая демонстрирует чуть более сложную разновидность динамического программирования. В предыдущем разделе рекуррентное отношение было разработанно на основе выбора, который был, по сути, бинарным: интервал *n* либо принадлежал оптимальному решению, либо не принадлежал ему. В задаче, рассматриваемой ниже, в рекуррентности будет задействовано то, что можно назвать «многовариантным выбором»: на каждом шаге существует

полиномиальное количество вариантов, которые могут рассматриваться как кандидаты для структуры оптимального решения. Как вы увидите, метод динамического программирования очень естественно адаптируется к этой более общей ситуации.

Отдельно стоит сказать о том, что задача этого раздела также хорошо показывает, как четкое алгоритмическое определение может формализовать понятие, изначально казавшееся слишком нечетким и нелогичным для математического представления.

Задача

При работе с научными и статистическими данными, нанесенными на плоскость, аналитики часто пытаются найти «линию наилучшего соответствия» для этих данных (рис. 6.6).

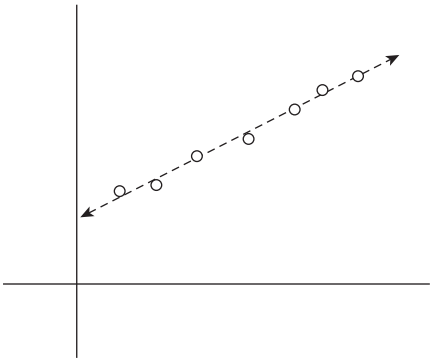


Рис. 6.6. Линия наилучшего соответствия

Эта основополагающая задача из области статистики и вычислительной математики формулируется следующим образом. Допустим, данные представляют собой множество P из n точек на плоскости, обозначаемых (x_1, y_1), (x_2, y_2), ..., (x_n, y_n); предполагается, что $x_1 < x_2 < \ldots < x_n$. Для линии L, определяемой уравнением $y = ax + b$, *погрешностью* L относительно P называется сумма возведенных в квадрат «расстояний» от точек до P:

$$\mathrm{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

Естественной целью в такой ситуации будет нахождение линии с минимальной погрешностью. Оказывается, у этой задачи существует компактное решение, которое легко вычисляется методами математического анализа. Опуская промежуточные выкладки, приведем результат: линия минимальной погрешности определяется формулой $y = ax + b$, где

$$a = \frac{n\sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n\sum_i x_i^2 - \left(\sum_i x_i\right)^2} \text{ и } b = \frac{\sum_i y_i - a\sum_i x_i}{n}.$$

Но существует проблема, для которой эти формулы не подойдут. Часто набор данных выглядит примерно так, как показано на рис. 6.7. В таком случае нам хотелось бы сделать утверждение вида «Точки лежат приблизительно на серии из двух линий». Как формализовать такое понятие?

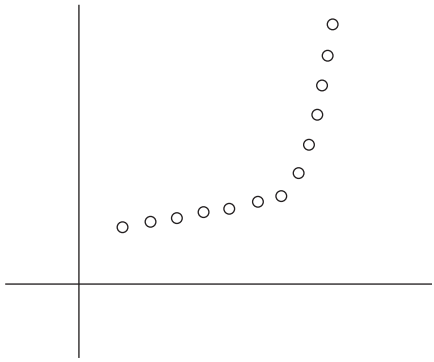


Рис. 6.7. Набор точек, расположенных приблизительно вдоль двух линий

Для любой одиночной линии погрешность у точек на рисунке будет запредельной; но если использовать две линии, ошибка может быть относительно невелика. Итак, новую задачу можно попытаться сформулировать следующим образом: вместо того, чтобы искать одну линию наилучшего соответствия, мы ищем набор линий, обеспечивающий минимальную погрешность. Но такая формулировка задачи никуда не годится, потому что у нее имеется тривиальное решение: при сколь угодно большом наборе линий можно добиться идеального соответствия, соединив линиями каждую пару последовательных точек в P.

С противоположной стороны можно жестко «запрограммировать» число 2 в задаче — искать лучшее соответствие не более чем с двумя линиями. Но и такая формулировка противоречит здравому смыслу: изначально не было никакой априорной информации о том, что точки лежат приблизительно на двух линиях; мы пришли к такому выводу, взглянув на иллюстрацию. Вероятно, большинство людей также скажет, что точки на рис. 6.8 лежат приблизительно на трех линиях.

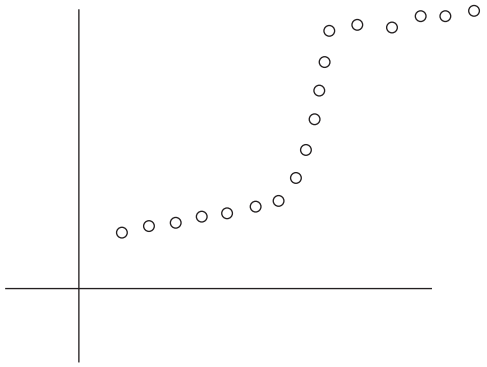


Рис. 6.8. Набор точек, расположенных приблизительно вдоль трех линий

На интуитивном уровне формулировка задачи должна обеспечивать хорошее прилегание точек с минимально возможным количеством линий. А теперь сформулируем *сегментированную задачу наименьших квадратов*, которая достаточно четко отражает эти представления. Эта задача является конкретным воплощением более общей проблемы из области анализа данных и статистики, называемой *обнаружением изменений*: в заданной последовательности точек данных требуется найти несколько точек, в которых происходят дискретные *изменения* (в нашем случае — переход к другой линейной аппроксимации).

Формулировка задачи

Как и в приведенном выше описании, имеется множество точек $P = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, в котором $x_1 < x_2 < ... < x_n$. Точка (x_i, y_i) будет обозначаться p_i. Сначала необходимо разбить P на некоторое количество сегментов. Каждый сегмент является подмножеством P, представляющим непрерывный набор координат x; другими словами, это подмножество вида $\{p_i, p_{i+1}, ..., p_{j-1}, p_j\}$ для некоторых индексов $i \le j$. Затем для каждого сегмента S в разбиении P вычисляется линия, минимизирующая погрешность в отношении точек S для приведенных выше формул.

Штраф разбиения определяется как сумма следующих слагаемых:

(i) Количество сегментов, на которые разбивается P, умноженное на фиксированный множитель $C > 0$.

(ii) Для каждого сегмента — значение погрешности для оптимальной линии через этот сегмент.

Нашей целью в сегментированной задаче наименьших квадратов является нахождение разбиения с минимальным штрафом. Минимизация этой характеристики отражает компромиссы, упоминавшиеся ранее. При поиске решения могут рассматриваться разбиения на любое количество сегментов; с увеличением количества сегментов уменьшаются слагаемые штрафа из части (ii) определения, но увеличивается слагаемое в части (i). (Множитель C предоставляется со входными данными; настраивая C, можно увеличить или уменьшить штраф за использование дополнительных линий.)

Количество возможных разбиений в P растет по экспоненте, и изначально непонятно, возможен ли эффективный поиск оптимального разбиения. Сейчас мы покажем, как использовать динамическое программирование для нахождения минимального штрафа за время, полиномиальное по n.

Разработка алгоритма

Для начала вспомним, какие ингредиенты необходимы для алгоритма динамического программирования (см. в конце раздела 6.2). Это полиномиальное количество подзадач, решения которых должны дать решение исходной задачи; и решения этих подзадач должны строиться с использованием рекуррентного отношения. Как и в случае задачи взвешенного интервального планирования, полезно поразмыслить над некоторыми простыми свойствами оптимального решения. Однако

учтите, что прямой аналогии с задачей взвешенного интервального планирования здесь нет: в той задаче мы искали подмножество из n объектов, а здесь ищется способ разбиения n объектов.

Для сегментированной задачи наименьших квадратов очень полезно следующее наблюдение: последняя точка p_n принадлежит одному сегменту оптимального разбиения, и этот сегмент начинается в некоторой более ранней точке p_i. Подобные наблюдения могут подсказать правильное множество подзадач: зная последний сегмент p_i, ..., p_n (рис. 6.9), мы можем исключить эти точки из рассмотрения и рекурсивно решить задачу для оставшихся точек p_1, ..., p_{i-1}.

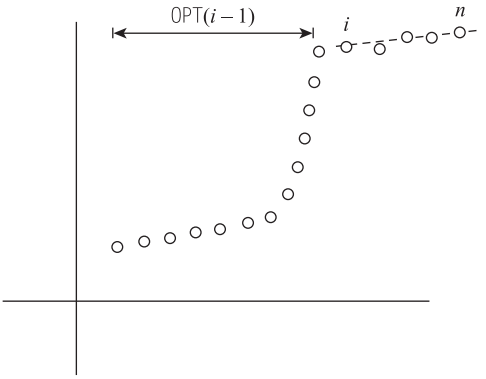


Рис. 6.9. Возможное решение: подбираем один сегмент для точек $p_i, p_{i+1}, \ldots, p_n$, после чего переходим к поиску оптимального решения для оставшихся точек $p_1, p_2, \ldots, p_{i-1}$

Предположим, $\mathrm{OPT}(i)$ обозначает оптимальное решение для точек p_1, ..., p_i, а $e_{i,j}$ — минимальную погрешность для любой линии в отношении p_i, p_{i+1}, ..., p_j. (Запись $\mathrm{OPT}(0) = 0$ используется как граничный случай.) Тогда приведенное выше наблюдение означает следующее:

(6.6) Если последний сегмент оптимального решения состоит из точек p_i, ..., p_n, то значение оптимального решения равно $\mathrm{OPT}(n) = e_{i,n} + C + \mathrm{OPT}(i-1)$.

Используя то же наблюдение для подзадачи, состоящей из точек p_1, ..., p_j, мы видим, что для получения $\mathrm{OPT}(j)$ необходимо найти лучший способ получения завершающего сегмента p_i, ..., p_j — с оплатой погрешности и слагаемого C для этого сегмента — в сочетании с оптимальным решением $\mathrm{OPT}(i-1)$ для остальных точек. Другими словами, мы обосновали следующее рекуррентное отношение:

(6.7) Для подзадачи с точками p_1, ..., p_j

$$\mathrm{OPT}(j) = \min_{1 \le i \le j} (e_{i,j} + C + \mathrm{OPT}(i-1)),$$

а сегмент p_i, ..., p_j используется в оптимальном решении подзадачи в том и только в том случае, если минимум достигается при использовании индекса i.

Самая сложная часть разработки алгоритма осталась позади. Далее можно просто строить решения $\mathrm{OPT}(i)$ в порядке возрастания i.

```
Segmented-Least-Squares(n)
   Массив M[0... n]
   Присвоить M[0] = 0
   Для всех пар i ≤ j
      Вычислить наименьшую квадратичную погрешность e_{i,j}
         для сегмента p_i, ..., p_j
   Конец цикла
   For j = 1,2, ..., n
      Использовать рекуррентное отношение (6.7) для вычисления M[j]
   Конец For
   Вернуть M[n]
```

По аналогии с рассуждениями для взвешенного интервального планирования правильность этого алгоритма напрямую доказывается посредством индукции ((6.7) предоставляет шаг индукции).

Как и в алгоритме взвешенного интервального планирования, оптимальное разбиение вычисляется обратным отслеживанием по M.

```
Find-Segments(j)
   Если j = 0
      Ничего не выводить
   Иначе
      Найти значение i, минимизирующее e_{i,j} + C + M[i − 1]
      Вывести сегмент {p_i, ..., p_j} и результат Find-Segments(i − 1)
   Конец Если
```

Анализ алгоритма

Осталось проанализировать время выполнения `Segmented-Least-Squares`. Сначала необходимо вычислить значения всех погрешностей наименьших квадратов $e_{i,j}$. Чтобы учесть время выполнения, расходуемое на эти вычисления, заметим, что существуют $O(n^2)$ пар (i,j), для которых эти вычисления необходимы; для каждой пары (i,j) можно использовать формулу, приведенную в начале этого раздела, для вычисления $e_{i,j}$ за время $O(n)$. Следовательно, общее время выполнения для вычисления всех значений $e_{i,j}$ равно $O(n^3)$.

Алгоритм состоит из n итераций для значений $j = 1, ..., n$. Для каждого значения j необходимо определить минимум в рекуррентном отношении (6.7) для заполнения элемента массива $M[j]$; это занимает время $O(n)$ для каждого j, что в сумме дает $O(n^2)$. Итак, после определения всех значений $e_{i,j}$ время выполнения равно $O(n^2)$[1].

[1] В этом анализе во времени выполнения доминирует фактор $O(n^3)$, необходимый для вычисления всех значений $e_{i,j}$. Но на самом деле все эти значения можно вычислить за время $O(n_2)$, что сокращает время выполнения всего алгоритма до $O(n^2)$. Идея, подробности которой остаются читателю для самостоятельной работы, заключается в том, чтобы сначала вычислить $e_{i,j}$ для всех пар (i,j), для которых $j - i = 1$, затем для всех пар с $j - i = 2$, затем для $j - i = 3$ и т. д. Таким образом, добравшись до конкретного значения $e_{i,j}$, мы можем использовать ингредиенты вычисления $e_{i,j-1}$ для определения $e_{i,j}$ за постоянное время.

6.4. Задача о сумме подмножеств и задача о рюкзаке: добавление переменной

Мы видим все больше примеров того, что область планирования предоставляет богатый источник алгоритмических задач, обладающих практическим смыслом. Ранее мы рассматривали задачи, в которых заявки задаются интервалом времени, а также задачи, в которых для заявки определяется продолжительность и предельное время, но не указывается конкретный интервал времени, в течение которого они должны выполняться.

В этом разделе рассматривается разновидность задач второго типа, с продолжительностями и предельным временем, которую трудно решать напрямую с использованием методов, применявшихся ранее. Для решения задачи будет применен метод динамического программирования, но с небольшими изменениями: «очевидного» множества подзадач оказывается недостаточно, и нам придется создавать расширенный набор подзадач. Как вы вскоре увидите, для этого в рекуррентное отношение, лежащее в основе динамической программы, будет добавлена новая переменная.

Задача

В рассматриваемой задаче планирования имеется одна машина, способная обрабатывать задания, и набор заявок $\{1,2, ..., n\}$. Ресурсы могут обрабатываться только в период времени от 0 до W (для некоторого числа W). Каждая заявка представляет собой задание, для обработки которого требуется время w_i.

Если нашей целью является обработка заданий, при которой обеспечивается максимальная занятость машины до «граничного времени» W, какие задания следует выбрать?

В более формальном виде: имеются n элементов $\{1, ..., n\}$, каждому из которых присвоен неотрицательный вес w_i (для $i = 1, ..., n$). Также задана граница W. Требуется выбрать подмножество S элементов, для которого $\sum_{i \in S} w_i \leq W$, чтобы этого ограничения значение $\sum_{i \in S} w_i$ было как можно больше. Эта задача называется *задачей о сумме подмножеств*.

Она является естественным частным случаем более общей задачей, называемой *задачей о рюкзаке*, в которой у каждого элемента i имеется как значение v_i, так и вес w_i. Целью в этой общей задачи является нахождение подмножества с максимальным суммарным значением, при котором общий вес не превышает W. Название «задача о рюкзаке» происходит от аналогии с наполнением рюкзака емкостью W на максимальную часть объема (или максимальной стоимостью груза) с использованием подмножества элементов $\{1, ..., n\}$. При упоминании величин w_i и W мы будем использовать термины «вес» и «время».

Поскольку ситуация напоминает другие задачи планирования, уже встречавшиеся ранее, естественно спросить, можно ли найти оптимальное решение при по-

мощи жадного алгоритма. Похоже, ответ на этот вопрос отрицателен — по крайней мере, не известно никакое эффективное жадное правило, которое всегда строит оптимальное решение. Один из естественных жадных методов основан на сортировки элементов по убыванию веса (или по крайней мере для всех элементов с весом, не превышающим W), а затем выборе элементов в этом порядке, пока общий вес остается меньше W. Но если значение W кратно 2, и имеются три элемента с весами $\{W/2 + 1, W/2, W/2\}$, этот жадный алгоритм не обеспечит оптимального решения. Также можно провести сортировку по возрастанию веса, а затем проделать то же самое; но этот способ не работает для входных данных вида $\{1, W/2, W/2\}$.

В этом разделе мы постараемся показать, как применить динамическое программирование для решения задачи. Вспомните основные принципы динамического программирования: нужно перейти к небольшому количеству подзадач, чтобы каждая подзадача легко решалась по «меньшим» подзадачам, а решение исходной задачи легко строилось по известным решениям всех подзадач. Проблема в том, чтобы определить хорошее множество подзадач.

Разработка алгоритма

Неудачная попытка

В случае взвешенного интервального планирования сработала общая стратегия с рассмотрением подзадач, в которых задействованы только первые i заявок. Попробуем применить эту стратегию в данном случае. Лучшее возможное решение, использующее подмножество заявок $\{1,...,i\}$, будет обозначаться $\texttt{OPT}(i)$ (как это делалось ранее). В задаче взвешенного интервального планирования ключевой момент заключался в том, чтобы сосредоточиться на оптимальном решении O нашей задачи и рассмотреть два случая в зависимости от того, принимается или отвергается последняя заявка n этим оптимальным решением. Как и в том случае, одна из частей следует непосредственно из определения $\texttt{OPT}(i)$.

- Если $n \notin O$, то $\texttt{OPT}(n) = \texttt{OPT}(n-1)$.

Затем необходимо рассмотреть случай, в котором $n \in O$. Хотелось бы иметь простую рекурсию, которая сообщит лучшее возможное значение для решений, содержащих последнюю заявку n. Для взвешенного интервального планирования это было несложно, так как мы могли просто удалить каждую заявку, конфликтующую с n. В текущей задаче дело обстоит сложнее. Из принятия заявки n не следует, что какую-то другую заявку нужно отклонить, — а лишь то, что для подмножества принимаемых заявок $S \subseteq \{1, ..., n-1\}$ остается меньше свободного веса: вес w_n используется для принятой заявки n, так что для множества S остальных принимаемых заявок остается только вес $W - w_n$ (рис. 6.10).

Улучшенное решение

Все это наводит на мысль, что решение потребует больше подзадач: для определения значения $\texttt{OPT}(n)$ необходимо знать не только значение $\texttt{OPT}(n-1)$, но и лучшее

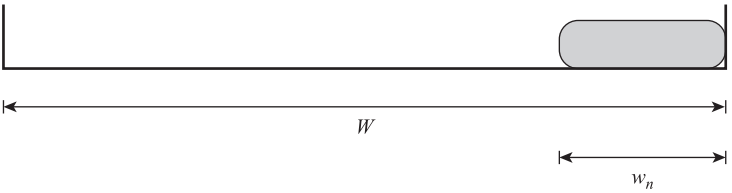


Рис. 6.10. При включении в решение элемента n вес w_n занят и для остальных заявок остается доступный вес $W-w_n$

решение, которое может быть получено с использованием подмножества первых $n-1$ элементов и общего доступного веса $W-w_n$. А следовательно, подзадач будет намного больше: по одной для каждого исходного множества $\{1, ..., i\}$ элементов и для каждого возможного значения оставшегося доступного веса w. Предположим, W — целое число, а все запросы $i = 1, ..., n$ имеют целые веса w_i. Тогда подзадача создается для каждого $i = 0,1, ..., n$ и каждого целого числа $0 \le w \le W$. Для оптимального решения, использующего подмножество элементов $\{1,..., i\}$ с максимально допустимым весом w, будет использоваться решение $\mathtt{OPT}(i, w)$, то есть

$$\mathtt{OPT}(i, w) = \max_S \sum_{j \in S} w_j,$$

где максимум определяется по подмножествам $S \subseteq \{1, ..., i\}$, удовлетворяющим условию $\sum_{j \in S} w_j \le w$. С этим новым множеством подзадач мы сможем представить значение $\mathtt{OPT}(i, w)$ в виде простого выражения, использующего значения меньших подзадач. Более того, $\mathtt{OPT}(n, W)$ — значение, которое мы ищем в конечном итоге. Как и прежде, пусть O обозначает оптимальное решение исходной задачи.

- Если $n \notin O$, то $\mathtt{OPT}(n, W) = \mathtt{OPT}(n-1, W)$, так как элемент n можно просто игнорировать.
- Если $n \in O$, то $\mathtt{OPT}(n, W) = w_n + \mathtt{OPT}(n-1, W-w_n)$, так как теперь ищется вариант оптимального использования оставшейся емкости $W - w_n$ между элементами $1, 2, ..., n-1$.

Когда n-й элемент слишком велик, то есть $W < w_n$, должно выполняться условие $\mathtt{OPT}(n, W) = \mathtt{OPT}(n-1, W)$. В противном случае мы получаем оптимальное решение, допускающее все n заявок, выбирая лучший из этих двух вариантов. Применяя аналогичные рассуждения для подзадачи с элементами $\{1, ..., i\}$ и максимальным допустимым весом w, получаем следующее рекуррентное отношение:

(6.8) Если $w < w_i$, то $\mathtt{OPT}(i, w) = \mathtt{OPT}(i-1, w)$. В противном случае $\mathtt{OPT}(i, w) = \max(\mathtt{OPT}(i-1, w), w_i + \mathtt{OPT}(i-1, w-w_i))$.

Как и прежде, мы хотим создать алгоритм, который строит таблицу всех значений $\mathtt{OPT}(i, w)$, вычисляя каждое из них не более одного раза.

```
Subset-Sum(n, W)
  Массив M[0...n, 0...W]
  Инициализировать M[0, w] = 0 для всех w = 0,1, ..., W
  For i = 1,2, ..., n
```

```
    For w = 0, ..., W
      Использовать рекуррентное отношение (6.8) для вычисления M[i,w]
    Конец For
  Конец For
  Вернуть M[n, W]
```

При помощи (6.8) можно немедленно доказать посредством индукции, что возвращаемое значение $M[n, W]$ является значением оптимального решения для заявок 1, ..., n и доступного веса W.

Анализ алгоритма

Вспомните одномерную последовательность ячеек для задачи взвешенного интервального планирования на рис. 6.5, где был представлен способ итеративного заполнения массива M для этого алгоритма. Для только что разработанного алгоритма можно использовать аналогичное представление, но на этот раз понадобится двумерная таблица, отражающая двумерный массив подзадач. На рис. 6.11 изображено построение подзадач в этой ситуации: значение $M[i, w]$ вычисляется по двум другим значениям $M[i-1, w]$ и $M[i-1, w-w_i]$.

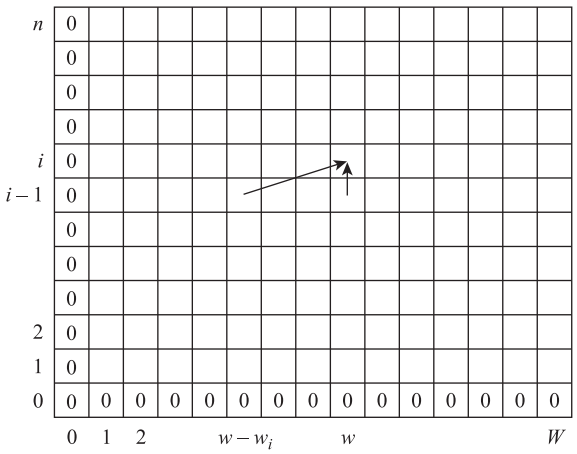


Рис. 6.11. Двумерная таблица значений OPT. Крайний левый столбец и нижняя строка всегда содержат 0. Значение OPT(i, w) вычисляется по двум другим элементам — OPT($i-1$, w) и OPT($i-1$, $w-w_i$) в соответствии со стрелками

В качестве примера выполнения этого алгоритма рассмотрим экземпляр с предельным весом $W = 6$ и $n = 3$ элементами с размерами $w_1 = w_2 = 2$ и $w_3 = 3$. Оптимальное значение OPT(3, 6) = 5 достигается при использовании третьего элемента и одного из первых двух элементов. На рис. 6.12 продемонстрирован процесс заполнения алгоритмом двумерной таблицы значениями OPT строка за строкой.

Следующий вопрос — время выполнения алгоритма. Как и прежде в задаче взвешенного интервального планирования, мы строим таблицу решений M

и вычисляем каждое из значений $M[i, w]$ за время $O(1)$ с использованием предыдущих значений. Следовательно, время выполнения пропорционально количеству элементов в таблице.

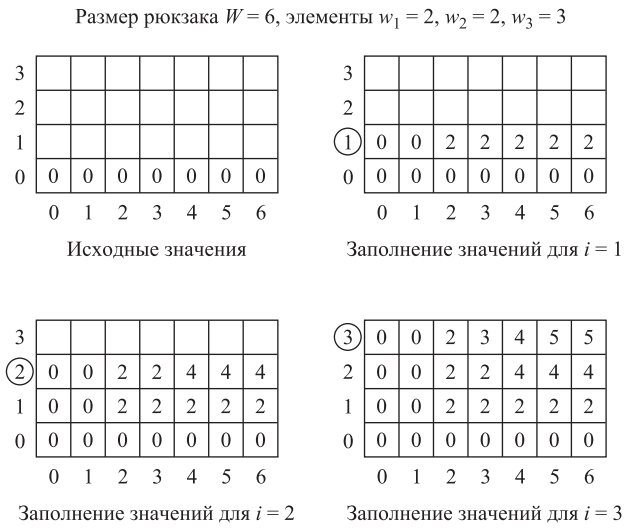


Рис. 6.12. Итерации алгоритма для простого экземпляра задачи о сумме подмножеств

(6.9) Алгоритм `Subset-Sum`(n, W) правильно вычисляет оптимальное значение для задачи и выполняется за время $O(nW)$.

Обратите внимание: этот метод менее эффективен, чем динамическая программа для задачи взвешенного интервального планирования. В самом деле, его время выполнения не является полиномиальной функцией n; это полиномиальная функция n и W — наибольшего целого, задействованного в определении задачи. Такие алгоритмы называются *псевдополиномиальными*. Псевдополиномиальные алгоритмы могут обладать неплохой эффективностью при достаточно малых числах $\{w_i\}$ во входных данных; с увеличением этих чисел их практическая применимость снижается.

Чтобы получить оптимальное множество элементов S, можно выполнить обратное отслеживание по массиву M по аналогии с тем, как это делалось в предыдущих разделах:

(6.10) Для таблицы M оптимальных значений подзадач оптимальное множество S может быть найдено за время $O(n)$.

Расширение: задача о рюкзаке

Задача о рюкзаке немного сложнее задачи планирования, которая рассматривалась нами ранее. Возьмем ситуацию, в которой каждому элементу i поставлен в соответствие неотрицательный вес w_i, как и прежде, и отдельное значение v_i. Цель —

найти подмножество S с максимальным значением $\sum_{i \in S} v_i$, в котором общий вес множества не превышает W: $\sum_{i \in S} w_i \leq W$.

Алгоритм динамического программирования несложно расширить до этой более общей задачи. Аналогичное множество подзадач $\texttt{OPT}(i, w)$ используется для представления значения оптимального решения с использованием подмножества элементов $\{1, ..., i\}$ и максимального доступного веса w. Рассмотрим оптимальное решение O и определим два случая в зависимости от того, выполняется ли условие $n \in O$:

- Если $n \notin O$, то $\texttt{OPT}(n, W) = \texttt{OPT}(n - 1, W)$.
- Если $n \in O$, то $\texttt{OPT}(n, W) = v_n + \texttt{OPT}(n - 1, W - w_n)$.

Применение этой аргументации к подзадачам приводит к следующей аналогии с (6.8).

(6.11) Если $w < w_i$, то $\texttt{OPT}(i, w) = \texttt{OPT}(i - 1, w)$. В противном случае $\texttt{OPT}(i, w) = \max(\texttt{OPT}(i - 1, w), v_i + \texttt{OPT}(i - 1, w - w_i))$.

При помощи этого рекуррентного отношения можно записать полностью аналогичный алгоритм динамического программирования, из чего вытекает следующий факт:

(6.12) Задача о рюкзаке решается за время $O(nW)$.

6.5. Вторичная структура РНК: динамическое программирование по интервалам

В задаче о рюкзаке нам удалось сформулировать алгоритм динамического программирования при добавлении новой переменной. Существует и другой очень распространенный сценарий, в котором в динамическую программу добавляется переменная. Мы начинаем с рассмотрения множества подзадач $\{1, 2, ..., j\}$ для всех возможных j, но найти естественное рекуррентное отношение не удается. Тогда мы рассматриваем большее множество подзадач из $\{i, i + 1, ..., j\}$ для всех возможных i и j (где $i \leq j$) и находим естественное рекуррентное отношение для этих подзадач. Таким образом, в задачу добавляется вторая переменная i; в результате рассматривается подзадача для каждого непрерывного интервала в $\{1, 2, ..., n\}$.

Существует ряд канонических задач, соответствующих этому описанию; читатели, изучавшие алгоритмы разбора для контекстно-независимых грамматик, вероятно, видели хотя бы один алгоритм динамического программирования в этом стиле. А здесь мы сосредоточимся на задаче предсказания вторичной структуры РНК, одной из фундаментальных проблем в области вычислительной биологии.

Задача

Как известно из начального курса биологии, Уотсон и Крик установили, что двухцепочечная ДНК «связывается» комплементарным спариванием оснований. Каждая цепь ДНК может рассматриваться как цепочка *оснований*, каждое из которых выбирается из множества $\{A, C, G, T\}$[1]. Пары образуются как основаниями A и T, так и основаниями C и G; именно связи A-T и C-G удерживают две цепочки вместе.

Одноцепочечные молекулы РНК являются ключевыми компонентами многих процессов, происходящих в клетках, и строятся по более или менее одинаковым структурным принципам. Однако в отличие от двухцепочечной ДНК, у РНК не существует «второй цепи» для связывания, поэтому РНК обычно образует пары оснований сама с собой; это приводит к образованию интересных форм наподобие изображенной на рис. 6.13. Совокупности пар (и полученная форма), образованные молекулой РНК в этом процессе, называются *вторичной структурой*; понимание вторичной структуры необходимо для понимания поведения молекулы.

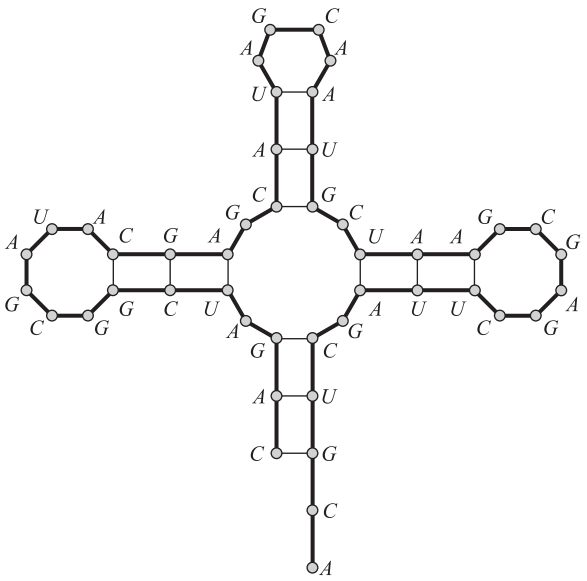


Рис. 6.13. Вторичная структура РНК. Жирными линиями обозначаются смежные элементы последовательности, а тонкими — элементы, образующие пары

Для наших целей одноцепочечная молекула РНК может рассматриваться как последовательность из n символов (оснований) алфавита $\{A, C, G, U\}$[2]. Пусть $B = b_1 b_2 \ldots b_n$ — одноцепочечная молекула РНК, в которой все $b_i \in \{A, C, G, U\}$. В первом приближении вторичная структура может моделироваться следующим

[1] Аденин, гуанин, цитозин и тимин — четыре основные структурные единицы ДНК.

[2] Заметьте, что символ T из алфавита ДНК заменяется символом U, но для нас это сейчас несущественно.

образом. Как обычно, основание A образует пары с U, а основание C образует пары с G; также требуется, чтобы каждое основание могло образовать пару с не более чем одним другим основанием, иначе говоря, множество пар оснований образует *паросочетание*. Также вторичные структуры (снова в первом приближении) не образуют узлов, что ниже формально определяется как своего рода условие отсутствия пересечений.

А если более конкретно, вторичная структура B представляет собой множество пар $S = \{(i, j)\}$, где $i, j \in \{1, 2, ..., n\}$, удовлетворяющих следующим условиям:

(i) (Отсутствие крутых поворотов) Концы каждой пары в S разделяются минимум четырьмя промежуточными основаниями; иначе говоря, если $(i, j) \in S$, то $i < j - 4$.

(ii) Любая пара в S состоит из элементов $\{A, U\}$ или $\{C, G\}$ (в произвольном порядке).

(iii) S является паросочетанием; никакое основание не входит более чем в одну пару.

(iv) (Отсутствие пересечений) Если (i, j) и (k, l) — две пары в S, то невозможна ситуация $i < k < j < l$ (рис. 6.14).

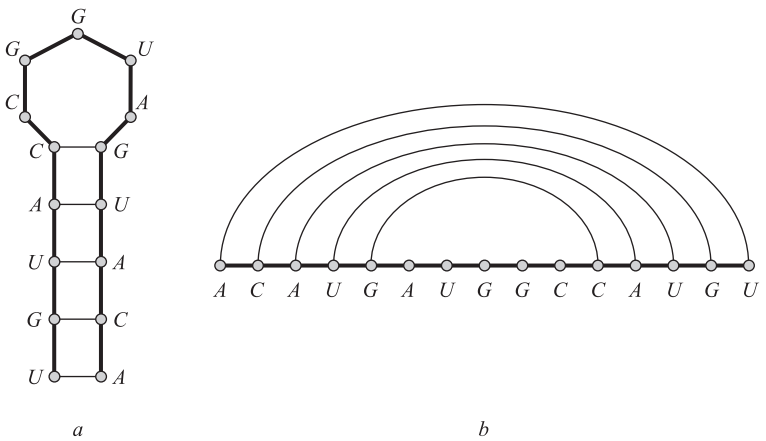


Рис. 6.14. Два представления вторичной структуры РНК. Во втором представлении (*b*) цепочка «вытянута в длину», а отрезки, соединяющие пары, выглядят как непересекающиеся «купола» над цепочкой

Вторичная структура РНК на рис. 6.13 удовлетворяет свойствам (i)–(iv). Со структурной точки зрения условие (i) выполняется просто из-за того, что молекула РНК не может слишком резко изгибаться, а условия (ii) и (iii) определяются фундаментальными правилами сочетания оснований Уотсона–Крика. С условием (iv) дело обстоит сложнее: непонятно, почему оно должно выполняться в природе. Но хотя в реальных молекулах встречаются отдельные исключения (так называемые *псевдоузлы*), как выясняется, это ограничение хорошо аппроксимирует пространственные ограничения реальных вторичных структур РНК.

Итак, из всего множества вторичных структур, возможных для одной молекулы РНК, какие конфигурации могут возникнуть в физиологических условиях? Обычно предполагается, что одноцепочечная молекула РНК образует вторичную структуру с оптимальной общей свободной энергией. На тему правильной модели свободной энергии вторичной структуры идут ожесточенные споры; но мы будем в первом приближении считать, что свободная энергия вторичной структуры пропорциональна количеству содержащихся в ней пар оснований.

После всего сказанного базовая задача предсказания вторичной структуры РНК формулируется очень просто: требуется найти эффективный алгоритм, который получает одноцепочечную молекулу РНК $B = b_1 b_2 \ldots b_n$ и определяет вторичную структуру S с максимально возможным количеством пар оснований.

Разработка и анализ алгоритма

Динамическое программирование: первая попытка

Вероятно, первая естественная попытка применения динамического программирования будет основана на следующих подзадачах: мы говорим, что $\texttt{OPT}(j)$ — максимальное количество пар оснований во вторичной структуре $b_1, b_2, \ldots, b_j$. Согласно приведенному выше запрету на резкие повороты, мы знаем, что $\texttt{OPT}(j) = 0$ для $j \leq 5$; также известно, что $\texttt{OPT}(n)$ — искомое решение.

Проблемы начинаются тогда, когда мы пытаемся записать рекуррентное отношение, выражающее $\texttt{OPT}(j)$ в контексте решений меньших подзадач. Здесь можно пройти часть пути: в оптимальной вторичной структуре $b_1, b_2, \ldots, b_j$ возможно одно из двух:

- j не участвует в паре; или
- j участвует в паре с t для некоторого $t < j - 4$.

В первом случае необходимо обратиться к решению для $\texttt{OPT}(j-1)$. Второй случай изображен на рис. 6.15, *a*; из-за запрета на пересечения мы знаем, что не может существовать пары, один конец которой находится между 1 и $t-1$, а другой — между $t+1$ и $j-1$. Таким образом, мы фактически выделили две новые подзадачи: для оснований $b_1 b_2 \ldots b_{t-1}$ и для оснований $b_{t+1} \ldots b_{j-1}$. Первая решается как $\texttt{OPT}(t-1)$, а вторая в наш список подзадач не входит, потому что она не начинается с b_1.

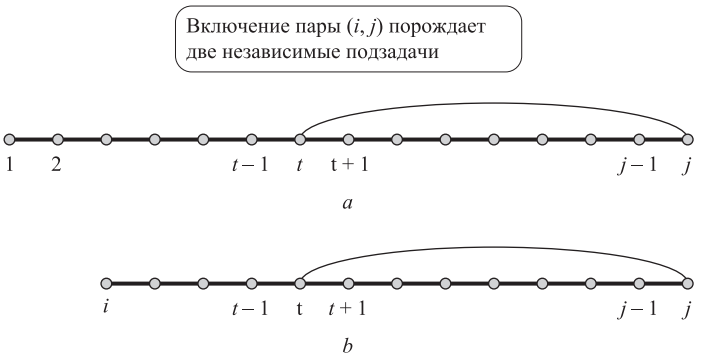


Рис. 6.15. Схематическое представление рекуррентности динамического программирования с использованием (*a*) одной переменной, (*b*) двух переменных

Это соображение наводит на мысль о том, что в задачу следует добавить переменную. Нужно иметь возможность работать с подзадачами, не начинающимися с b_1; другими словами, необходимо иметь возможность рассматривать подзадачи для $b_i b_{i+1} \ldots b_j$ при любых $i \le j$.

Динамическое программирование по интервалам

После принятия этого решения приведенные выше рассуждения ведут прямо к успешной рекурсии. Пусть $\mathtt{OPT}(i, j)$ — максимальное количество пар оснований во вторичной структуре $b_i b_{i+1} \ldots b_j$. Запрет на резкие повороты позволяет инициализировать $\mathtt{OPT}(i, j) = 0$ для всех $i \ge j - 4$.

(Для удобства записи мы также разрешим ссылки $\mathtt{OPT}(i, j)$ даже при $i > j$; в этом случае значение равно 0).

Теперь в оптимальной вторичной структуре $b_i b_{i+1} \ldots b_j$ существуют те же альтернативы, что и прежде:

- j не участвует в паре; или
- j находится в паре с t для некоторого $t < j - 4$.

В первом случае имеем $\mathtt{OPT}(i, j) = \mathtt{OPT}(i, j - 1)$. Во втором случае, изображенном на рис. 6.15, *b*, порождаются две подзадачи $\mathtt{OPT}(i, t - 1)$ и $\mathtt{OPT}(t + 1, j - 1)$; как было показано выше, эти две задачи изолируются друг от друга условием отсутствия пересечений.

Следовательно, мы только что обосновали следующее рекуррентное отношение:

(6.13) $\mathtt{OPT}(i, j) = \max(\mathtt{OPT}(i, j - 1), \max(1 + \mathtt{OPT}(i, t - 1) + \mathtt{OPT}(t + 1, j - 1)))$, где max определяется по t, для которых b_t и b_j образуют допустимую пару оснований (с учетом условий (i) и (ii) из определения вторичной структуры).

Теперь нужно убедиться в правильном понимании порядка построения решений подзадач. Из (6.13) видно, что решения подзадач всегда вызываются для *более коротких* интервалов: тех, для которых $k = j - i$ меньше. Следовательно, схема будет работать без каких-либо проблем, если решения строятся в порядке возрастания длин интервалов.

```
Инициализировать OPT(i,j) = 0 для всех i ≥ j–4
For k = 5, 6, ..., n – 1
   For i = 1,2, ..., n – k
      Присвоить j = i + k
      Вычислить OPT(i,j) с использованием рекуррентного отношения из (6.13)
   Конец For
Конец For
Вернуть OPT(1,n)
```

Рассмотрим пример выполнения этого алгоритма для входных данных *ACCGGUAGU* (рис. 6.14). Как и в задаче о рюкзаке, для представления массива M понадобятся два массива: для левой и для правой конечных точек рассматриваемого интервала. На рисунке представлены только элементы, соответствующие парам $[i, j]$ с $i < j - 4$, потому что только они могут быть отличны от нуля.

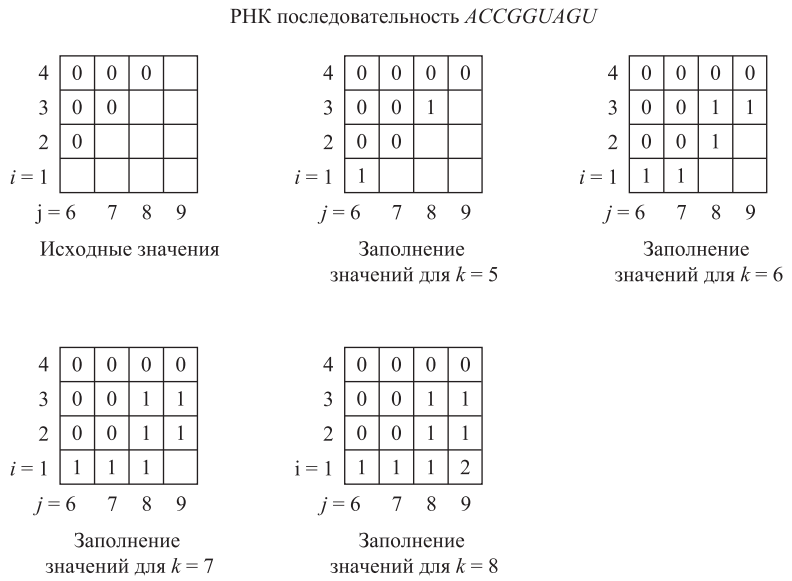


Рис. 6.16. Итерации алгоритма для конкретного экземпляра задачи предсказания вторичной структуры РНК

Найти границу для времени выполнения несложно: существуют $O(n^2)$ подзадач, которые необходимо решить, а вычисление рекуррентного отношения в (6.13) занимает время $O(n)$ для каждой задачи. Следовательно, время выполнения будет равно $O(n^3)$.

Как обычно, саму вторичную структуру (а не только ее значение) можно восстановить сохранением информации о достижении минимума в (6.13) и обратного отслеживания по вычислениям.

6.6. Выравнивание последовательностей

В оставшейся части главы рассматриваются еще два алгоритма динамического программирования, находящие широкое практическое применение. В следующих двух разделах рассматривается *выравнивание последовательностей* — фундаментальная задача, встречающаяся при сравнении строк. Затем мы обратимся к задаче вычисления кратчайших путей в графах, которые могут содержать ребра с отрицательной стоимостью.

Задача

Словари в Интернете становятся все более удобными: часто бывает проще открыть сетевой словарь по закладке, чем снимать бумажный словарь с полки. К тому же многие электронные словари предоставляют возможности, отсутствующие в бу-

мажных словарях: если вы ищете определение и введете слово, отсутствующее в словаре (например, *ocurrance*), словарь поинтересуется: «Возможно, вы имели в виду *occurrence*?» Как он это делает? Он действительно знает, что происходит у вас в голове?

Отложим второй вопрос до другой книги, а пока поразмыслим над первым. Чтобы решить, что вы могли иметь в виду, наиболее естественно провести в словаре поиск слова, наиболее «похожего» на введенное. Для этого нужно ответить на вопрос: как определить сходство между двумя словами или строками?

Интуитивно понятно, что «ocurrance» и «occurrence» похожи, потому что для совпадения этих двух слов достаточно добавить *c* в первое слово и заменить *a* на *e*. Так как ни одно из этих изменений не кажется слишком значительным, можно сделать вывод о том, что между этими двумя словами существует серьезное сходство. Иначе говоря, эти два слова можно выстроить параллельно, буква за буквой:

```
o-currance
occurrence
```

Дефис (-) обозначает разрыв, который нужно добавить в первое слово для совмещения его со вторым. Более того, расположение не идеально, поскольку буква *e* стоит напротив *a*.

Нужно построить модель, в которой сходство оценивается по количеству разрывов и несовпадений, возникающих при параллельном размещении двух слов. Конечно, слова можно разместить многими разными способами, например:

```
o-curr-ance
occurre-nce
```

с тремя разрывами и без несовпадений. Что лучше: один разрыв и одно несовпадение или три разрыва без несовпадений?

Обсуждение упрощалось тем, что мы приблизительно представляли, как должно выглядеть соответствие. Если две строки не похожи на английскую слова (например, *abbbaabbbbaab* и *ababaaabbbbbab*), решить, можно их выстроить параллельно или нет, становится сложнее:

```
abbbaa--bbbbaab
ababaaabbbbba-b
```

Электронные словари и системы проверки орфографии — не самые требовательные приложения такого типа. Оказывается, определение сходства между строками является одной из основных вычислительных задач, которые приходится решать в наши дни специалистам по молекулярной биологии.

Строки встречаются в биологии абсолютно естественно: геном организма (полный набор его генетического материала) делится на гигантские линейные молекулы ДНК, называемые *хромосомами*, каждая из которых может рассматриваться как одномерное химическое устройство хранения информации. Более того, можно вполне адекватно представить себе хромосому как огромную ленту, на которой записана строка из символов алфавита {*A*, *C*, *G*, *T*}. В этой строке закодированы инструкции по построению белковых молекул; используя химический механизм

для чтения частей хромосомы, клетка может строить белки, которые, в свою очередь, управляют ее метаболизмом.

Почему сходству отводится важная роль в этой картине? В первом приближении последовательность символов в геноме организма может рассматриваться как определение свойств организма. Предположим, имеются два штамма бактерий X и Y, между которыми существует тесная эволюционная связь. Также допустим, что нам удалось определить, что некоторая подстрока в ДНК X кодирует некоторую разновидность токсина. Если удастся обнаружить очень «похожую» подстроку в ДНК Y, еще до проведения каких-либо экспериментов можно будет выдвинуть гипотезу, что эта часть ДНК Y кодирует аналогичный токсин. Подобное применение вычислений для принятия решений о биологических экспериментах стало одной из характерных особенностей современной *вычислительной биологии*.

Итак, мы приходим к вопросу, который был задан изначально применительно к вводу неправильных слов в электронный словарь: как определить концепцию сходства между двумя строками?

В начале 1970-х годов молекулярные биологи Нидлман и Вунш предложили определение сходства, которое практически в неизменном виде стало стандартным определением, используемым в наши дни. Его положение как стандарта подкреплялось простотой и интуитивной привлекательностью, а также тем фактом, что оно было независимо открыто несколькими учеными примерно в одно время. Кроме того, к определению сходства прилагался эффективный алгоритм динамического программирования для его вычисления. Таким образом, парадигма динамического программирования была независимо открыта биологами примерно через 20 лет после того, как она была впервые сформулирована математиками и специалистами по информатике.

В основу определения заложены соображения, изложенные выше, и особенно концепция «выстраивания в линию» двух строк. Допустим, имеются две строки X и Y; строка X содержит последовательность символов $x_1, x_2, ..., x_m$, а строка Y — последовательность символов $y_1, y_2, ..., y_n$. Рассмотрим множества $\{1, 2, ..., m\}$ и $\{1, 2, ..., n\}$, представляющие разные позиции в строках X и Y, и рассмотрим паросочетание этих множеств (напомним, что *паросочетанием* называется множество упорядоченных пар, обладающее тем свойством, что каждый элемент входит не более чем в одну пару). Паросочетание M этих двух множеств называется *выравниванием* при отсутствии «пересекающихся» пар: если $(i, j), (i', j') \in M$ и $i < i'$, то $j < j'$. На интуитивном уровне выравнивание предоставляет способ параллельного расположения двух строк: оно сообщает, какие пары позиций будут находиться друг напротив друга. Например,

```
stop-
-tops
```

соответствует выравниванию $\{(2, 1), (3, 2), (4, 3)\}$.

Наше определение сходства будет основано на нахождении оптимального выравнивания между X и Y по следующему критерию. Предположим, M — заданное выравнивание между X и Y.

- Во-первых, должен существовать параметр $\delta > 0$, определяющий *штраф за разрыв*. Для каждой позиции X или Y, не имеющей пары в M (то есть разрыва), вводится штраф δ.
- Во-вторых, для каждой пары букв p, q в нашем алфавите существует *штраф за несоответствие* α_{pq} за совмещение p с q. Таким образом, для всех $(i, j) \in M$ совмещение x_i с y_j приводит к выплате соответствующего штрафа за несоответствие $\alpha_{x_i y_j}$. В общем случае предполагается, что $\alpha_{pp} = 0$ для любого символа p, то есть совмещение символа со своей копией не приводит к штрафу за несоответствие — хотя это предположение не является обязательным для каких-либо дальнейших рассуждений.
- *Стоимость* выравнивания M вычисляется как сумма штрафов за разрывы и несоответствия. Требуется найти выравнивание с минимальной стоимостью.

В литературе по биологии процесс минимизации стоимости часто называется *выравниванием последовательностей*. Величины δ и $\{\alpha_{pq}\}$ представляют собой внешние параметры, которые должны передаваться программе для выравнивания последовательностей; и действительно, значительная часть работы связана с выбором значений этих параметров. При разработке алгоритма выравнивания последовательностей мы будем рассматривать их как заданные извне. Возвращаясь к первому примеру, обратите внимание на то, как эти параметры определяют, какое из выравниваний *ocurrance* и *occurrence* следует предпочесть: первый вариант строго лучше других в том и только в том случае, если $\delta + \alpha_{ae} < 3\delta$.

Разработка алгоритма

Для оценки сходства между строками X и Y имеется конкретное числовое определение: это минимальная стоимость выравнивания между X и Y. Чем ниже эта стоимость, тем более похожими объявляются строки. Теперь обратимся к задаче вычисления этой минимальной стоимости и оптимального выравнивания, которое обеспечивает эту стоимость для заданной пары строк X и Y.

Для решения этой задачи можно попытаться применить динамическое программирование, руководствуясь следующей базовой дихотомией.

- В оптимальном выравнивании M либо $(m, n) \in M$, либо $(m, n) \notin M$. (То есть два последних символа двух строк либо сопоставляются друг с другом, либо нет.)

Самого по себе этого факта недостаточно для того, чтобы предоставить решение методом динамического программирования. Однако предположим, что он дополняется следующим базовым фактом:

(6.14) Пусть M — произвольное выравнивание X и Y. Если $(m, n) \notin M$, то либо m-я позиция X, либо n-я позиция Y не имеют сочетания в M.

Доказательство. Действуя от обратного, предположим, что $(m, n) \notin M$ и существуют числа $i < m$ и $j < n$, для которых $(m, j) \in M$ и $(i, n) \in M$. Но это противоречит нашему определению выравнивания: имеем $(i, n), (m, j) \in M$ с $i < m$, но $n > i$, поэтому пары (i, n) и (m, j) пересекаются. ■

Существует эквивалентный вариант записи (6.14) с тремя альтернативными возможностями, приводящий напрямую к формулировке рекуррентного отношения.

(6.15) В оптимальном выравнивании M истинно хотя бы одно из следующих трех условий:

- (i) $(m, n) \in$ M; или
- (ii) m-я позиция X не имеет сочетания; или
- (iii) n-я позиция Y не имеет сочетания.

Теперь пусть OPT(i, j) обозначает минимальную стоимость выравнивания между $x_1x_2 \cdots x_i$ и $y_1y_2 \cdots y_j$. Если выполняется условие (i) из (6.15), мы платим $\alpha_{x_m y_n}$ и выравниваем $x_1x_2 \cdots x_{m-1}$ с $y_1y_2 \cdots y_{n-1}$ настолько хорошо, насколько это возможно; получаем OPT$(m, n) = \alpha_{x_m y_n}$ + OPT$(m-1, n-1)$. Если выполняется условие (ii), мы платим штраф за разрыв δ, потому что m-я позиция X не имеет сочетания, и выравниваем $x_1x_2 \cdots x_{m-1}$ с $y_1y_2 \cdots y_{n-1}$ настолько хорошо, насколько это возможно. В этом случае получаем OPT$(m, n) = \delta +$ OPT$(m-1, n)$. Аналогичным образом для случая (iii) получаем OPT$(m, n) = \delta +$ OPT$(m, n-1)$.

Используя аналогичные рассуждения для подзадачи нахождения выравнивания с минимальной стоимостью для $x_1x_2 \cdots x_i$ и $y_1y_2 \cdots y_j$, приходим к следующему факту:

(6.16) Стоимости минимального выравнивания удовлетворяют следующему рекуррентному отношению для $i \geq 1$ и $j \geq 1$:

$$\text{OPT}(i, j) = \min[\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1)].$$

Кроме того, (i, j) принадлежит оптимальному выравниванию M для этой подзадачи в том и только в том случае, если минимум достигается на первом из этих значений.

Мы добрались до точки, в которой алгоритм динамического программирования стал ясен: мы строим значения OPT(i, j), используя рекуррентное отношение в (6.16). Существуют только $O(mn)$ подзадач, и OPT(m, n) дает искомое значение.

Теперь определим алгоритм для вычисления значения оптимального выравнивания. Для выполнения инициализации заметим, что OPT$(i, 0) =$ OPT$(0, i) = i\delta$ для всех i, так как совместить i-буквенное слово с 0-буквенным словом возможно только с использованием i разрывов.

```
Alignment(X, Y)
   Массив A[0...m, 0...n]
   Инициализировать A[i, 0] = iδ для всех i
   Инициализировать A[0, j] = jδ для всех j
   For j = 1, ..., n
      For i = 1, ..., m
         Использовать рекуррентное отношение из (6.16) для вычисления A[i, j]
      Конец For
   Конец For
   Вернуть A[m, n]
```

Как и в случае с предыдущими алгоритмами динамического программирования, построение самого выравнивания осуществляется обратным отслеживанием по массиву A с использованием второй части факта (6.16).

Анализ алгоритма

Правильность алгоритма следует непосредственно из (6.16). Время выполнения равно $O(mn)$, так как массив A содержит только $O(mn)$ элементов и в худшем случае на каждый тратится постоянное время.

У алгоритма выравнивания последовательностей существует элегантное визуальное представление. Допустим, имеется двумерный решетчатый граф G_{XY} с размерами $m \times n$; строки помечены символами X, столбцы помечены символами Y, а ребра направлены так, как показано на рис. 6.17.

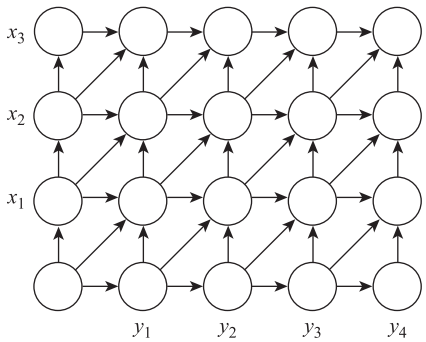


Рис. 6.17. Представление выравнивания последовательностей в форме графа

Строки нумеруются от 0 до m, а столбцы от 0 до n; узел на пересечении i-й строки и j-го столбца обозначается (i, j). Ребрам G_{XY} назначаются стоимости: стоимость каждого горизонтального и вертикального ребра равна δ, а стоимость диагонального ребра из $(i-1, j-1)$ в (i, j) равна $\alpha_{x_i y_j}$.

Теперь смысл этой схемы становится понятным: рекуррентное отношение для $\mathrm{OPT}(i, j)$ из (6.16) в точности соответствует рекуррентному отношению для пути минимальной стоимости от (0, 0) до (i, j) в G_{XY}. Следовательно, мы можем показать

(6.17) Пусть $f(i, j)$ — стоимость минимального пути из (0, 0) в (i, j) в G_{XY}. Тогда для всех i, j выполняется условие $f(i, j) = \mathrm{OPT}(i, j)$.

Доказательство. Утверждение легко доказывается индукцией по $i + j$. Если $i + j = 0$, значит, $i = j = 0$, и тогда $f(i, j) = \mathrm{OPT}(i, j) = 0$.

Теперь рассмотрим произвольные значения i и j и предположим, что утверждение истинно для всех пар (i', j') с $i' + j' < i + j$. Последнее ребро кратчайшего пути к (i, j) проходит либо из $(i-1, j-1)$, либо из $(i-1, j)$, либо из $(i, j-1)$. Следовательно, имеем

$$f(i,j) = \min[\alpha_{x_i y_j} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1)]$$
$$= \min[\alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1), \delta + \mathrm{OPT}(\text{i-1,j}), \delta + \mathrm{OPT}(i, j-1)]$$
$$= \mathrm{OPT}(i, j),$$

Переход от первой строки ко второй осуществляется по индукционной гипотезе, а переход от второй к третьей — по (6.16). ■

Таким образом, значение оптимального выравнивания равно длине кратчайшего пути в G_{XY} от (0,0) до (m, n). (Любой путь в G_{XY} из (0,0) в (m, n) будет называться *путем из угла в угол.*) Более того, диагональные ребра, используемые в кратчайшем пути, точно соответствуют парам выравнивания с минимальной стоимостью. Эти связи в задаче нахождения кратчайшего пути в графе G_{XY} не приводят к прямому улучшению времени выполнения для задачи выравнивания последовательностей; однако они способствуют интуитивному пониманию задачи, а также помогают в поиске алгоритмов для более сложных видов выравнивания последовательностей.

Например, на рис. 6.18 приведены значения кратчайшего пути из (0, 0) в каждый из узлов (i, j) для задачи выравнивания слов *mean* и *name*. В контексте нашего примера предполагается, что $\delta = 2$; стоимость сочетания гласной с другой гласной или согласной с другой согласной равна 1; стоимость сочетания гласной с согласной равна 3. Для каждой ячейки в таблице (представляющей соответствующий узел) стрелкой обозначается последний шаг кратчайшего пути, ведущего к этому узлу, — другими словами, вариант достижения минимума в (6.16). Следовательно, возвращаясь по стрелкам в обратном направлении от узла (4, 4), можно построить выравнивание методом обратного отслеживания.

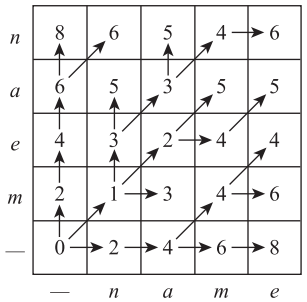


Рис. 6.18. Значения OPT для задачи выравнивания слов mean и name

6.7. Выравнивание последовательностей в линейном пространстве по принципу «разделяй и властвуй»

В предыдущем разделе был рассмотрен процесс вычисления оптимального выравнивания между двумя строками X и Y длины m и n соответственно. Построение двумерного массива $m \times n$ оптимальных решений подзадач, $OPT(\cdot, \cdot)$, оказалось эквивалентным построению графа G_{XY} с mn узлами, образующими решетку, и поиску пути наименьшей стоимости между противоположными углами. В каждом из

этих способов формулировки алгоритма динамического программирования время выполнения равно $O(mn)$, потому что значение каждой из mn ячеек массива OPT определяется за постоянное время; затраты памяти также равны $O(mn)$, потому что в них доминируют затраты на хранение массива (или графа G_{XY}).

Задача

В этом разделе мы зададимся следующим вопросом: следует ли удовольствоваться границей затрат памяти $O(mn)$? Если приложение сравнивает слова (или даже предложения) английского языка, такие затраты вполне приемлемы. Однако в биологических задачах выравнивания последовательностей часто приходится сравнивать очень длинные строки и в таких ситуациях затраты памяти $\Theta(mn)$ могут стать гораздо более серьезной проблемой, чем время $\Theta(mn)$. Например, допустим, что сравниваются две строки, каждая из которых состоит из 100 000 символов. В зависимости от процессора перспектива выполнения примерно 10 миллиардов примитивных операций вызывает меньше беспокойства, чем перспектива работы с одним 10-гигабайтным массивом.

В этом разделе будет описано очень умное усовершенствование алгоритма выравнивания последовательностей, с которым работа будет выполняться за время $O(mn)$, тогда как затраты памяти будут составлять только $O(m + n)$. Другими словами, затраты памяти сокращаются до линейных, тогда как время выполнения возрастает не более чем с постоянным множителем. Для простоты изложения шаги алгоритма будут описываться в контексте путей на графе G_{XY} — естественного эквивалента задачи выравнивания последовательностей. Таким образом, задача поиска пар в оптимальном выравнивании может быть сформулирована как задача поиска ребер в кратчайшем пути из угла в угол графа G_{XY}.

Сам алгоритм является удачным практическим применением принципа «разделяй и властвуй». В его основе лежит один важный факт: при делении задачи на несколько рекурсивных вызовов память, необходимая для вычислений, может использоваться для других вызовов. Впрочем, эта идея применяется достаточно нетривиальным образом.

Разработка алгоритма

Сначала мы покажем, что если интерес представляет только значение оптимального выравнивания, а не само выравнивание, обойтись линейными затратами памяти несложно. Дело в том, что для заполнения элемента массива A рекуррентному отношению из (6.15) необходима только информация из текущего столбца A и предыдущего столбца A. Это позволяет «свернуть» массив A в массив B с размерами $m \times 2$: при переборе алгоритмом значений j элементы вида $B[i, 0]$ содержат значение «предыдущего» столбца $A[i, j - 1]$, а элементы вида $B[i, 1]$ — значение «текущего» столбца $A[i, j]$.

```
Space-Efficient-Alignment(X, Y)
   Массив B[0...m, 0...1]
   Инициализировать B[i,0] = iδ для всех i (как в столбце 0 массива A)
   For j = 1, ..., n
      B[0,1] = jδ (соответствует элементу A[0,j])
      For i = 1, ..., m
         B[i,1] = min[ α_{x_i y_j} + B[i − 1, 0],
                       δ + B[i − 1, 1], δ + B[i, 0]]
      Конец For
      Переместить столбец 1 массива B в столбец 0,
         чтобы освободить место для следующей итерации:
            Обновить B[i,0] = B[i,1] для всех i
   Конец For
```

Как нетрудно убедиться, при завершении этого алгоритма элемент массива $B[i, 1]$ содержит значение $\mathtt{OPT}(i, n)$ для $i = 0, 1, ..., m$. Кроме того, затраты времени этого алгоритма равны $O(mn)$, а затраты памяти — $O(m)$. Но тут возникает проблема: как получить само выравнивание? Мы не располагаем достаточной информацией для выполнения такой процедуры, как `Find-Alignment`. Так как B в конце работы алгоритма содержит только последние два столбца исходного массива A, при попытке обратного отслеживания доступная информация ограничивается этими двумя столбцами. Можно попытаться обойти эту сложность «прогнозированием» выравнивания в процессе выполнения процедуры, эффективной по затратам памяти. В частности, при вычислении значений j-го столбца (теперь неявно заданного) массива A можно предположить, что при очень малом значении некоторого элемента выравнивание, проходящее через этот элемент, является перспективным кандидатом на роль оптимального. Но у этого перспективного выравнивания могут возникнуть большие проблемы в будущем, и оптимальным может оказаться другое выравнивание, которое в данный момент выглядит далеко не так перспективно.

На самом деле у этой задачи имеется решение (само выравнивание можно восстановить с затратами памяти $O(m + n)$), но оно требует совершенно нового подхода, основанного на применении принципа «разделяй и властвуй», представленного в книге ранее. Начнем с простого альтернативного пути реализации базового решения, построенного по принципам динамического программирования.

Обратная формулировка динамического программирования

Вспомните, что запись $f(i, j)$ использовалась для обозначения длины кратчайшего пути из $(0, 0)$ в (i, j) в графе G_{XY}. (Как было показано в исходном варианте алгоритма выравнивания последовательностей, $f(i, j)$ имеет то же значение, что и $\mathtt{OPT}(i, j)$). Теперь определим $g(i, j)$ как длину кратчайшего пути из (i, j) в (m, n) в G_{XY}. Функция g предоставляет столь же естественный подход к решению задачи выравнивания последовательностей методом динамического программирования,

не считая того, что на этот раз решение строится в обратном порядке: мы начинаем с $g(m, n) = 0$, а искомым ответом является $g(0, 0)$.

По строгой аналогии с (6.16) имеем следующее рекуррентное отношение для g:

(6.18) Для $i < m$ и $j < n$

$g(i,j) = \min[\alpha_{x_{i+1}y_{j+1}} + g(i+1, j+1), \delta + g(i, j+1), \delta + g(i+1, j)]$.

Это же рекуррентное отношение будет получено, если взять граф G_{XY}, «развернуть» его так, чтобы узел (m, n) находился в левом нижнем углу, и воспользоваться приведенным выше методом. По этой схеме также можно проработать полный алгоритм динамического программирования для построения значений g в обратном направлении, начиная с (m, n). Также существует версия этого обратного алгоритма динамического программирования, эффективная по затратам памяти (аналог описанного выше алгоритма), которая вычисляет значение оптимального выравнивания с затратами памяти, не превышающими $O(m + n)$. Эта обратная версия далее будет обозначаться именем `Backward-Space-Efficient-Alignment`.

Объединение прямой и обратной формулировок

Итак, у нас имеются симметричные алгоритмы для построения значений функций f и g. Идея заключается в том, чтобы использовать эти два алгоритма совместно для поиска оптимального выравнивания. Прежде всего стоит выделить два основных факта, характеризующих отношения между функциями f и g.

(6.19) Длина кратчайшего пути из угла в угол в G_{XY}, проходящего через (i, j), равна $f(i, j) + g(i, j)$.

Доказательство. Пусть ℓ_{ij} — длина кратчайшего пути из угла в угол, проходящего через (i, j). Очевидно, что любой такой путь должен перейти от (0,0) к (i, j), а затем от (i, j) к (m, n). Следовательно, его длина равна как минимум $f(i, j) + g(i, j)$, а значит, $\ell_{ij} \geq f(i, j) + g(i, j)$. С другой стороны, рассмотрим путь из угла в угол, состоящий из пути минимальной длины из $(0, 0)$ в (i, j), за которым следует путь минимальной длины из (i, j) в (m, n). Длина этого пути равна $f(i, j) + g(i, j)$, поэтому $\ell_{ij} \leq f(i, j) + g(i, j)$. Из этого следует, что $\ell_{ij} = f(i, j) + g(i, j)$. ■

(6.20) Пусть k — произвольное число из диапазона $\{0, ..., n\}$, а q — индекс, минимизирующий величину $f(q, k) + g(q, k)$. В этом случае существует путь минимальной длины из угла в угол, проходящий через узел (q, k).

Доказательство. Пусть ℓ^* — длина кратчайшего пути из угла в угол в G_{XY}. Зафиксируем значение $k \in \{0, ..., n\}$. Кратчайший путь из угла в угол должен проходить через *какой-то* из узлов k-го столбца G_{XY} — предположим, это узел (p, k), — а следовательно, из (6.19)

$$\ell^* = f(p, k) + g(p, k) \geq \min_q f(q, k) + g(q, k).$$

Рассмотрим индекс q, с которым достигается минимум в правой стороне этого выражения; имеем

$$\ell^* \geq f(q, k) + g(q, k).$$

Снова согласно (6.19) кратчайший путь из угла в угол, использующий узел (q, k), имеет длину $f(q, k) + g(q, k)$, а поскольку ℓ^* является минимальной длиной среди *всех* путей из угла в угол,

$$\ell^* \leq f(q, k) + g(q, k).$$

Следовательно, $\ell^* = f(q, k) + g(q, k)$. Таким образом, кратчайший путь из угла в угол, использующий узел (q, k), имеет длину ℓ^*, что доказывает (6.20). ■

Используя (6.20) и наши алгоритмы, эффективные по затратам памяти, для вычисления значения оптимального выравнивания, продолжим следующим образом: разделим G_{XY} по центральному столбцу и вычислим значение $f(i, n/2)$ и $g(i, n/2)$ для каждого значения i, используя два алгоритма, эффективные по затратам памяти. Затем можно определить минимальное значение $f(i, n/2)+g(i, n/2)$ и с учетом (6.20) сделать вывод о существовании кратчайшего пути из угла в угол, проходящего через узел $(i, n/2)$. В этих условиях можно провести рекурсивный поиск кратчайшего пути в части G_{XY} между (0,0) и $(i, n/2)$ и в части между $(i, n/2)$ и (m, n). Здесь принципиально то, что мы применяем эти рекурсивные вызовы последовательно и повторно используем рабочее пространство памяти между вызовами. А поскольку мы работаем только с одним рекурсивным вызовом за раз, общие затраты памяти составят $O(m + n)$. Остается ответить на ключевой вопрос: останется ли время выполнения этого алгоритма равным $O(mn)$?

При выполнении алгоритма ведется глобально доступный список P, в котором сохраняются узлы кратчайшего пути из угла в угол по мере их обнаружения. Изначально список P пуст. Он состоит из $m+n$ элементов, поскольку ни один путь из угла в угол не может содержать большего количества узлов. Используем следующие обозначения: $X[i: j]$ для $1 \leq i \leq j \leq m$ обозначает подстроку X, состоящую из $x_i x_{i+1} \ldots x_j$; аналогичным образом определяется $Y[i: j]$. Для простоты будем считать, что n является степенью 2; это предположение существенно упрощает анализ, хотя при желании его легко можно снять.

```
Divide-and-Conquer-Alignment(X, Y)
   Присвоить m количество символов в X
   Присвоить n количество символов в Y
   Если m ≤ 2 или n ≤ 2
      Вычислить оптимальное выравнивание с использованием Alignment(X, Y)
      Вызвать Space-Efficient-Alignment(X, Y[1:n/2])
      Вызвать Backward-Space-Efficient-Alignment(X, Y[n/2+ 1: n])
      Присвоить q индекс, минимизирующий f(q, n/2)+g(q, n/2)
      Добавить (q, n/2) в глобальный список P
      Divide-and-Conquer-Alignment(X[1:q], Y[1:n/2])
      Divide-and-Conquer-Alignment(X[q + 1: n], Y[n/2 + 1: n])
      Вернуть P
```

Пример первого уровня рекурсии представлен на рис. 6.19. Если минимизирующий индекс q оказывается равным 1, мы получаем две изображенные подзадачи.

Анализ алгоритма

Приведенные выше рассуждения уже доказывают, что алгоритм возвращает правильный ответ и использует память $O(m + n)$. Остается проверить только следующий факт.

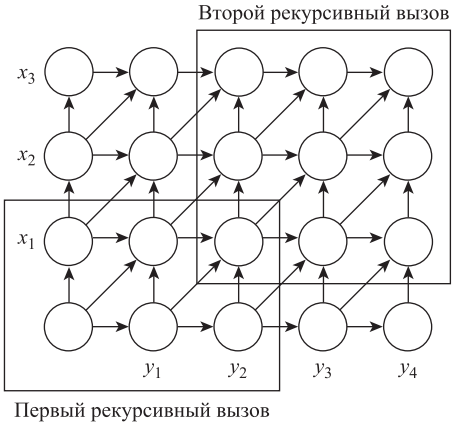


Рис. 6.19. Первый уровень рекуррентности для процедуры Divide-and-Conquer-Alignment, эффективной по затратам памяти. Две области, выделенные рамкой, обозначают входные данные для двух рекурсивных вызовов

(6.21) Время выполнения для процедуры *Divide-and-Conquer-Alignment* для строк длины m и n равно $O(mn)$.

Доказательство. Обозначим $T(m, n)$ максимальное время выполнения алгоритма для строк m и n. Алгоритм выполняет работу $O(mn)$ для построения массивов B и B'; затем он рекурсивно выполняется для строк с размерами q и $n/2$ и строк с размерами $m - q$ и $n/2$. Следовательно, для некоторой константы c и некоторого выбора индекса q:

$$T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$$

$$T(m, 2) \leq cm$$

$$T(2, n) \leq cn$$

Эта рекуррентность сложнее тех, которые встречались нам ранее в примерах «разделяй и властвуй» из главы 5. Прежде всего, время выполнения является функцией двух переменных (m и n), а не одной; кроме того, разбиение на подзадачи не обязательно является «равным», а зависит от значения q, найденного в ходе предшествующей работы, выполненной алгоритмом.

Как же подойти к разрешению этой рекуррентности? Можно попытаться угадать форму решения, рассмотрев частный случай рекуррентности, и затем воспользоваться частичной подстановкой для заполнения параметров. А именно предположим, что $m = n$, а точка разбиения q находится точно в середине.

В этом (надо признаться, весьма ограниченном) частном случае мы можем записать функцию $T(\cdot)$ для одной переменной n, присвоить $q = n/2$ (предполагается идеальное деление пополам) и прийти к условию

$$T(n) \le 2T(n/2) + cn^2.$$

Это выражение полезно, потому что оно уже было разрешено при обсуждении рекуррентности в начале главы 5, а именно: из этой рекуррентности следует $T(n) = O(n^2)$.

Итак, при $m = n$ и равном разбиении время выполнения возрастает в квадратичной зависимости от n. Вооружившись этой информацией, мы возвращаемся к общему рекуррентному отношению текущей задачи и предполагаем, что $T(m, n)$ растет со скоростью произведения m и n. А именно предположим, что $T(m, n) \le kmn$ для некоторой константы k, и посмотрим, удастся ли доказать это утверждение по индукции. Начиная с базовых случаев $m \le 2$ и $n \le 2$, мы видим, что они выполняются при $k \ge c/2$. Теперь предполагая, что $T(m', n') \le km'n'$ выполняется для пар (m', n') с меньшим произведением, имеем

$$T(m, n) \le cmn + T(q, n/2) + T(m - q, n/2)$$

$$\le cmn + kqn/2 + k(m - q)n/2$$

$$= cmn + kqn/2 + kmn/2 - kqn/2$$

$$= (c + k/2)mn$$

Следовательно, шаг индукции выполняется, если выбрать $k = 2c$, а доказательство на этом завершается. ■

6.8. Кратчайшие пути в графе

В трех последних разделах этой главы мы займемся задачей нахождения кратчайших путей в графе, а также другими вопросами, тесно связанными с этой темой.

Задача

Имеется направленный граф $G = (V, E)$, каждому ребру которого $(i, j) \in E$ присвоен вес c_{ij}. Веса могут моделировать самые разные характеристики; в нашей интерпретации вес c_{ij} представляет стоимость прямого перехода от узла i к узлу j в графе.

Ранее мы уже обсуждали алгоритм Дейкстры для нахождения кратчайших путей в графах с положительной стоимостью ребер. Здесь рассматривается более сложная задача нахождения кратчайших путей, в которых стоимости могут быть отрицательными. Решение этой задачи актуально во многих ситуациях, среди которых особенно важны две. Во-первых, отрицательные стоимости играют ключевую роль при моделировании некоторых процессов с кратчайшими путями: например,

узлы могут представлять агентов в финансовых отношениях, а c_{ij} — стоимость операции, в которой вы покупаете у агента i, а затем немедленно продаете агенту j. В этом случае путь представляет серию операций, а ребра с отрицательной стоимостью — операции, приносящие прибыль. Во-вторых, алгоритм, который мы разработаем для обработки ребер с отрицательной стоимостью, во многих важных отношениях оказывается более гибким и *децентрализованным*, чем алгоритм Дейкстры. Как следствие, он находит важное применение при проектировании алгоритмов распределенной маршрутизации, определяющих наиболее эффективный путь в сети передачи данных.

В этом и двух следующих разделах рассматриваются две взаимосвязанные задачи:

- Для заданного графа G с весами (см. выше) решить, существует ли в G *отрицательный цикл*, то есть направленный цикл C, для которого

$$\sum_{ij \in C} c_{ij} < 0.$$

- Если граф не содержит отрицательных циклов, найти путь P от начального узла s к конечному узлу t с минимальной общей стоимостью; сумма

$$\sum_{ij \in P} c_{ij}$$

должна быть минимально возможной для любого пути s–t. Обычно эта задача называется как *задачей нахождения пути с минимальной стоимостью*, так и *задачей нахождения кратчайшего пути*.

В приведенном выше финансовом примере отрицательный цикл соответствует прибыльной серии операций, которая возвращается к начальной точке: мы покупаем у i_1, продаем i_2, покупаем у i_3 и т. д., в итоге возвращаясь к i_1 с чистой прибылью. Отрицательные циклы в таких сетях могут стать поводом для обращения в арбитраж.

Задачу нахождения пути s–t с минимальной стоимостью стоит рассматривать в предположении об отсутствии отрицательных циклов. Как показано на рис. 6.20, если бы в графе существовал отрицательный цикл C, путь P_s от s к циклу и другой путь P_t от цикла к t позволил бы построить путь s–t с произвольной отрицательной стоимостью: сначала по P_s переходим к отрицательному циклу C, «крутимся» в цикле сколько угодно раз и, наконец, используем P_t для перехода от C к конечному узлу t.

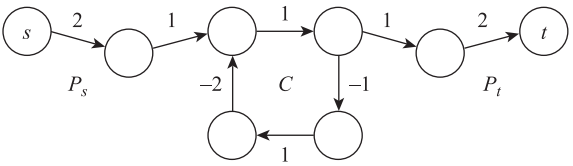


Рис. 6.20. В этом графе присутствуют пути s–t с произвольной отрицательной стоимостью (с многократным прохождением цикла C)

Разработка и анализ алгоритма

Несколько неудачных попыток

Для начала припомним алгоритм Дейкстры для нахождения кратчайшего пути при отсутствии ребер с отрицательной стоимостью. Этот метод вычисляет кратчайший путь от начальной точки s до любого другого узла v в графе, фактически с использованием жадного алгоритма. Основная идея заключается в построении такого множества S, для которого известен кратчайший путь от s к каждому узлу S. Работа начинается с $S = \{s\}$ (мы знаем, что при отсутствии отрицательных ребер кратчайший путь из s в s имеет стоимость 0), после чего начинается жадное добавление элементов в множество S. На первом жадном шаге рассматривается ребро минимальной стоимости, выходящее из узла i, то есть $min_i \in_V c_{si}$. Пусть v — узел, для которого достигается минимум. Ключевой факт, заложенный в основу алгоритма Дейкстры, заключается в том, что кратчайшим путем из s в v является путь из одного ребра $\{s, v\}$. Следовательно, узел v можно немедленно добавить в множество S. Путь $\{s, v\}$ очевидно является кратчайшим путем к v при отсутствии ребер с отрицательной стоимостью: любой другой путь из s в v должен начинаться с ребра, выходящего из s, стоимость которого по крайней мере не меньше стоимости ребра (s, v).

Если ребра могут иметь отрицательную стоимость, это рассуждение перестает быть истинным. Как подсказывает пример на рис. 6.21, *a*, путь, начинающийся с ребра с высокой стоимостью, которая компенсируется последующими ребрами с отрицательной стоимостью, может быть «дешевле» пути, начинающегося с ребра низкой стоимости. Следовательно, жадный метод в стиле алгоритма Дейкстры здесь не сработает.

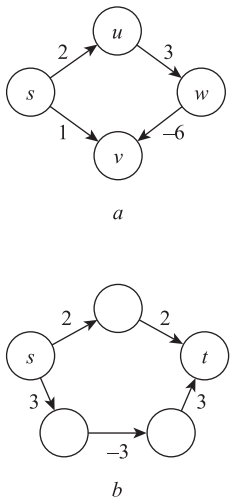


Рис. 6.21. *a* — с отрицательной стоимостью ребер алгоритм Дейкстры может дать неверный ответ на задачу кратчайшего пути; *b* — при увеличении стоимости каждого ребра на 3 стоимости всех ребер становятся неотрицательными, но при этом изменяется кратчайший путь s–t

Другая естественная идея — выполнить предварительное изменение стоимостей ребер c_{ij}, прибавив к каждой некоторую большую константу M; иначе говоря, для каждого ребра $(i, j) \in E$ выполняется $c'_{ij} = c_{ij} + M$. Если константа M достаточно велика, все измененные стоимости будут неотрицательными, и мы сможем воспользоваться алгоритмом Дейкстры для нахождения пути с минимальной стоимостью, складывающейся из c'. Тем не менее этот метод не всегда находит правильный путь с минимальной стоимостью в отношении исходных стоимостей c. Дело в том, что изменение стоимости с c на c' изменяет путь с минимальной стоимостью. Например (как показано на рис. 6.21, *b*), если стоимость пути P, состоящего из трех ребер, лишь ненамного меньше стоимости другого пути P', состоящего из двух ребер, то после изменения стоимостей P' станет «дешевле», потому что стоимость P' увеличивается на $2M$, а стоимость P — на $3M$.

Решение методом динамического программирования

Попробуем воспользоваться динамическим программированием для решения задачи поиска кратчайшего пути из s в t при наличии отрицательных стоимостей ребер, но без отрицательных циклов. Также можно опробовать идею, которая работала в прошлом: подзадача i может искать кратчайший путь с использованием только первых i узлов. Эта идея не работает немедленно, но ее можно заставить работать ценой некоторых усилий. А здесь мы рассмотрим более простое и эффективное решение — *алгоритм Беллмана–Форда*. Разработку динамического программирования как общей алгоритмической методологии часто приписывают работе Беллмана, опубликованной в 1950-х годах; а алгоритм нахождения кратчайшего пути Беллмана–Форда стал одним из ее первых практических применений.

Решение из области динамического программирования, которое мы разработаем, будет основано на следующем ключевом факте.

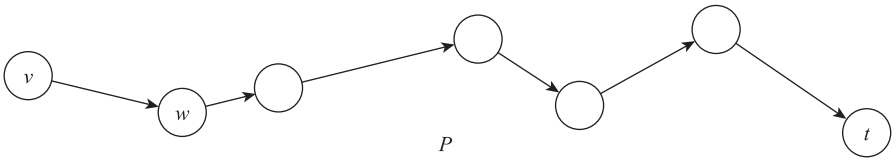


Рис. 6.22. Путь с минимальной стоимостью P из v в t, использующий не более i ребер

(6.22) Если граф G не содержит отрицательных циклов, то существует кратчайший путь из s в t, который является простым (то есть не содержит повторяющихся узлов), а следовательно, содержит не более n-1 ребер.

Доказательство. Так как каждый цикл имеет неотрицательную стоимость, в кратчайшем пути P из s в t с наименьшим числом ребер не повторяется никакая вершина v. Если бы в P повторялась вершина v, то мы могли бы удалить часть P между последовательными посещениями v, что привело бы к построению пути с не большей стоимостью и меньшим количеством ребер.

Воспользуемся `OPT`(i, v) для обозначения минимальной стоимости пути v-t с использованием не более i ребер. Согласно (6.22), исходная задача заключается

в вычислении OPT($n - 1, s$). (Также алгоритм можно спроектировать так, чтобы подзадачи соответствовали минимальной стоимости пути s–v, использующего не более i ребер. Такая форма образует более естественную параллель с алгоритмом Дейкстры, но она не столь естественна в контексте протоколов маршрутизации, о которых речь пойдет позднее.)

Теперь нужно найти простой способ выражения OPT(i, v) с использованием меньших подзадач. Как вы увидите, в наиболее естественном решении задействовано много разных параметров; это еще один пример принципа «многовариантного выбора», представленного в алгоритме сегментированной задачи наименьших квадратов.

Зафиксируем оптимальный путь P, представляющий OPT(i, v) на рис. 6.22.

- Если путь P содержит не более $i - 1$ ребер, то OPT(i, v) = OPT($I - 1, v$).
- Если путь P содержит i ребер и первым является ребро (v, w), то OPT(i, v) = $= cvw + \text{OPT}(i - 1, w)$.

Так мы приходим к следующей рекурсивной формуле.

(6.23) Если $i > 0$, то

$$\text{OPT}(i, v) = \min(\text{OPT}(i - 1, v), \min_{w \in V} (\text{OPT}(i - 1, w) + c_{vw})).$$

Используя это рекуррентное соотношение, получаем следующий алгоритм динамического программирования для вычисления значения OPT($n - 1, s$).

```
Shortest-Path(G, s, t)
  n = количество узлов в G
  Массив M[0... n − 1, V]
  Определить M[0, t] = 0 и M[0, v] = ∞ для всех остальных v ∈ V
  For i = 1, ..., n−1
     For v ∈ V в произвольном порядке
        Вычислить M[i, v] с использованием рекуррентного отношения (6.23)
     End For
  End For
  Вернуть M[n − 1,s]
```

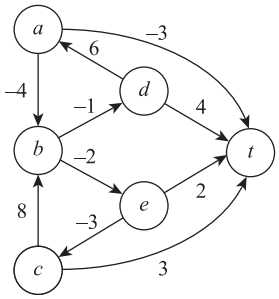
Правильность метода доказывается индукцией напрямую из (6.23). Время выполнения можно ограничить следующим образом: таблица M состоит из n^2 элементов; на вычисление каждого элемента может потребоваться время $O(n)$, так как нам придется рассмотреть максимум n узлов $w \in V$. ■

(6.24) Метод кратчайшего пути правильно вычисляет минимальную стоимость пути s–t в графе, не содержащем отрицательных циклов, и выполняется за время $O(n^3)$.

Для таблицы M, содержащей оптимальные значения подзадач, кратчайший путь, содержащий не более i ребер, может быть получен за время $O(in)$ посредством обратного отслеживания по меньшим подзадачам.

В качестве примера рассмотрим граф на рис. 6.23, *a*, в котором требуется найти кратчайший путь от каждого узла до t. В таблице на рис. 6.23, *b* представлен массив M, элементы которого соответствуют значениям $M[i, v]$ из алгоритма. Одна строка таблицы соответствует кратчайшему пути из конкретного узла в t, так как

количество ребер в пути может увеличиться. Например, кратчайший путь из узла *d* в *t* обновляется четыре раза: из *d*-*t* в *d*-*a*-*t*, *d*-*a*-*b*-*e*-*t* и, наконец, *d*-*a*-*b*-*e*-*c*-*t*.



a

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	•	–3	–3	–4	–6	–6
b	•	•	0	–2	–2	–2
c	•	3	3	3	3	3
d	•	4	3	3	2	0
e	•	2	0	0	0	0

b

Рис. 6.23. Для направленного графа в (*a*) алгоритм кратчайшего пути строит таблицу динамического программирования (*b*)

Расширения: основные усовершенствования алгоритма

Анализ улучшенного времени выполнения

Как выясняется, мы можем обеспечить более точный анализ времени выполнения для случая, если граф G содержит не слишком много ребер. Направленный граф с n узлами может иметь примерно до n^2 ребер, так как теоретически ребро может связывать каждую пару узлов, но многие графы оказываются намного более разреженными. Как мы уже неоднократно видели в примерах, при работе с графом, у которого количество ребер m значительно меньше n^2, время выполнения может быть полезно записывать формулой, содержащей как m, так и n; это позволит ускорить обработку для графов с относительно небольшим количеством ребер.

Если действовать чуть внимательнее при анализе приведенного выше метода, можно улучшить время выполнения до $O(mn)$ без значительного изменения самого алгоритма.

(6.25) Метод кратчайшего пути может быть реализован за время $O(mn)$.

Доказательство. Рассмотрим вычисление элемента массива $M[i, v]$ в соответствии с рекуррентным отношением (6.23); имеем

$$M[i, v] = \min(M[i-1, v], \min_{w \in V} (M[i-1, w] + c_{vw})).$$

Мы предположили, что вычисление этого минимума может занять время до $O(n)$, так как существуют n возможных узлов w. Но разумеется, этот минимум необходимо вычислять только по узлам w, с которым узел v связан ребром; обозначим это число n_v. В этом случае вычисление элемента массива $M[i,v]$ занимает время $O(n_v)$. Элемент должен вычисляться для каждого узла v и каждого индекса $0 \le i \le n-1$, поэтому в результате получается время выполнения

$$O\left(n \sum_{v \in V} n_v\right).$$

В главе 3 точно такой же анализ проводился для других алгоритмов, работающих с графами, а утверждение (3.9) из этой главы использовалось для ограничения выражения $\sum_{v \in V} n_v$ для ненаправленных графов. Здесь мы имеем дело с направленными графами, а n_v обозначает количество ребер, выходящих из v. В некотором смысле работать со значением $\sum_{v \in V} n_v$ для направленных графов даже проще: каждое ребро выходит ровно из одного узла V, поэтому каждое ребро учитывается в этом выражении ровно один раз. Следовательно, имеем $\sum_{v \in V} n_v = m$. Подставляя этот результат в выражение

$$O\left(n \sum_{v \in V} n_v\right)$$

для времени выполнения, получаем границу времени выполнения $O(mn)$. ■

Снижение затрат памяти

Всего одно небольшое изменение реализации позволит значительно снизить затраты памяти. Типичным недостатком многих алгоритмов динамического программирования являются высокие требования к памяти, обусловленные необходимостью хранения массива M. В алгоритме Беллмана–Форда в том виде, в каком он записан, этот массив имеет размер n^2; тем не менее сейчас мы покажем, как сократить его до $O(n)$. Вместо того чтобы сохранять $M[i, v]$ для каждого значения i, мы будем использовать и обновлять одно значение $M[v]$ для каждого узла v — длину кратчайшего пути из v в t, найденного на данный момент. Алгоритм, как и прежде, выполняется для итераций $i = 1, 2, ..., n-1$, но i теперь используется как простой счетчик; при каждой итерации и для каждого узла v выполняется обновление

$$M[v] = \min(M[v], \min_{w \in V} (c_{vw} + M[w])).$$

А теперь отметим следующий факт.

(6.26) На протяжении работы алгоритма $M[v]$ содержит длину некоторого пути из v в t и после i циклов обновлений значение $M[v]$ не превышает длину кратчайшего пути из v в t, использующего не более i ребер.

С учетом (6.26) мы теперь снова можем использовать (6.22), чтобы показать, что работа завершена после $n - 1$ итераций. Так как сохраняется только массив M, индексируемый по узлам, для его хранения достаточно памяти $O(n)$.

Поиск кратчайших путей

Одна из проблем, о которой стоит помнить, — содержит ли версия алгоритма, эффективная по затратам памяти, достаточно информации для восстановления самих кратчайших путей? В случае алгоритма выравнивания последовательностей из предыдущего раздела нам пришлось прибегнуть к хитроумному методу «разделяй и властвуй» для восстановления решения в реализации, эффективной по затратам памяти. Однако на этот раз короткие пути восстанавливаются намного проще.

Чтобы облегчить восстановление кратчайших путей, мы усовершенствуем код: каждый узел v будет сохранять первый узел (после себя) на пути к конечной точке t; обозначим этот первый узел *first*$[v]$. Значение *first*$[v]$ обновляется при каждом обновлении расстояния $M[v]$. Другими словами, когда значение $M[v]$ сбрасывается до минимума $\min_{w \in V} (c_{vw} + M[w])$, мы присваиваем *first*$[v]$ узел w, для которого этот минимум достигается.

Пусть теперь P обозначает направленный «граф указателей», узлами которого являются V, а ребрами — $\{(v, \textit{first}[v])\}$. Ключевой факт заключается в следующем:

(6.27) Если граф указателей P содержит цикл C, то этот цикл должен иметь отрицательную стоимость.

Доказательство. Если *first*$[v] = w$ в любой момент времени, то должно выполняться условие $M[v] \geq c_{vw} + M[w]$. В самом деле, левая и правая стороны равны после обновления, которое задает *first*$[v]$ равным w; а поскольку $M[w]$ может уменьшаться, это уравнение может превратиться в неравенство.

Пусть $v_1, v_2, ..., v_k$ — узлы, входящие в цикл C графа указателей; будем считать, что (v_k, v_1) — последнее добавляемое ребро. Теперь рассмотрим значения непосредственно перед последним обновлением. В этот момент $M[v_i] \geq c_{v_i v_{i+1}} + M[v_i{+}_1]$ для всех $i = 1, ..., k - 1$, а также $M[v_k] > c_{v_k v_1} + M[v_1]$, потому что мы собираемся обновить $M[v_k]$ и изменить *first*$[v_k]$ на v_1. При суммировании всех этих неравенств значения $M[v_i]$ аннулируются, и мы получаем $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$: отрицательный цикл, как и утверждалось. ■

Теперь заметим, что если G не содержит отрицательных циклов, то из (6.27) следует, что в графе указателей P никогда не будет циклов. Для узла v рассмотрим путь, получаемый переходами по ребрам P: от v к *first*$[v] = v_1$, затем *first*$[v_1] = v_2$, и т. д. Так как граф указателей не содержит циклов, а приемник t — единственный узел, не имеющий исходящего ребра, этот путь должен вести к t. Утверждается,

что при завершении алгоритма он действительно будет кратчайшим путем из v в t для графа G.

Доказательство. Рассмотрим узел v; пусть $w = first[v]$. Так как алгоритм завершился, должно выполняться условие $M[v] = c_{vw} + M[w]$. Значение $M[t] = 0$, а следовательно, длина пути, отслеживаемого по графу указателей, равна точно $M[v]$ — что, как мы знаем, является расстоянием кратчайшего пути.

Обратите внимание: в версии алгоритма Беллмана–Форда, более эффективной по затратам памяти, количество ребер в пути, длина которого равна $M[v]$ после i итераций, может существенно превышать i. Например, если граф представляет собой единственный путь из s в t и обновления выполняются в порядке, обратном порядку следования ребер в пути, окончательные значения кратчайшего пути будут получены всего за одну итерацию. Такое происходит не всегда, поэтому говорить об улучшении времени выполнения худшего случая не приходится, но было бы неплохо воспользоваться этим фактом для ускорения алгоритма в тех экземплярах, где это все же происходит. Для этого в алгоритме нужно предусмотреть «стоп-сигнал» — некий признак, который сообщит о возможности безопасного завершения перед достижением итерации $n - 1$.

Таким «стоп-сигналом» является простая последовательность следующих наблюдений: если когда-либо будет выполнена полная итерация i, в которой не изменится ни одно значение $M[v]$, то ни одно значение $M[v]$ не изменится в будущем, потому что будущие итерации будут начинаться с тем же набором элементов массива. Следовательно, выполнение алгоритма может быть безопасно прервано. Обратите внимание: если *отдельное* значение $M[v]$ остается неизменным, этого недостаточно; для безопасного завершения все эти значения должны остаться неизменными в течение одной итерации. ■

6.9. Кратчайшие пути и дистанционно-векторные протоколы

В частности, задача нахождения кратчайшего пути имеет практическое применение в сетях передачи данных для определения самого эффективного пути к конечной точке. Сеть представляется графом, узлы которого соответствуют маршрутизаторам; между v и w существует ребро, если два маршрутизатора соединены прямым каналом связи. Стоимость c_{vw} определяет задержку в канале (v, w); задача нахождения кратчайшего пути со стоимостями определяет путь с наименьшей задержкой от начального узла s к конечному узлу t. Естественно, величины задержек неотрицательные, поэтому для вычисления кратчайшего пути можно воспользоваться алгоритмом Дейкстры. Однако для вычисления кратчайшего пути по Дейкстре необходимо иметь глобальную информацию о сети: алгоритм должен хранить множество S узлов, для которых кратчайшие пути были определены, и принимать глобальное решение о том, какой узел добавить в S следующим. Хотя на маршрутизаторах может в фоновом режиме работать протокол, который собирает глобаль-

ную информацию для реализации такого алгоритма, решения, ограничивающиеся локальной информацией о соседних узлах, часто оказываются более простыми и гибкими.

Если задуматься, алгоритм Беллмана–Форда, рассмотренный в предыдущем разделе, обладает как раз таким свойством «локальности». Предположим, в каждом узле v будет храниться его значение $M[v]$; чтобы обновить это значение, v достаточно получить значение $M[w]$ от каждого соседа w и вычислить

$$\min_{w \in P} (c_{vw} + M[w])$$

на основании полученной информации.

А теперь рассмотрим усовершенствование алгоритма Беллмана–Форда, которое делает его более подходящим для применения в маршрутизаторах, а заодно и ускоряет его работу в практических условиях. Текущая реализация алгоритма Беллмана–Форда может рассматриваться как *алгоритм извлечения* (pull-based). При каждой итерации i каждый узел v должен связаться с каждым соседом w и «извлечь» из него новое значение $M[w]$. Если значение узла w не изменилось, то v не нужно получать его заново; тем не менее узел v знать об этом не может, и поэтому он все равно должен извлечь информацию.

Эта неэффективность наводит на мысль о симметричной реализации, основанной на *продвижении*, в которой значения передаются только при изменении. А именно, каждый узел w, значение расстояния которого $M[w]$ изменяется при итерации, оповещает всех своих соседей о новом значении при следующей итерации; это позволяет соседям обновить свою информацию. Если значение $M[w]$ не изменилось, то текущее значение уже известно соседям w и «продвигать» его заново не нужно. Такая схема экономит время выполнения, так как не все значения должны продвигаться при каждой итерации. Также алгоритм может быть завершен преждевременно, если во время итерации ни одно значение не изменилось. Ниже приводится полное описание реализации, основанной на продвижении информации.

```
Push-Based-Shortest-Path(G, s, t)
   n = количество узлов в G
   Массив M[V]
   Инициализировать M[t] = 0 и M[v] = ∞ для всех остальных v ∈ V
   For i = 1, ..., n − 1
      For w ∈ V в произвольном порядке
         Если значение M[w] обновлялось при предыдущей итерации
            Для всех узлов (v, w) в произвольном порядке
               M[v] = min(M[v], c_vw + M[w])
               Если при этом изменяется значение M[v], присвоить first[v] = w
            Конец цикла
      Конец For
      Если ни одно значение не изменилось в этой итерации,
         завершить алгоритм
   Конец For
   Вернуть M[s]
```

В этом алгоритме рассылка информации об изменении расстояний соседей осуществляется по раундам, а каждый узел отправляет обновление при каждой итерации, в которой он изменился. Однако если узлы соответствуют маршрутизаторам в сети, вряд ли следует ожидать, что все будет работать в идеальном порядке; некоторые маршрутизаторы могут сообщать об обновлениях намного быстрее других, а на каких-то маршрутизаторах могут возникнуть задержки. Следовательно, маршрутизаторы в конечном итоге будут выполнять асинхронную версию алгоритма: каждый раз, когда на узле w обновляется значение $M[w]$, узел «активизируется» и оповещает своих соседей о новом значении. Если понаблюдать за поведением всех связанных маршрутизаторов, это будет выглядеть примерно так:

```
Asynchronous-Shortest-Path(G, s, t)
   n = количество узлов в G
   Массив M[V]
   Инициализировать M[t] = 0 и M[v] = ∞ для всех остальных v ∈ V
   Объявить t активным узлом, а все остальные узлы - неактивными
   Пока существует активный узел
      Выбрать активный узел w
         Для всех ребер (v,w) в произвольном порядке
            M[v] = min(M[v],c_vw+M[w])
               Если при этом изменяется значение M[v], присвоить first[v] = w
               Узел v становится активным
            Конец цикла
      Конец For
      Узел w становится неактивным
   Конец Пока
```

Можно показать, что даже эта версия алгоритма, практически не координирующая порядок обновлений, придет к правильным значениям расстояний кратчайшего пути до t при условии, что при каждой активизации узел в конечном итоге сможет связаться со своими соседями.

Разработанный алгоритм использует одну конечную точку t, а все узлы $v \in V$ вычисляют кратчайший путь к t. Вероятно, более общая постановка задачи требует определения расстояний и кратчайших путей между всеми парами узлов в графе. Для получения таких расстояний фактически используются n разных вычислений, по одному для каждой конечной точки. Такой алгоритм называется *дистанционно-векторным протоколом*, потому что каждый узел хранит вектор расстояний до всех остальных узлов в сети.

Недостатки дистанционно-векторного протокола

Одна из главных проблем распределенной реализации алгоритма Беллмана–Форда для маршрутизаторов (протокол, описанный выше) заключается в том, что она строится на базе исходного алгоритма динамического программирования, предполагающего, что стоимости ребер остаются неизменными на протяжении работы алгоритма. До настоящего момента мы разрабатывали алгоритмы, подразумевая, что программа выполняется на одном компьютере (или группе компьютеров под

централизованным управлением) и обрабатывает конкретные входные данные. В таком контексте предположение о том, что входные данные не будут изменяться во время выполнения программы, выглядит вполне оправданно. Но стоит перейти к условиям сети с маршрутизаторами, как такое предположение начинает создавать проблемы. Стоимости ребер могут изменяться по самым разным причинам: отдельные каналы могут быть перегружены, передача данных по ним замедляется, а технический сбой в канале (v, w) может повысить стоимость c_{vw} до ∞.

Следующий пример показывает, какие проблемы могут возникнуть с алгоритмом кратчайшего пути в таких ситуациях. При удалении ребра (v, w) (допустим, канал связи вышел из строя) для узла v будет естественно реагировать следующим образом: узел проверяет, входит ли ребро (v, w) в кратчайший путь к некоторому узлу t, и если входит, — увеличивает расстояние с использованием других соседей. Следует учесть, что это увеличение расстояния от v может вызвать увеличения расстояний у соседей v, если те использовали путь, проходящий через v; в сети начинают распространяться каскадные изменения. Рассмотрим очень простой пример на рис. 6.24: исходный граф состоит из трех ребер (s, v), (v, s) и (v, t), каждое из которых имеет стоимость 1.

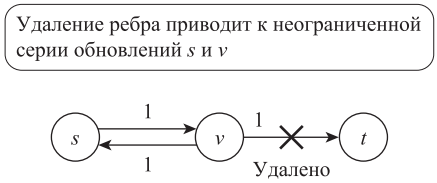


Рис. 6.24. При удалении ребра (v, t) распределенный алгоритм Беллмана–Форда начинает «отсчет до бесконечности»

Теперь предположим, что ребро (v, t) на рис. 6.24 удаляется. Как отреагирует узел v? К сожалению, у узла нет глобальной карты сети; он знает только расстояния кратчайших путей каждого из его соседей к t. Следовательно, он не знает, что с удалением (v, t) были потеряны все пути от s к t. Он видит, что $M[s] = 2$, и обновляет $M[v] = c_{vs} + M[s] = 3$, считая, что он будет использовать свое ребро стоимостью 1 к s, с последующим предполагаемым путем стоимостью 2 из s в t. Узнав об этом изменении, узел s обновляет $M[s] = c_{sv} + M[v] = 4$; при этом он считает, что будет использовать свое ребро стоимостью 1 к v, с последующим предполагаемым путем стоимостью 3 из v к t. Узлы s и v будут попеременно обновлять свое расстояние до t, пока один из них не найдет альтернативный маршрут; если сеть действительно теряет связность, как в нашем случае, эти обновления будут продолжаться бесконечно долго (так называемая проблема отсчета до бесконечности).

Чтобы избежать возникновения этой проблемы и других сложностей, которые возникают из-за ограниченного объема информации, доступной узлам в алгоритме Беллмана–Форда, проектировщики схем сетевой маршрутизации предпочитают вместо дистанционно-векторных протоколов применять более выразительные *протоколы векторов путей*, в которых каждый узел хранит не только расстояние и первый переход на пути к цели, но и некоторое представление всего пути.

При наличии информации о путях узлам не нужно обновлять свои пути для использования заведомо удаленных ребер; в то же время для хранения полных путей требуется существенно больше памяти. В истории Интернета некогда произошел переход с дистанционно-векторных протоколов на протоколы векторов путей; в настоящее время метод векторов путей задействован в протоколе *BGP* (Border Gateway Protocol), лежащем в основе маршрутизации в Интернете.

6.10. Отрицательные циклы в графе

До настоящего времени при рассмотрении алгоритма Беллмана–Форда мы предполагали, что граф содержит ребра с отрицательной стоимостью, но не имеет отрицательных циклов. А теперь рассмотрим более общий случай графа, который может содержать отрицательные циклы.

Задача

Для начала зададим себе два естественных вопроса:

- Как определить, содержит ли граф отрицательный цикл?
- Как найти отрицательный цикл в графе, в котором он присутствует?

Алгоритм, разработанный для поиска отрицательных циклов, также приводит к улучшению практической реализации алгоритма Беллмана–Форда из предыдущих разделов.

Оказывается, идеи, описанные ранее, позволяют найти отрицательные циклы, которые имеют путь, достигающий конечной точки t. Но прежде чем углубляться в подробности, сравним задачу нахождения отрицательного цикла, способного достичь заданного t, с более естественной (казалось бы) задачей нахождения отрицательного цикла в *любом* месте графа независимо от его расположения относительно конечной точки. Как выясняется, разработка решения первой задачи позволяет также получить решение второй задачи следующим образом. Предположим, мы начинаем с графа G, добавляем в него новый узел t и соединяем каждый из остальных узлов v в графе с узлом t через ребро стоимости 0, как показано на рис. 6.25. Назовем новый «дополненный граф» G'.

(6.29) Дополненный граф G' содержит такой отрицательный цикл C, что существует путь из C в конечную точку t в том и только в том случае, если исходный граф содержит отрицательный цикл.

Доказательство. Предположим, G содержит отрицательный цикл. Тогда цикл C очевидно содержит ребро к t в G', поскольку все узлы содержат ребро к t.

Теперь предположим, что G' содержит отрицательный цикл с путем к t. Так как никакое ребро не выходит из t в G', этот цикл не может содержать t. А поскольку G' совпадает с G во всем, кроме узла t, этот цикл также является отрицательным циклом в G. ■

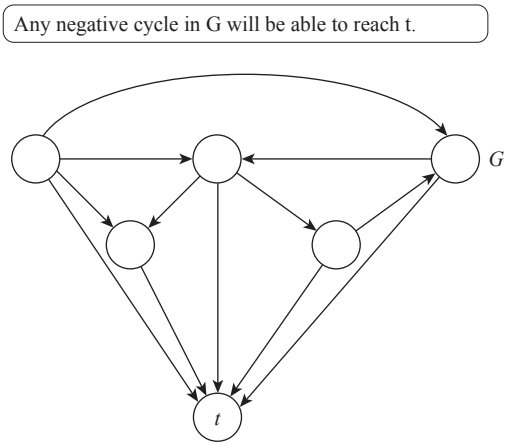


Рис. 6.25. Дополненный граф

Итак, для решения задачи достаточно определить, содержит ли G отрицательный цикл с путем к заданному конечному узлу t; этим мы сейчас и займемся.

Разработка и анализ алгоритма

Наша работа над алгоритмом начнется с адаптации исходной версии алгоритма Беллмана–Форда, которая была менее эффективной в использовании памяти. Начнем с расширения определений $\text{OPT}(i, v)$ из алгоритма Беллмана–Форда, определив их для значений $i \geq n$. При наличии в графе отрицательного цикла (6.22) уже не действует, и кратчайший путь может становиться все короче и короче при многократном проходе отрицательного цикла. Для любого узла v в отрицательном цикле, у которого имеется путь к t, действует следующее утверждение:

(6.30) Если узел v может достигнуть узла t и входит в отрицательный цикл, то

$$\lim_{i \to \infty} \text{OPT}(i, v) = -\infty.$$

Если граф не содержит отрицательных циклов, то из (6.22) вытекает следующее утверждение:

(6.31) Если G не содержит отрицательных циклов, то $\text{OPT}(i, v) = \text{OPT}(n - 1, v)$ для всех узлов v и всех $i \geq n$.

Но для насколько больших значений i нужно вычислить значения $OPT(i, v)$, прежде чем сделать вывод об отсутствии в графе отрицательных циклов? Например, узел v может удовлетворять уравнению $\text{OPT}(n, v) = OPT(n - 1, v)$ и при этом все равно лежать на отрицательном цикле. (А вы видите почему?) Как выясняется, гарантии дает только выполнение условия для всех узлов.

(6.32) В графе не существует отрицательного цикла с путем к узлу t в том и только в том случае, если $\text{OPT}(n, v) = \text{OPT}(n - 1, v)$ для всех узлов v.

Доказательство. Утверждение (6.31) уже доказано в прямом направлении. Что касается обратного направления, мы воспользуемся аргументом, примененным ранее для обоснования безопасности преждевременной остановки алгоритма Беллмана–Форда. А именно, предположим, что $\texttt{OPT}(n, v) = \texttt{OPT}(n-1, v)$ для всех узлов v. Значения $\texttt{OPT}(n+1, v)$ могут быть вычислены по $OPT(n, v)$; но все эти значения совпадают с соответствующими $\texttt{OPT}(n-1, v)$. Следовательно, выполняется равенство $\texttt{OPT}(n+1, v) = \texttt{OPT}(n-1, v)$. Распространяя эти рассуждения на будущие итерации, мы видим, что ни одно из значений никогда не изменится снова, то есть $\texttt{OPT}(i, v) = \texttt{OPT}(n-1, v)$ для всех узлов v и всех $i \geq n$. Следовательно, не может быть отрицательного цикла C, имеющего путь к t; для любого узла w в этом цикле C из (6.30) следует, что значения $\texttt{OPT}(i, w)$ становятся сколь угодно отрицательными с увеличением i. ■

Утверждение (6.32) предоставляет механизм $O(mn)$ для принятия решения о том, существует ли в G отрицательный цикл, способный достичь t. Мы вычисляем значения $\texttt{OPT}(i, v)$ для узлов G и для значений i вплоть до n. Согласно (6.32), отрицательного цикла не существует в том, и только в том случае, если существует некоторое значение $i \leq n$, для которого $\texttt{OPT}(i, v) = \texttt{OPT}(i-1, v)$ для всех узлов v.

Итак, мы определили, имеется ли в графе отрицательный цикл с путем из цикла к t, но сам цикл еще не найден. Чтобы найти отрицательный цикл, рассмотрим такой узел v, что $\texttt{OPT}(n, v) = \texttt{OPT}(n-1, v)$: для этого узла путь P из v в t со стоимостью $\texttt{OPT}(n, v)$ должен использовать *ровно n* ребер. Этот путь минимальной стоимости P из v в t находится обратным отслеживанием по подзадачам. Как и в доказательстве (6.22), простой путь может содержать только n-1 ребер, поэтому путь P должен содержать цикл C. Утверждается, что цикл C имеет отрицательную стоимость.

(6.33) Если граф G состоит из n узлов и $\texttt{OPT}(n, v) = OPT(n-1, v)$, то путь P из v в t стоимостью $\texttt{OPT}(n, v)$ содержит цикл C и C имеет отрицательную стоимость.

Доказательство. Сначала заметим, что путь P должен содержать n ребер, так как $\texttt{OPT}(n, v) = \texttt{OPT}(n-1, v)$, поэтому каждый путь, содержащий $n-1$ ребер, имеет стоимость большую, чем у пути P. В графе с n узлами путь, состоящий из n ребер, должен содержать (хотя бы) повторяющийся узел; пусть w — узел, встречающий в P более одного раза. Пусть C — цикл в P между двумя последовательными вхождениям узла w. Если бы цикл C не был отрицательным, то удаление C из P дало бы путь v-t, содержащий менее n ребер и имеющий не большую стоимость. Это противоречит нашему предположению о том, что $\texttt{OPT}(n, v) = \texttt{OPT}(n-1, v)$, а следовательно, цикл C должен быть отрицательным. ■

(6.34) Описанный выше алгоритм находит отрицательный цикл в G, если такой цикл существует, и выполняется за время $O(mn)$.

Расширения: улучшенные алгоритмы нахождения кратчайшего пути и отрицательного цикла

В конце раздела 6.8 рассматривалась реализация алгоритма Беллмана–Форда, эффективная по затратам памяти, для графов, не содержащих отрицательных циклов.

В этом разделе мы реализуем обнаружение отрицательных циклов способом, не менее эффективным в отношении затрат памяти. Помимо экономии памяти, он также значительно ускоряет работу алгоритма даже для графов без отрицательных циклов. Реализация будет базироваться на том же графе указателей P, построенном из «первых ребер» (v, *first*[v]), который использовался в реализации из раздела 6.8. Согласно (6.27) мы знаем, что если граф указателей содержит цикл, то этот цикл имеет отрицательную стоимость, и работа на этом завершается. Но если G содержит отрицательный цикл, гарантирует ли это, что граф указателей будет содержать цикл? Кроме того, сколько лишнего вычислительного времени уйдет на периодические проверки наличия цикла в P?

В идеале нам хотелось бы определять, появляется ли цикл в графе указателей при каждом добавлении нового ребра (v, w) с *first*[v] = w. Дополнительное преимущество такого «моментального» обнаружения циклов заключается в том, что нам не придется ожидать n итераций, чтобы узнать, содержит ли граф отрицательный цикл: алгоритм может завершиться сразу же после обнаружения отрицательного цикла. Ранее было показано, что если граф G не содержит отрицательных циклов, алгоритм может быть остановлен на ранней стадии, если при какой-то итерации значения кратчайшего пути $M[v]$ остаются неизменными для всех узлов v. Моментальное обнаружение отрицательного цикла станет аналогичным правилом раннего завершения для графов, содержащих отрицательные циклы.

Рассмотрим новое ребро (v, w) с *first*[v] = w, добавленное в граф указателей P. Перед добавлением (v, w) граф указателей не содержит циклов, поэтому он состоит из путей от каждого узла v к конечной точке t. Наиболее естественный способ проверки того, приведет ли добавление ребра (v, w) к созданию цикла в P, заключается в отслеживании текущего пути от w к t за время, пропорциональное длине этого пути. Если на этом пути будет обнаружен узел v, значит, образовался цикл и, согласно (6.27), граф содержит отрицательный цикл. Пример представлен на рис. 6.26, где указатель *first*[v] обновляется с u на w; в части a это не приводит к появлению (отрицательного) цикла, а в части b приводит. Однако при таком отслеживании серии указателей от v можно потратить время до $O(n)$, если дойти по пути до t, так и не обнаружив цикла. Сейчас мы рассмотрим метод, не требующий увеличения время выполнения на $O(n)$.

Мы знаем, что перед добавлением нового ребра (v, w) граф указателей был направленным деревом. Другой способ проверки того, приводит ли добавление (v, w) к созданию цикла, основан на рассмотрении всех узлов поддерева, направленного к v. Если w входит в это поддерево, то (v, w) образует цикл; в противном случае этого не происходит. (И снова рассмотрите два примера на рис. 6.26.) Чтобы найти все узлы в поддереве, направленном к v, в каждом узле v необходимо хранить список всех остальных узлов, ребра которых указывают на v. С такими указателями поддерево можно найти за время, пропорциональное размеру поддерева, указывающего на v, — снова максимум $O(n)$. Однако на этот раз нам удастся извлечь дополнительную пользу из уже проделанной работы. Обратите внимание на то, что текущее значение расстояния $M[x]$ для всех узлов x в поддереве было вычислено на основании старого значения узла v. Мы только что обновили расстояние узла v, а следовательно, знаем, что расстояния всех этих узлов будут обновлены снова. По-

метим каждый из этих узлов x как «пассивный», удалим ребро $(x, first[x])$ из графа указателей и не будем использовать x для будущих обновлений, пока не изменится его значение расстояния.

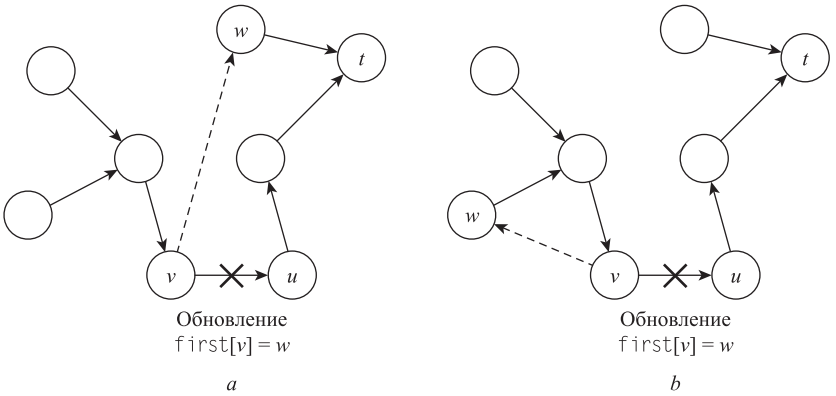


Рис. 6.26. Изменение графа указателей P при обновлении first[v] с u на w. На рис. b это приводит к появлению (отрицательного) цикла, а на рис. a — нет

Такие изменения могут сэкономить немало работы при обновлениях, но как это отразится на времени выполнения в худшем случае? Пометка пассивных узлов после каждого обновления расстояний может потребовать дополнительного времени до $O(n)$. Однако узел может быть помечен как пассивный только в том случае, если в прошлом для него был определен указатель, поэтому затраты времени на пометку пассивных узлов не превышают время, потраченное алгоритмом на обновление расстояний.

А теперь рассмотрим время, проводимое алгоритмом за другими операциями, кроме пометки пассивных узлов. Вспомните, что алгоритм разбит на итерации, а итерация $i + 1$ обрабатывает узлы, расстояние которых было обновлено на итерации i. Для исходной версии алгоритма в (6.26) мы показали, что после i итераций значение $M[v]$ не превышает значение кратчайшего пути из v в t, использующего не более i ребер. Однако при большом количестве пассивных узлов на каждой итерации это может быть не так. Например, если кратчайший путь из v в t, использующий максимум i ребер, начинается с ребра $e = (v, w)$ и узел w пассивен в этой итерации, обновить значение расстояния $M[v]$ не удастся, а значит, останется значение, превышающее длину пути через ребро (v, w). Казалось бы, возникает проблема, однако в данном случае путь через ребро (v, w) не является кратчайшим, поэтому у $M[v]$ позднее будет возможность обновиться еще меньшим значением.

Итак, вместо простого свойства, действовавшего для $M[v]$ в исходных версиях алгоритма, мы имеем следующее утверждение:

(6.35) Во время работы алгоритма $M[v]$ — длина некоторого простого пути из v в t; путь содержит не менее i ребер, если значение расстояния $M[v]$ обновляется на итерации i; после i итераций значение $M[v]$ представляет длину кратчайшего

пути для всех узлов *v*, для которых существует кратчайший путь *v*-*t*, содержащий не более *i* ребер.

Доказательство. Указатели *first* образуют дерево путей к *t*, из чего следует, что все пути, используемые для обновления расстояний, являются простыми. Тот факт, что обновления на итерации *i* порождаются путями, содержащими не менее *i* ребер, легко доказывается индукцией по *i*. Аналогичным образом индукция используется для доказательства того, что после итерации *i* значение $M[v]$ представляет расстояние для всех узлов *v*, для которых кратчайший путь из *v* в *t* содержит не более *i* ребер. Обратите внимание: узлы *v*, для которых $M[v]$ содержит фактическое расстояние кратчайшего пути, не могут быть пассивными, так как значение $M[v]$ будет обновлено при следующей итерации для всех пассивных узлов. ■

Используя это утверждение, мы видим, что время выполнения алгоритма в худшем случае по-прежнему ограничивается $O(mn)$: время, потраченное на пометку пассивных узлов, игнорируем; каждая итерация реализуется за время $O(m)$, и может быть не более $n - 1$ итераций, обновляющих значения в массиве M без нахождения отрицательного цикла, так как простые пути могут содержать не более $n - 1$ ребер. Наконец, время, потраченное на пометку пассивных узлов, ограничивается временем, потраченным на обновления. Обсуждение завершается следующим утверждением относительно быстродействия алгоритма в худшем случае: как упоминалось ранее, эта новая версия на практике оказывается самой быстрой реализацией алгоритма для графов, не содержащих отрицательных циклов, и даже ребер с отрицательной стоимостью.

(6.36) Усовершенствованная версия алгоритма, описанная выше, находит отрицательный цикл в *G*, если такой цикл существует. Она завершается немедленно, если граф указателей *P* из указателей $first[v]$ содержит цикл *C* или существует итерация, при которой никакие значения расстояний $M[v]$ не изменяются. Алгоритм использует память $O(n)$, выполняется за максимум *n* итераций и за время $O(mn)$ в худшем случае.

Упражнения с решениями

Упражнение с решением 1

Вы управляете строительством рекламных щитов на дороге с интенсивным движением, идущей с запада на восток на протяжении *M* миль. Возможные точки для установки рекламных щитов определяются числами $x_1, x_2, ..., x_n$, каждое из которых принадлежит интервалу $[0, M]$ (эти числа определяют позицию в милях от западного конца дороги). Разместив рекламный щит в точке x_i, вы получите прибыль $r_i > 0$.

По правилам дорожной службы округа два рекламных щита не могут находиться на расстоянии 5 миль и менее. Требуется разместить рекламные щиты на подмножестве точек так, чтобы обеспечить максимальную прибыль с учетом ограничения.

Пример. Предположим, $M = 20$, $n = 4$, $\{x_1, x_2, x_3, x_4\} = \{6, 7, 12, 14\}$, и $\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$. В оптимальном решении рекламные щиты размещаются в точках x_1 и x_3, с суммарной прибылью 10.

Предложите алгоритм, который получает экземпляр этой задачи и возвращает максимальную суммарную прибыль, которая может быть получена для любого действительного подмножества точек. Время выполнения алгоритма должно быть полиномиальным по n.

Решение

Следующие рассуждения позволяют естественным образом применить к этой задаче принцип динамического программирования. Рассмотрим оптимальное решение для заданного входного экземпляра задачи; в этом решении рекламный щит либо размещается в точке x_n, либо не размещается. Если щит не размещается, то оптимальное решение для точек $x_1, ..., x_n$ совпадает с оптимальным решением для точек $x_1, ..., x_{n-1}$; если же щит размещается, следует исключить x_n и все остальные точки на расстоянии до 5 миль включительно и найти оптимальное решение для полученного множества. Аналогичные рассуждения применимы при рассмотрении задачи, определяемой только первыми j сайтами $x_1, ..., x_j$: точка x_j либо включается в оптимальное решение, либо не включается с теми же последствиями.

Определим вспомогательные обозначения, которые помогут выразить эту идею. Для точки x_j величина $e(j)$ обозначает крайнюю точку на восточном направлении, которая удалена от x_j не более чем на 5 миль. Так как точки нумеруются с запада на восток, это означает, что допустимыми вариантами после решения о размещении рекламного щита в точке x_j остаются точки $x_1, x_2, ..., x_{e(j)}$, но не точки $x_{e(j)+1}, ..., x_{j-1}$.

Теперь из приведенных выше рассуждений вытекает следующее рекуррентное отношение. Если обозначить $\mathtt{OPT}(j)$ доход от оптимального подмножества точек из $x_1, ..., x_j$, имеем

$$\mathtt{OPT}(j) = \max(r_j + \mathtt{OPT}(e(j)), \mathtt{OPT}(j-1)).$$

Итак, мы собрали основные ингредиенты, необходимых для работы алгоритма динамического программирования. Во-первых, это множество из n подзадач, состоящее из первых j точек для $j = 0, 1, 2, ..., n$. Во-вторых, имеется рекуррентное отношение $\mathtt{OPT}(j) = \max(r_j + \mathtt{OPT}(e(j)), \mathtt{OPT}(j-1))$, которое позволяет строить решения подзадач.

Чтобы преобразовать все это в алгоритм, достаточно определить массив M, в котором будут храниться значения OPT, и сформировать на основе рекуррентного отношения цикл, который строит значения $M[j]$ в порядке возрастания j.

```
Инициализировать M[0] = 0 и M[1] = r1
For j = 2,3, ..., n:
   Вычислить M[j] с использованием рекуррентного отношения
End For
Вернуть M[n]
```

Как и во всех алгоритмах динамического программирования, представленных в этой главе, оптимальное множество рекламных щитов строится обратным отслеживанием по значениям массива M.

При заданных значениях $e(j)$ для всех j время выполнения алгоритма составляет $O(n)$, потому что каждая итерация цикла выполняется за постоянное время. Мы также можем вычислить все значения $e(j)$ за время $O(n)$: для каждой точки x_i определяется $x'_i = x_i - 5$. Затем отсортированный список $x_1, ..., x_n$ объединяется с отсортированным списком $x'_1, ..., x'_n$ за линейное время, как было показано в главе 2. Теперь мы перебираем элементы объединенного списка; добравшись до элемента x'_j, мы знаем, что все последующие элементы до x_j не могут быть выбраны вместе с x_j (потому что находятся на расстоянии менее 5 миль), поэтому мы просто определяем $e(j)$ как наибольшее значение i, для которого мы обнаружили x_i при переборе.

И последнее замечание по поводу этой задачи. Очевидно, решение очень похоже на решение задачи взвешенного интервального планирования, и для этого существует веская причина. Дело в том, что задача размещения рекламных щитов может быть напрямую сформулирована как экземпляр задачи взвешенного интервального планирования. Предположим, для каждой точки x_i определяется интервал с конечными точками $[x_i - 5, x_i]$ и вес r_i. Затем для любого неперекрывающегося множества интервалов соответствующее множество точек обладает тем свойством, что никакие две точки не могут находиться на расстоянии менее 5 миль. Соответственно, для любого такого множества точек (никакие две на расстоянии менее 5 миль) интервалы, связанные с ними, не будут перекрываться. Следовательно, группы неперекрывающихся интервалов точно соответствуют множеству действительных размещений рекламных щитов, а подстановка только что определенного нами множества интервалов (с их весами) в алгоритм взвешенного интервального планирования даст желаемое решение.

Упражнение с решением 2

Вас приглашают для проведения консультаций в компанию Clones 'R' Us (CRU), специализирующуюся на передовых исследованиях в области биотехнологий. Поначалу вы не уверены в том, что ваш опыт решения алгоритмических задач пригодится в такой области, но вскоре вы общаетесь с двумя подозрительно похожими инженерами, ломающими голову над нетривиальной задачей.

Эта задача основана на конкатенации последовательностей генетических материалов. Если X и Y — строки с фиксированным алфавитом $\mathcal{S}$, то запись XY обозначает строку, полученную в результате их конкатенации: содержимое X, за которым следует содержимое Y. Ученые выявили целевую последовательность A генетического материала, состоящую из m символов, и они хотят построить последовательность, как можно более близкую к A. Для этого была сформирована библиотека $\mathcal{L}$, состоящая из k (более коротких) последовательностей, длина каждой из которых не превышает n. Ученые могут легко воспроизвести любую последовательность, состоящую из сцепленных копий строк из $\mathcal{L}$ (с возможными повторениями).

Конкатенацией по $\mathcal{L}$ называется любая последовательность в форме $B_1B_2...\ B_\ell$, в которой каждый элемент B_i принадлежит множеству $\mathcal{L}$. (Еще раз: повторения разрешены, так что B_i и B_j могут соответствовать одной строке в $\mathcal{L}$ для разных значений i и j). Требуется найти конкатенацию по $\{B_i\}$, для которой стоимость выравнивания последовательностей будет минимальной. (Для вычисления стоимости выравнивания последовательностей можно считать, что вам даны стоимости разрыва δ и несоответствия α_{pq} для каждой пары $p, q \in S$.)

Предложите алгоритм с полиномиальным временем для этой задачи.

Решение

Эта задача отдаленно напоминает сегментированную задачу наименьших квадратов: имеется длинная последовательность «данных» (строка A), которую требуется «аппроксимировать» более короткими сегментами (строки из $\mathcal{L}$).

Если бы мы хотели развить эту аналогию, решение можно было бы искать следующим образом. Пусть $B = B_1B_2...\ B_\ell$ — конкатенация по $\mathcal{L}$, которая как можно лучше совмещается с заданной строкой A. (Иначе говоря, B является оптимальным решением для входного экземпляра.) Рассмотрим оптимальное выравнивание M строки A с B; t — первая позиция в A, сопоставленная с некоторым символом в B_ℓ, а A_ℓ — подстрока A от позиции t до конца. (Иллюстрация для $\ell = 3$ приведена на рис. 6.27.) В этом оптимальном выравнивании M подстрока A_ℓ оптимально выравнивается с B_ℓ; действительно, если бы существовал лучший способ выравнивания A_ℓ с B_ℓ, мы могли бы подставить его в ту часть M, в которой A_ℓ выравнивается с B_ℓ, и улучшить общее выравнивание A с B.

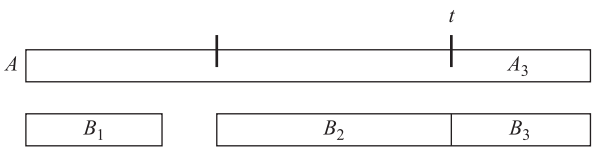


Рис. 6.27. В оптимальной конкатенации строк для выравнивания с A существует завершающая строка (B_3 на схеме), выравниваемая с подстрокой A (A_3 на схеме), которая проходит от позиции t до конца

Отсюда можно сделать вывод, что поиск оптимального решения может вестись следующим образом. Существует некий завершающий фрагмент A_ℓ, выровненный с одной из строк в $\mathcal{L}$, и для этого фрагмента мы просто находим строку в $\mathcal{L}$, которая выравнивается с ним настолько хорошо, насколько это возможно. После обнаружения оптимального выравнивания для A_ℓ этот фрагмент можно отделить и продолжить поиск оптимального решения для остатка A.

Такой подход к решению задачи ничего не говорит о том, как именно следует продолжать: мы не знаем, какую длину должен иметь фрагмент A_ℓ или с какой строкой из L он должен совмещаться. Тем не менее такие характеристики определяются алгоритмами динамического программирования. Фактически ситуация выглядит так же, как в сегментированной задаче наименьших квадратов: тогда мы знали, что должны прерваться где-то в последовательности входных точек, как можно лучше

аппроксимировать их одной линией, а затем продолжить с остальными входными точками.

Итак, подготовим все необходимое для того, чтобы сделать возможным поиск $A_{\mathcal{L}}$. Пусть $A[x:y]$ — подстрока A, состоящая из символов от позиции x до позиции y включительно, а $c(x, y)$ — стоимость оптимального выравнивания $A[x:y]$ с любой строкой из $\mathcal{L}$ (иначе говоря, мы перебираем все строки из L и находим ту, которая лучше всего выравнивается с $A[x:y]$). Пусть $\texttt{OPT}(j)$ обозначает стоимость выравнивания оптимального решения для строки $A[1:j]$.

Из приведенных выше рассуждений следует, что оптимальное решение для $A[1:j]$ складывается из выявления итоговой «границы сегмента» $t < j$, нахождения оптимального выравнивания $A[t:j]$ с одной строкой в L и итеративного выполнения для $A[1:t-1]$. Стоимость этого выравнивания для $A[t:j]$ равна $c(t,j)$, а стоимость выравнивания с остатком — $\texttt{OPT}(t-1)$. Это наводит на мысль о том, что подзадачи хорошо сочетаются друг с другом, и обосновывает следующее рекуррентное отношение.

(6.37) $\texttt{OPT}(j) = \min_{t<j} c(t,j) + \texttt{OPT}(t-1)$ для $j \geq 1$, и $\texttt{OPT}(0) = 0$.

Полный алгоритм состоит из вычисления величин $c(t, j)$ для $t < j$ и последующего построения значений $\texttt{OPT}(j)$ в порядке возрастания j. Эти значения хранятся в массиве M.

```
Присвоить M[0] = 0
Для всех пар 1 ≤ t ≤ j ≤ m
   Вычислить стоимость c(t,j) следующим образом:
   Для каждой строки B ∈ L
      Вычислить оптимальное выравнивание B с A[t : j]
   Конец цикла
   Выбрать B с лучшим выравниванием и использовать стоимость
       этого выравнивания как c(t, j)
Конец цикла
For j = 1, 2, ..., n
   Использовать рекуррентное отношение (6.37) для вычисления M[j]
Конец For
Вернуть M[n]
```

Как обычно, мы можем получить конкатенацию, достигающую этой цели, обратным отслеживанием по массиву значений OPT.

Рассмотрим время выполнения алгоритма. Во-первых, всего требуется вычислить $O(m^2)$ значений $c(t, j)$. Для каждого из них нужно проверить каждую из k строк $B \in \mathcal{L}$ и вычислить оптимальное выравнивание B с $A[t:j]$ за время $O(n(j-t)) = O(mn)$. Следовательно, общее время вычисления всех значений $c(t,j)$ составляет $O(km^3n)$.

Этот фактор доминирует над временем вычисления всех значений OPT: вычисление $\texttt{OPT}(j)$ использует рекуррентное отношение в (6.37), а это требует времени $O(m)$ для вычисления минимума. Суммируя по всем вариантам $j = 1, 2, ..., m$, мы получаем время $O(m^2)$ для этой части алгоритма.

Упражнения

1. Имеется ненаправленный граф с n узлами $G = (V, E)$. Напомним, что подмножество узлов называется *независимым множеством*, если никакие два узла не соединяются ребром. В общем виде задача нахождения большого независимого множества сложна, но при достаточно «простом» графе она имеет эффективное решение.

   Назовем граф $G = (V, E)$ *путевым графом*, если его узлы могут быть записаны в виде $v_1, v_2, ..., v_n$, причем v_i и v_j соединяются ребром в том и только в том случае, если i и j различаются ровно на 1. С каждым узлом v_i связывается положительный целый *вес* w_i.

   Для примера рассмотрим путевой граф из пяти узлов на рис. 6.28. *Веса* обозначены числами внутри узлов.

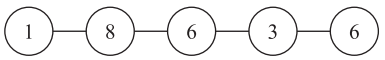


Рис. 6.28. Путевой граф с весами, обозначенными на узлах. Максимальный вес независимого множества равен 14

   Целью этого упражнения является решение следующей задачи:

   Найдите в путевом графе G независимое множество с максимальным общим весом.

   (a) Приведите пример, показывающий, что следующий алгоритм *не всегда* находит независимое множество с максимальным общим весом.

```
Жадный алгоритм «приоритетного выбора максимального веса»
   Начать с S, равного пустому множеству
   Пока в G остаются узлы
      Выбрать узел v_i с максимальным весом
      Добавить v_i в S
      Удалить v_i и его соседей из G
   Конец Пока
Вернуть S
```

   (b) Приведите пример, показывающий что следующий алгоритм тоже *не всегда* находит независимое множество с максимальным общим весом.

```
Присвоить S_1 множество всех v_i, для которых i нечетно
Присвоить S_2 множество всех v_i, для которых i четно
(обратите внимание: S_1 и S_2 являются независимыми множествами)
Определить, какое из множеств S_1 или S_2 имеет больший общий вес,
   и вернуть его.
```

   (c) Предложите алгоритм, который получает путевой граф G из n узлов с весами и возвращает независимое множество с максимальным общим весом. Время выполнения должно быть полиномиальным по n независимо от значений весов.

2. Предположим, вы руководите рабочей группой, состоящей из опытных хакеров, и каждую неделю выбираете для них задание. Задания делятся на *некритичные* (например, создание сайта для класса в местной начальной школе) и *критичные* (защита важнейших национальных секретов или помощь группе студентов Корнелльского университета в разработке компилятора). Каждую неделю приходится принимать решение: какое задание взять, критичное или некритичное?

   При выборе некритичного задания в неделю i вы получаете доход $\ell_i > 0$ долларов; при выборе критичного задания доход составит $h_i > 0$ долларов. Проблема в том, что группа может взять критичное задание в неделю i только в том случае, если в неделю $i - 1$ она не выполняла вообще никакого задания (критичного или некритичного); для такой ответственной работы необходима целая неделя подготовки. С другой стороны, если группа выполняла задание (любого типа) в неделю $i - 1$, ничто не помешает ей выполнить некритичное задание в неделю i.

   Итак, для заданной последовательности из n недель *план* задается выбором «некритичное задание», «критичное задание» или «нет задания» для каждой из n недель с тем свойством, что при выборе «критичное задание» для недели $i > 1$ в неделе $i - 1$ должно быть выбрано задание $i - 1$. (Допускается выбор критичного задания в неделю 1.) *Значение* плана определяется следующим образом: для каждого i значение увеличивается на l_i, если в неделю i было выбрано «некритичное задание», и значение увеличивается на h_i, если в неделю i было выбрано «критичное задание». (Если в неделю i не было выбрано никакое задание, значение увеличивается на 0.)

   Задача. Для заданных множеств значений $\ell_1, \ell_2, ..., \ell_n$ и $h_1, h_2, ..., h_n$ найти план с максимальным значением (такой план будет называться *оптимальным*).

   Пример. Допустим, $n = 4$, а значения l_i и h_i задаются следующей таблицей. В этом случае план с максимальным значением достигается выбором «нет задания» в неделю 1, выбором критичного задания в неделю 2 и выбором некритичного задания в недели 3 и 4. Значение такого плана составит 0 + 50 + 10 + 10 = 70.

	Неделя 1	**Неделя 2**	**Неделя 3**	**Неделя 4**
ℓ	10	1	10	10
h	5	50	5	1

   (а) Покажите, что следующий алгоритм не дает правильного решения этой задачи; для этого предложите экземпляр задачи, для которого алгоритм возвращает неправильный ответ.

```
Для итераций с i = 1 до n
   Если h_i + 1 > l_i + l_{i+1}
      Вывести "Не брать задание в неделю i"
      Вывести "Взять критичное задание в неделю i + 1"
      Продолжить с итерации i + 2
   Иначе
```

```
      Вывести "Взять некритичное задание в неделю i"
      Продолжить с итерации i+1
   Конец Если
Конец
```

Чтобы избежать проблем с выходом за границу массива, определим $h_i = l_i = 0$ при $i > n$.

Также приведите правильный алгоритм и укажите, какой результат находит приведенный алгоритм.

(b) Предложите эффективный алгоритм, который получает значения $l_1, l_2, ..., l_n$ и $h_1, h_2, ..., h_n$ и возвращает значение оптимального плана.

3. Имеется направленный граф $G = (V, E)$ с узлами $v_1, ..., v_n$. Граф G называется *упорядоченным*, если он обладает следующими свойствами:

(i) каждое ребро идет от узла с меньшим индексом к узлу с большим индексом. Иначе говоря, каждое направленное ребро имеет форму (v_i, v_j) с $i < j$;

(ii) из каждого узла, кроме v_n, выходит минимум одно ребро, то есть для каждого узла v_i с $i = 1, 2, ..., n - 1$ существует минимум одно ребро в форме (v_i, v_j).

Длина пути определяется количеством ребер, входящих в этот путь. В этом упражнении решается следующая задача (пример представлен на рис. 6.29):

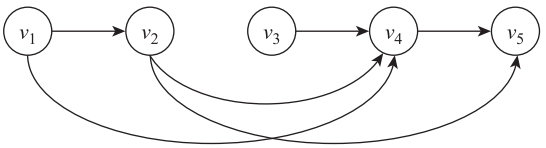


Рис. 6.29. Правильный ответ для этого упорядоченного графа равен 3: самый длинный путь из v_1 в v_n состоит из трех ребер (v_1,v_2),(v_2,v_4) и (v_4,v_5)

Для заданного графа G найдите длину самого длинного пути, который начинается в v_1 и заканчивается в v_n.

(a) Покажите, что следующий алгоритм не дает правильного решения этой задачи; для этого приведите пример упорядоченного графа, для которого алгоритм возвращает неправильный ответ.

```
Присвоить w = v1
Присвоить L = 0
Пока существует ребро, выходящее из узла w
   Выбрать ребро (w,vj)
      с минимально возможным значением j
   Присвоить w = vj
   Увеличить L на 1
Конец Пока
Вернуть L как длину самого длинного пути
```

В своем примере укажите правильный ответ и объясните, какой результат находит приведенный алгоритм.

(b) Предложите эффективный алгоритм, который получает упорядоченный граф G и возвращает *длину* самого длинного пути, который начинается в v_1 и заканчивается в v_n. (Еще раз: под *длиной* понимается количество ребер в пути.)

4. Вы руководите небольшой фирмой — вы, двое сотрудников и арендованная техника. Ваши клиенты распределены между Восточным и Западным побережьем.

В каждом месяце вы можете вести дела из офиса в Нью-Йорке (NY) или из офиса в Сан-Франциско (SF). В месяце i текущие издержки составляют N_i, если дела ведутся из Нью-Йорка, или S_i, если дела ведутся из Сан-Франциско (величина зависит от потребностей клиентов в этом месяце).

Но если в месяце i вы ведете дела из одного города, а в месяце $i + 1$ — из другого города, затраты на перемещение составляют фиксированную величину M.

Для заданной последовательности из n месяцев *план* представляет собой серию из n вариантов местонахождения (*NY* или *SF*); i-й элемент последовательности обозначает город, в котором вы будете вести дела в i-м месяце. *Стоимость* плана определяется как сумма текущих издержек для каждого из n месяцев и затрат M для каждого перемещения между городами. План может начинаться с любого города.

Задача. Для заданной стоимости перемещения M и последовательностей текущих издержек $N_1, ..., N_n$ и $S_1, ..., S_n$ найдите план с минимальной стоимостью (назовем такой план оптимальным).

Пример. Допустим, $n = 4$, $M = 10$, а текущие издержки задаются следующей таблицей.

	Месяц 1	**Месяц 2**	**Месяц 3**	**Месяц 4**
NY	1	3	20	30
SF	50	20	2	4

В этом случае план с минимальной стоимостью будет представлять собой последовательность [*NY*, *NY*, *SF*, *SF*] с общей стоимостью 1 + 3 + 2 + 4 + 10 = 20 (последнее слагаемое 10 появляется из-за однократного перемещения).

(a) Покажите, что следующий алгоритм не дает правильного решения этой задачи; для этого приведите пример экземпляра задачи, для которого алгоритм возвращает неправильный ответ.

```
For i = 1 To n
   Если N_i < S_i
      Вывести "NY в месяце i"
   Иначе
      Вывести "SF в месяце i"
Конец
```

В своем примере укажите правильный ответ и объясните, какой результат находит приведенный алгоритм.

(b) Приведите пример, в котором каждый оптимальный план требует не менее трех перемещений между городами.

Кратко объясните, почему ваш пример обладает таким свойством.

(c) Предложите эффективный алгоритм, который получает значения n, M и последовательности текущих издержек N_1, ..., N_n и S_1, ..., S_n и возвращает *стоимость* оптимального плана.

5. Как вам, вероятно, известно, в некоторых языках (включая китайский и японский) слова не разделяются пробелами. По этой причине программы, работающие с текстом, написанным на этих языках, должны решать проблему *сегментации слов*, то есть вычисления вероятных границ между смежными словами в тексте. Представьте, что английский текст записывается без пробелов; вы получаете строку вида «meetateight» и решаете, что лучшим разбиением будет «meet at eight» (а не «me et at eight», «meet ateight» или какой-нибудь менее вероятный вариант). Как автоматизировать этот процесс?

   Простой, но достаточно эффективный метод основан на поиске сегментации, которая максимизирует накапливаемое «качество» отдельных составляющих слов. Допустим, у вас есть «черный ящик», который для любой цепочки букв $x = x_1x_2 \ldots x_k$ возвращает число *quality*(x). Это число может быть как положительным, так и отрицательным; большие числа соответствуют более вероятным английским словам. (Значение *quality*("*me*") будет положительным, тогда как значение *quality*("*ght*") будет отрицательным).

   Для длинной цепочки букв $y = y_1y_2 \ldots y_n$ сегментация y представляет собой вариант разбиения последовательности на непрерывные блоки букв; каждый блок соответствует одному слову в сегментации. Общее качество сегментации вычисляется суммированием качества каждого блока. (Таким образом, правильный ответ для приводившейся выше строки будет получен при условии, что сумма *quality*("*meet*")+*quality*("*at*")+*quality*("*eight*") больше суммы любой другой сегментации строки.)

   Предложите эффективный алгоритм, который получает строку y и вычисляет сегментацию с максимальным общим качеством. (Одно обращение к «черному ящику» для вычисления *quality*(x) считается одним вычислительным шагом.)

   (Последнее замечание, не обязательное для решения задачи: для повышения быстродействия программы сегментации на практике используют более сложную формулировку задачи: например, предполагается, что решение не только должно быть разумным на уровне слов, но и образовывать связные обороты и предложения. Например, строку «theyouthevent» можно как минимум тремя способами разбить на обычные английские слова, но один из вариантов дает намного более осмысленную фразу, чем два других. Если рассматривать происходящее в терминологии формальных языков, расширенная задача напоминает поиск сегментации, которая также хорошо разбирается в соответствии с грамматикой используемого языка. Но даже с этими дополнительными критериями

и ограничениями в основе многих эффективных систем сегментации лежат методы динамического программирования.)

6. При форматировании текста часто встречается стандартная задача: взять текст с «рваным» правым краем:

```
Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.
```

и преобразовать его в текст, правый край которого «сглажен», насколько это возможно:

```
Call me Ishmael. Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

Чтобы формализовать эту задачу и начать размышлять над реализацией, сначала нужно определить, что же понимать под «сглаженным» правым краем. Допустим, текст состоит из последовательности слов $W = \{w_1, w_2, ..., w_n\}$, а слово w_i состоит из c_i символов. Максимальная длина строки равна L. Будем считать, что при выводе используется моноширинный шрифт, а знаки препинания и переносы не учитываются.

Форматирование W состоит из разбиения слов W на строки. В словах, назначенных одной строке, после каждого слова, кроме последнего, должен следовать пробел; соответственно если слова $w_j, w_{j+1}, ..., w_k$ назначены в одну строку, должно выполняться условие

$$\left[\sum_{i=j}^{k-1}(c_i + 1)\right] + c_k \le L.$$

Назовем назначение слов по строкам *действительным*, если оно удовлетворяет этому неравенству. Разность между левой и правой частью будет называться *допуском* строки (она равна количеству пробелов у правого края).

Предложите эффективный алгоритм для разбиения множества слов W на действительные строки, чтобы сумма квадратов допусков всех строк (включая последнюю) была минимальной.

7. В одном из упражнений с решениями главы 5 представлен алгоритм с временем выполнения $O(n \log n)$ для следующей задачи. Рассматриваются цены акций за n последовательных дней, пронумерованных $i = 1, 2, ..., n$. Для каждого дня i известна биржевая котировка этих акций $p(i)$ в этот день. (Для простоты будем

считать, что в течение дня котировка не изменяется.) Вопрос: как выбрать день i для покупки акций и последующий день j для их продажи, чтобы получить максимальную прибыль $p(j)-p(i)$? (Если за эти n дней перепродажа с прибылью невозможна, алгоритм должен сообщить об этом.)

В приведенном решении показано, как найти оптимальную пару дней i и j за время $O(n \log n)$. Однако на самом деле можно найти еще более эффективный алгоритм. Покажите, как найти оптимальные числа i и j за время $O(n)$.

8. Обитатели подземного города Зиона для защиты от нападающих машин применяют кунг-фу, тяжелую артиллерию и эффективные алгоритмы. Недавно они заинтересовались автоматизированными средствами отражения атак.

   Вот как проходит типичная атака:

   - Рой машин прибывает в течение n секунд; на i-й секунде появляются x_i роботов. Благодаря системе раннего оповещения последовательность $x_1, x_2, \ldots, x_n$ известна заранее.
   - В вашем распоряжении имеется электромагнитная пушка (ЭМП), которая может уничтожить часть роботов при появлении; мощность выстрела зависит от того, сколько времени заряжалась пушка. Или в более точной формулировке: имеется такая функция $f(\cdot)$, что по прошествии j секунд с момента последнего выстрела ЭМП может уничтожить до $f(j)$ роботов.
   - Итак, если пушка использовалась на k-й секунде и с момента последнего выстрела прошло j секунд, она может уничтожить $\min(x_k, f(j))$ роботов. (После использования пушка полностью разряжается).
   - Также будем считать, что в исходном состоянии ЭМП полностью разряжена, поэтому если она впервые используется на j-й секунде, она может уничтожить до $f(j)$ роботов.

   Задача. Для заданной последовательности прибытия роботов $x_1, x_2, \ldots, x_n$, и функции перезарядки $f(\cdot)$ выберите моменты времени, в которые следует активировать ЭМП для уничтожения максимального количества роботов.

   Пример. Допустим, $n = 4$, а значения x_i и $f(i)$ задаются следующей таблицей.

i	1	2	3	4
x_i	1	10	10	1
$f(i)$	1	2	4	8

   В оптимальном решении ЭМП активируется на 3 и 4 секундах. На 3 секунде ЭМП прошла 3-секундную зарядку и поэтому уничтожает $\min(10,4) = 4$ роботов; на 4 секунде с момента последнего выстрела прошла всего 1 секунда, поэтому уничтожается $min(1,1) = 1$ робот. В сумме получается 5.

   (а) Покажите, что следующий алгоритм не дает правильного решения этой задачи; для этого приведите пример экземпляра задачи, для которого алгоритм возвращает неправильный ответ.

```
Schedule-EMP(x_1, ..., x_n)
   Присвоить j наименьшее число, для которого f(j) ≥ x_n
      (если такого числа не существует, присвоить j = n)
   Активировать ЭМП на n-й секунде
   Если n − j ≥ 1
      Рекурсивно продолжить для данных x_1, ..., x_{n-j}
         (то есть вызвать Schedule-EMP(x_1, ..., x_{n-j}))
```

В своем примере укажите правильный ответ и объясните, какой результат находит приведенный алгоритм.

(b) Предложите эффективный алгоритм, который получает план прибытия роботов $x_1, x_2, ..., x_n$ и функцию перезарядки $f(\cdot)$ и возвращает максимальное количество роботов, которые могут быть уничтожены серией активаций ЭМП.

9. Вы участвуете в администрировании высокопроизводительной компьютерной системы, способной обрабатывать несколько терабайт данных в день. В каждый из n дней поступает различный объем данных; в день i вы получаете x_i терабайт. За каждый обработанный терабайт ваша фирма получает фиксированную оплату, но все необработанные данные становятся недоступными в конце дня (то есть их обработку нельзя продолжить на следующий день).

   Ежедневно обрабатывать все данные не удается из-за возможностей компьютерной системы, способной обрабатывать определенное количество терабайт в день. В системе работает уникальная программа — очень сложная, но не стопроцентно надежная, поэтому максимальный объем обрабатываемых данных убывает с каждым днем, прошедшим с момента последней перезагрузки. В первый день после перезагрузки система может обработать s_1 терабайт, во второй день — s_2 терабайта и т. д. вплоть до s_n; предполагается, что $s_1 > s_2 > s_3 > ... > s_n > 0$. (Конечно, в день i система может обработать не более x_i терабайт независимо от быстродействия вашей системы.) Чтобы вернуть систему к максимальной производительности, можно перезагрузить ее; однако в любой день, в который система будет перезагружаться, никакие данные вообще не обрабатываются.

   Задача. Для заданных объемов данных $x_1, x_2, ..., x_n$ на следующие n дней и профиля системы, описанного серией $s_1, s_2, ..., s_n$ (начиная с только что перезагруженной системы в день 1), выберите дни для проведения перезагрузки, чтобы максимизировать общий объем обрабатываемых данных.

   Пример. Допустим, $n = 4$, а значения x_i и s_i задаются следующей таблицей.

	День 1	**День 2**	**День 3**	**День 4**
x	10	1	7	7
s	8	4	2	1

   В оптимальном решении перезагрузка выполняется только в день 2; в этом случае в день 1 обрабатываются 8 терабайт, затем 0 терабайт в день 2, 7 терабайт в день 3 и 4 терабайта в день 4, итого 19. (Обратите внимание: без перезагрузки

было бы обработано только 8 + 1 + 2 + 1 = 12 терабайт, а с другими стратегиями перезагрузки также получается менее 19 терабайт.)

(a) Приведите пример, обладающий следующими свойствами:

- Объем поставляемых данных превышает возможности обработки, то есть $x_i > s_i$ для каждого i.
- В оптимальном решении система перезагружается минимум дважды.

Кроме примера, приведите оптимальное решение. Предоставлять доказательства его оптимальности не обязательно.

(b) Предложите эффективный алгоритм, который получает значения $x_1, x_2, ..., x_n$ и $s_1, s_2, ..., s_n$ и возвращает общее *количество* терабайт, обрабатываемых оптимальным решением.

10. Вы пытаетесь провести крупномасштабное моделирование физической системы, вычислительная модель которого должна содержать как можно больше дискретных *шагов*. В вашей лаборатории установлены два суперкомпьютера (назовем их A и B), способных выполнить эту работу. Тем не менее вы не являетесь привилегированным пользователем ни на одном из этих компьютеров и можете использовать только выделенные вам вычислительные ресурсы.

    При этом вы сталкиваетесь с проблемой: в любую минуту задание может работать только на одной из этих машин. Для каждой из ближайших n минут доступен «профиль», который описывает свободные вычислительные мощности на каждой машине. В минуту i можно провести $a_i > 0$ шагов моделирования, если задание выполняется на машине A, или $b_i > 0$ шагов, если задание выполняется на машине B. У вас также есть возможность передать задание с одной машины на другую, но передача занимает минуту времени, в течение которой задание не обрабатывается.

    Итак, для заданной последовательности из n минут план определяется выбором действия (A, B или переход) с тем свойством, что варианты A и B не могут следовать непосредственно друг за другом. Например, если в минуту i задание выполняется на машине A и вы хотите переключиться на машину B, то в минуту $i + 1$ должно быть выбрано действие перехода и только в минуту $i + 2$ можно будет выбрать B. *Значение* плана вычисляется как общее количество шагов моделирования, выполненных за n минут; таким образом, оно представляет собой сумму a_i по всем минутам, в течение которых задание выполняется на машине A, вместе с суммой b_i по всем минутам, в течение которых задание выполняется на машине B.

    Задача. Для заданных значений $a_1, a_2, ..., a_n$ и $b_1, b_2, ..., b_n$ найдите план с максимальным значением. (Такая стратегия будет называться оптимальной.) План может начинаться с любой из машин A и B в минуту 1.

    Пример. Допустим, $n = 4$, а значения $a_1, a_2, ..., a_n$ и $b_1, b_2, ..., b_n$ задаются следующей таблицей.

	Минута 1	Минута 2	Минута 3	Минута 4

A	10	1	1	10
B	5	1	20	20

В плане с максимальным значением в минуту 1 задание выполняется на машине *A*, затем в минуту 2 происходит переход, а в минуты 3 и 4 задание выполняется на машине *B*. Значение этого плана составит 10 + 0 + 20 + 20 = 50.

(a) Покажите, что следующий алгоритм не дает правильного решения этой задачи; для этого приведите пример экземпляра задачи, для которого алгоритм возвращает неправильный ответ.

```
В минуту 1 выбрать машину, на которой достигается большее значение из a_1, b_1
Присвоить i = 2
Пока i ≤ n
   Какой выбор был сделан в минуту i − 1?
   Если A:
      Если b_{i+1} > a_i + a_{i+1}
         Выбрать переход в минуту i и B в минуту i + 1
         Перейти к итерации i + 2
      Иначе
         Выбрать A в минуту i
         Перейти к итерации i + 1
      Конец Если
   Если B: действовать, как описано выше, со сменой ролей A и B
Конец Пока
```

В своем примере укажите правильный ответ и объясните, какой результат находит приведенный алгоритм.

(b) Предложите эффективный алгоритм, который получает значения $a_1, a_2, ..., a_n$ и $b_1, b_2, ..., b_n$, и возвращает значение оптимального плана.

11. Вы работаете на компанию, которая производит компьютерное оборудование и рассылает его фирмам-дистрибьюторам по всей стране. Для каждой из следующих n недель установлен запланированный объем поставки s_i (в килограммах) оборудования, которое пересылается грузовым авиаперевозчиком.

- Грузы каждой недели могут перевозиться одной из двух фирм-перевозчиков, *A* или *B*. Компания *A* взимает фиксированную плату r за килограмм (так что стоимость еженедельной поставки составляет $r \cdot s_i$).
- Компания *B* заключает контракты на фиксированную сумму c независимо от веса. При этом контракты с компанией *B* должны заключаться сразу на четыре недели подряд.

Назовем *графиком* выбор авиаперевозчика (*A* или *B*) для каждой из n недель — с тем ограничением, что компания *B*, когда она выбирается, должна выбираться на четыре недели подряд. *Стоимость* графика вычисляется как общая сумма, выплаченная компаниям *A* и *B* в соответствии с приведенным выше описанием.

Предложите алгоритм с полиномиальным временем, который получает последовательность объемов поставок $s_1, s_2, ..., s_n$ и возвращает *график* с минимальной стоимостью.

Пример. Допустим, $r = 1$, $c = 10$, а серия поставок имеет вид

$$11, 9, 9, 12, 12, 12, 12, 9, 9, 11.$$

В этом случае в оптимальном графике компания *A* выбирается на первые три недели, затем компания *B* на четыре следующие недели и снова компания *A* на три последние недели.

12. Допустим, вы хотите реплицировать файл в группе из n серверов, обозначенных $S_1, S_2, ..., S_n$. За размещение копии файла на сервере S_i вносится целочисленная оплата $c_i > 0$.

    Если пользователь запрашивает файл с сервера S_i, а копия файла на этом сервере отсутствует, система по порядку проверяет серверы $S_{i+1}, S_{i+2}, S_{i+3}, ...$, пока не найдет копию файла — например, на сервере S_j, где $j > i$. Передача файла с другого сервера требует оплаты в размере $j–i$. (Обратите внимание: серверы с меньшими индексами $S_{i–1}, S_{i–2}, ...$ в поиск не включаются). Стоимость передачи равна 0, если копия файла доступна на сервере S_i. Копия файла обязательно размещается на сервере *Sn*, так что все операции поиска гарантированно завершатся на S_n.

    Требуется разместить копии файлов на серверах так, чтобы сумма оплаты за размещение и передачу файлов была минимальной. Формально *конфигурацией* называется решение о том, должна ли размещаться копия файла на каждом сервере S_i для $i = 1, 2, ..., n - 1$. (Еще раз напомним, что копия всегда размещается на сервере S_i.) Общая стоимость конфигурации вычисляется как сумма всех оплат за размещение на серверах с копией файла и сумма всех оплат за передачу файлов, связанных с n серверами.

    Предложите алгоритм с полиномиальным временем для поиска конфигурации с минимальной общей стоимостью.

13. Задача поиска циклов в графах естественным образом встречается в системах электронной торговли. Допустим, фирма выполняет операции с акциями n разных компаний. Для каждой пары $i \neq j$ поддерживается обменный курс r_{ij}, который означает, что одна акция i обменивается на r_{ij} акций j. Курс может быть дробным: иначе говоря, $r_{ij} = 2/3$ означает, что вы можете отдать три акции i для получения двух акций j.

    Обменный цикл для серии акций $i_1, i_2, ..., i_k$ состоит из последовательного обмена акций компании i_1 на акции компании i_2, затем акций компании i_2 на акции компании i_3, и т. д.; в конечном итоге акции i_k обмениваются на акции i_1. После такой серии обменов фирма получает акции той же компании i_1, с которой все начиналось. Циклические операции обычно нежелательны, поскольку, как правило, в результате у вас остается меньше акций, чем было в самом начале. Но время от времени на короткий промежуток времени открывается возможность увеличить количество акций. Цикл, операции по которому приводят

к увеличению количества исходных акций, будем называть *благоприятным циклом*. Такая возможность открывается в точности тогда, когда произведение всех дробей в цикле превышает 1. Фирма анализирует состояние рынка и хочет знать, существуют ли на нем благоприятные циклы.

Предложите алгоритм с полиномиальным временем для поиска благоприятного цикла (если он существует).

14. Большая группа мобильных устройств может образовать естественную сеть, в которой устройства являются узлами, а два устройства x и y соединяются ребром, если они могут напрямую взаимодействовать друг с другом (например, по каналу радиосвязи ближнего действия). Такие сети беспроводных устройств чрезвычайно динамичны: ребра появляются и исчезают при перемещении устройства. Например, ребро (x, y) может исчезнуть, когда x и y отдалятся друг от друга и потеряют возможность взаимодействовать напрямую.

    В сети, изменяющейся со временем, естественно возникает задача поиска эффективных способов *поддержания* путей между заданными узлами. При поддержании такого пути приходится учитывать два противоположных фактора: пути должны быть короткими, но при этом путь не должен часто изменяться при изменении структуры сети. (Другими словами, нам хотелось бы, чтобы один путь продолжал работать даже при добавлении и исчезновении ребер, если это возможно.) Ниже описан возможный способ моделирования этой задачи.

    Предположим, имеется множество мобильных узлов V, в какой-то момент времени существует множество E_0 ребер между этими узлами. При перемещении узлов множество ребер преобразуется из E_0 в E_1, затем в E_2, затем в E_3 и т. д., вплоть до множества ребер E_b. Для $i = 0, 1, 2, ..., b$ граф (V, E_i) обозначается G_i. Итак, если рассматривать структуру сети на базе узлов V «по кадрам», она будет представлять собой последовательность графов $G_0, G_1, G_2, ..., G_{b-1}, G_b$. Предполагается, что все эти графы G являются связными.

    Теперь рассмотрим два конкретных узла $s, t \in V$. Для пути s–t P в одном из графов G_i *длина* P определяется как количество ребер в P; обозначим ее $\ell(P)$. Наша цель — построить такую последовательность путей $P_0, P_1, ..., P_b$, чтобы для каждого i существовал путь s–t P_i в G_i. Пути должны быть относительно короткими. Кроме того, нежелательно слишком большое количество изменений — точек, в которых изменяется структура пути. Формально определим $changes(P_0, P_1, ..., P_b)$ как количество индексов i $(0 \le i \le b - 1)$, для которых $P_i \neq P_{i+1}$.

    Возьмем фиксированную константу $K > 0$. Стоимость последовательности путей $P_0, P_1, ..., P_b$ определяется по формуле

$$\text{cost}(P_0, P_1, \ldots, P_b) = \sum_{i=0}^{b} l(P_i) + K \cdot changes(P_0, P_1, ..., P_b).$$

    (а) Предположим, можно выбрать один путь P, который является путем s–t в каждом из графов $G_0, G_1, ..., G_b$. Предложите алгоритм с полиномиальным временем для нахождения кратчайшего из таких путей.

(b) Предложите алгоритм с полиномиальным временем для нахождения последовательности путей $P_0, P_1, ..., P_b$ с минимальной стоимостью, где P_i является путем s–t в G_i для $i = 0, 1, ..., b$.

15. В ясные дни ваши друзья с астрономического факультета собираются и договариваются о том, какие астрономические события они будут наблюдать этой ночью. Об этих событиях известно следующее.

- Всего существуют n событий; для простоты мы будем считать, что они происходят последовательно с интервалом ровно в одну минуту. Таким образом, событие j происходит в минуту j; если астрономы не наблюдают за ним ровно в минуту j, то они пропустят это событие.
- Небесный свод размечен в одномерной координатной системе (в градусах от некоторой центральной базовой линии); событие j происходит в точке с координатой d_j для некоторого целого d_j. В минуту 0 телескоп направлен в координату 0.
- Последнее событие n намного важнее других; наблюдать за ним обязательно.

У астрономического факультета имеется большой телескоп, в который можно наблюдать все эти события. Телескоп устроен очень сложно и может поворачиваться только на один градус в минуту. Астрономы не надеются пронаблюдать за всеми n событиями; они хотят всего лишь увидеть как можно больше с учетом ограничений телескопа и требования об обязательном наблюдении последнего события.

Назовем подмножество S событий *наблюдаемым*, если возможно наблюдать каждое событие $j \in S$ в назначенное время j и у телескопа хватает времени (с поворотом максимум на один градус в минуту) на перемещение между последовательными событиями S.

Задача. По координатам каждого из n событий найдите наблюдаемое подмножество максимального размера с учетом требования, что оно должно содержать событие n. Назовем такое решение *оптимальным*.

Пример. Одномерные координаты событий приведены в следующей таблице.

События	1	2	3	4	5	6	7	8	9
Координаты	1	–4	–1	4	5	–4	6	7	–2

В этом случае в оптимальном решении наблюдаются события 1, 3, 6 и 9. Обратите внимание: у телескопа хватает времени на перемещение от одного события к другому, хотя он и поворачивается всего на один градус в минуту.

(a) Покажите, что следующий алгоритм не дает правильного решения этой задачи; для этого приведите пример экземпляра задачи, для которого алгоритм возвращает неправильный ответ.

```
Пометить все события j with |d_n − d_j| > n − j как недопустимые (так как
   их наблюдение не позволит наблюдать событие n)
Пометить все остальные события как допустимые
```

```
Инициализировать текущую позицию координатой 0 для минуты 0
Пока не будет достигнут конец последовательности
   Найти самое раннее допустимое событие j, которое может быть достигнуто
      без превышения максимальной скорости поворота телескопа
   Добавить j в множество S
   Обновить текущую позицию координатой d_j в минуту j
Конец Пока
Вывести множество S
```

В своем примере укажите правильный ответ и объясните, какой результат находит приведенный алгоритм.

(b) Предложите эффективный алгоритм, который получает координаты $d_1, d_2, ..., d_1, d_2, ..., d_n$ событий и возвращает *размер* оптимального решения.

16. В Итаке (штат Нью-Йорк) много солнечных дней; но в этом году весенний пикник фирмы, в которой работают ваши друзья, пришелся на ненастье. Директор фирмы решает отменить пикник, но для этого нужно оповестить всех участников по телефону. Вот как он решает это сделать.

У каждого работника фирмы, кроме директора, есть начальник. Таким образом, иерархия может быть описана деревом T, корнем которого является директор, а у любого другого узла v имеется родительский узел u, представляющий его начальника. Соответственно, мы будем называть *v прямым подчиненным* по отношению к u. На рис. 6.30 узел A обозначает директора, узлы B и D — прямых подчиненных A, а узел C — прямого подчиненного B.

Чтобы оповестить всех об отмене пикника, директор сначала поочередно звонит каждому из своих прямых подчиненных. Сразу же после звонка подчиненный должен оповестить каждого из своих прямых подчиненных (тоже поочередно). Процесс продолжается до тех пор, пока не будут оповещены все сотрудники. Обратите внимание: каждый участник процесса звонит только своим прямым подчиненным: например, на рис. 6.30 директор A не будет звонить C.

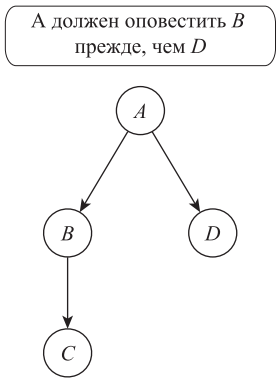


Рис. 6.30. Иерархия с четырьмя участниками. В самой быстрой схеме оповещения A звонит B в первом раунде. Во втором раунде A звонит D, а B звонит C. Если бы A начал со звонка D, то C не узнал бы о новостях до третьего шага

Можно рассматривать процесс распространения информации как разделенный на раунды. В одном раунде каждый работник, уже знающий об отмене, звонит одному из своих прямых подчиненных по телефону. Количество раундов, необходимое для оповещения всех участников, зависит от последовательности, в которой каждый участник звонит своим прямым подчиненным. Например, на рис. 6.30 оповещение будет состоять из двух раундов, если A сначала позвонит B, но займет целых три раунда, если A начнет со звонка D.

Предложите эффективный алгоритм для определения минимального количества раундов, необходимого для оповещения всех участников. Алгоритм должен выводить последовательность телефонных звонков, при которой достигается этот минимум.

17. Ваши друзья занимаются анализом динамики цен на акции технологических фирм. Они определили закономерность, которая была названа *возрастающим трендом*; формальное описание этой закономерности приводится ниже.

Известна последовательность цен на некие акции $P[1]$, $P[2]$, ..., $P[n]$, зафиксированных на момент закрытия биржи для n последовательных дней. Возрастающим трендом в этой серии называется такая подпоследовательность $P[i_1], P[i_2], ..., P[i_k]$ для дней $i_1 < i_2 < ... < i_k$, у которой:

- $i_1 = 1$;
- $P[i_j] < P[i_{j+1}]$ для всех $j = 1, 2, ..., k - 1$.

Таким образом, возрастающий тренд представляет собой подпоследовательность дней (начинающуюся с первого дня и не обязательно непрерывную), в которой цена строго возрастает.

Требуется найти самый длинный возрастающий тренд для заданной последовательности цен.

Пример. Допустим, $n = 7$, а последовательность цен выглядит так:

$$10, 1, 2, 11, 3, 4, 12.$$

В этом случае самый длинный возрастающий тренд образуется в дни 1, 4 и 7. Обратите внимание: дни 2, 3, 5 и 6 тоже образуют серию возрастающих цен, но так как эта подпоследовательность не начинается с дня 1, она не соответствует определению возрастающего тренда.

(а) Покажите, что следующий алгоритм не всегда возвращает *длину* самого длинного возрастающего тренда; для этого приведите пример экземпляра задачи, для которого алгоритм возвращает неправильный ответ.

```
Определить i = 1
     L = 1
For j = 2 to n
   Если If P[j] > P[i]
      Присвоить i = j.
      Увеличить L на 1.
   Конец Если
Конец For
```

В своем примере приведите длину самого длинного возрастающего тренда и укажите, какой результат находит приведенный алгоритм.

(b) Предложите эффективный алгоритм, который получает последовательность цен $P[1], P[2], ..., P[n]$ и возвращает *длину* самого длинного возрастающего тренда.

18. Рассмотрим задачу выравнивания последовательностей для алфавита из четырех букв $\{z_1, z_2, z_3, z_4\}$ с заданными стоимостями разрыва и несовпадения. Предположим, каждый из этих параметров является положительным целым числом.

Имеются две строки $A = a_1a_2...a_m$ и $B = b_1b_2...b_n$ с предложенным выравниванием между ними. Предложите алгоритм $O(mn)$, который решит, является ли выравнивание уникальным выравниванием минимальной стоимости между A и B.

19. Вы консультируете группу специалистов (лучше обойтись без подробностей), занимающихся отслеживанием и анализом электронных сигналов от кораблей в прибрежных водах Атлантики. Они хотят получить быстрый алгоритм для базовой операции, который встречается достаточно часто: «распутывание» суперпозиции двух известных сигналов. А именно, ваши работодатели представляют ситуацию, в которой два корабля снова и снова передают короткую последовательность из 0 и 1; нужно убедиться в том, что полученный сигнал представляет собой результат простого *чередования* этих двух передач и в него не добавлено ничего лишнего.

Немного уточним формулировку задачи. Для заданной строки x, состоящей из 0 и 1, x_k обозначает результат конкатенации k копий строки x. Мы будем называть строку x' *повторением* x, если она является префиксом x_k для некоторого числа k. Таким образом, $x' = 10110110110$ является повторением $x = 101$.

Строка s называется *чередованием* x и y, если ее символы можно разбить на две (необязательно смежные) последовательности s' и s'', так что s' является повторением x, а s'' является повторением y. (При этом каждый символ s принадлежит либо s', либо s''). Например, если $x = 101$ и $y = 00$, строка $s = 100010101$ является чередованием x и y, поскольку символы 1, 2, 5, 7, 8, 9 образуют серию 101101 (повторение x), а оставшиеся символы 3, 4 и 6 образуют серию 000 — повторение y.

В контексте нашего приложения x и y являются повторяющимися последовательностями, полученными от двух кораблей, а s — прослушиваемый сигнал: нужно убедиться в том, что s «разворачивается» в простые повторения x и y. Предложите эффективный алгоритм, который получает строки s, x и y и решает, является ли s чередованием x и y.

20. Семестр близится к завершению, и вам предстоит выполнить курсовые проекты по n учебным курсам. За каждый проект будет выставлена оценка — целое число в диапазоне от 1 до $g > 1$; более высокие числа означают лучшие оценки. Разумеется, вы стремитесь к максимизации средней оценки по n проектам.

Вы располагаете временем $H > n$ часов для работы над проектами (с перерывами); нужно решить, как распределить это время. Для простоты будем считать, что H — положительное целое число и на каждый проект расходуется целое количество часов. Чтобы определить, как лучше распределить время, вы опре-

деляете множество функций $\{f_i: i = 1, 2, ..., n\}$ (конечно, оценки весьма приблизительные) для каждого из n проектов: потратив $h \le H$ часов на проект по курсу i, вы получите оценку $f_i(h)$. (Функции f_i не убывают: если $h < h'$, то $f_i(h) \le f_i(h')$.)

Задача. Для заданный функций $\{f_i\}$ решите, сколько часов следует потратить на каждый проект (только в целых числах), чтобы средняя оценка, вычисленная в соответствии с f_i, была как можно больше. Для повышения эффективности время выполнения алгоритма должно быть полиномиальным по n, g и H: ни одна из этих величин не должна входить в экспоненциальную зависимость времени выполнения.

21. Совсем недавно вы помогали своим друзьям, которые занимались обменом акций, а они пришли к вам с новой задачей. Имеются данные по курсу некоторых акций за n дней подряд (в какое-то время в прошлом). Дни пронумерованы $i = 1, 2, ..., n$; для каждого дня i известна цена акции $p(i)$ за этот день.

    Для некоторых (возможно, больших) значений k требуется проанализировать так называемые *k-операционные стратегии* — множества из m пар за дни $(b_1,s_1), ..., (b_m, s_m)$, где $0 \le m \le k$ и

    $$1 \le b_1 < s_1 < b_2 < s_2 \ldots < b_m < s_m \le n.$$

    Они рассматриваются как множество из k неперекрывающихся интервалов, во время каждого из которых инвесторы покупают 1000 акций на бирже (в день b_i), а затем продают их (в день s_i). *Доход* от k-операционной стратегии определяется как прибыль, полученная в результате m операций купли-продажи, а именно

    $$1000\sum_{i=1}^{m} p(s_i) - p(b_i).$$

    Инвесторы хотят оценить значение k-операционных стратегий по результатам моделирования своих n-дневных данных котировок. Ваша цель — спроектировать эффективный алгоритм, который определяет для заданной серии цен k-операционную стратегию с максимально возможным доходом. Так как значение k в этих моделях может быть достаточно большим, время выполнения должно быть полиномиальным по n и k (значение k не должно присутствовать в экспоненте).

22. Чтобы оценить, насколько «надежно связаны» два узла в направленном графе, можно рассмотреть не только длину кратчайшего пути между ними, но и количество кратчайших путей.

    Как выясняется, эта задача имеет эффективное решение при некоторых ограничениях на стоимости ребер. Допустим, имеется направленный граф $G = (V, E)$ со стоимостями ребер; стоимости могут быть как положительными, так и отрицательными, но каждый цикл в графе имеет строго положительную стоимость. Также даны два узла $v, w \in V$. Предложите эффективный алгоритм для вычисления количества кратчайших путей v–w в G. (Алгоритм не должен выводить все пути; достаточно только одного количества.)

23. Имеется направленный граф $G = (V, E)$ со стоимостями ребер c_e для $e \in E$ и стоком t (стоимости могут быть отрицательными). Также предполагается наличие конечных значений $d(v)$ для $v \in V$. Утверждается, что для каждого узла $v \in V$ величина $d(v)$ определяет стоимость пути с минимальной стоимостью от узла v к стоку t.

    (a) Предложите алгоритм с линейным временем (время $O(m)$, если граф содержит m ребер), который проверяет правильность этого утверждения.

    (b) Предположите, что расстояния верны, а $d(v)$ конечны для всех $v \in V$. Теперь требуется вычислить расстояния до другого стока t'. Предложите алгоритм $O(m \log n)$ для вычисления расстояний $d'(v)$ от всех узлов $v \in V$ к стоку t'. (Подсказка: будет полезно рассмотреть новую функции стоимости, которая определяется следующим образом: для ребра $e = (v, w)$ пусть $c'_e = c_e - d(v) + d(w)$. Связаны ли между собой стоимости путей для двух разных c и c'?)

24. *Перекраиванием* называется практика тщательно продуманного изменения избирательных округов, при котором результат оказывается угодным конкретной политической партии. В ходе разбирательств по недавним судебным искам было показано, что тщательное перекраивание приводит к фактическому (и намеренному) лишению избирательного права больших групп избирателей.

    Как выясняется, при описании этой темы в новостях «источником зла» называют компьютеры: благодаря мощным программам перекраивание превратилось из занятия группы людей с картами, карандашами и блокнотами в высокотехнологичный процесс. Почему перекраивание стало вычислительной задачей? Базы данных позволяют отследить демографию избирателей до уровня отдельных улиц и домов, а группировка избирателей по округам создает некоторые проблемы чисто алгоритмического плана.

    Допустим, имеется множество из n участков $P_1, P_2, ..., P_n$, каждый из которых содержит m зарегистрированных избирателей. Требуется разделить эти участки на два *округа*, каждый из которых состоит из $n/2$ участков. Для каждого участка доступна информация о том, сколько избирателей поддерживают каждую из двух политических партий. (Для простоты будем считать, что каждый избиратель поддерживает одну из двух партий.) Множество участков будет подвержено перекраиванию, если возможно выполнить разбиение на два округа таким образом, что одна партия будет иметь большинство в обоих округах.

    Предложите алгоритм для определения того, подвержено ли заданное множество участков перекраиванию; время выполнения алгоритма должно быть полиномиальным по n и m.

    Пример. Допустим, имеется $n = 4$ участков и доступна следующая информация о зарегистрированных избирателях.

Участок	**1**	**2**	**3**	**4**
Количество зарегистрированных сторонников партии A	55	43	60	47
Количество зарегистрированных сторонников партии B	45	57	40	53

Это множество участков создает опасность перекраивания: если сгруппировать участки 1 и 4 в один округ, а участки 2 и 3 — в другой, партия А получит большинство в обоих округах (предполагается, что группировку выполняют члены партии А). Этот пример наглядно демонстрирует изначальную несправедливость перекраивания: хотя партия А имеет небольшое преимущество в общем населении (205 к 195), она получает преимущество не в одном, а в обоих округах.

25. Биржевой брокер пытается продать большое количество акций, цена которых неуклонно снижается. Всегда трудно спрогнозировать идеальный момент для продажи акций, но держатели большого количества акций одной компании сталкиваются с дополнительной проблемой: сам факт продажи большого количества акций в один день отрицательно влияет на цену.

    Будущие рыночные цены и эффект от продажи большого количества акций очень трудно спрогнозировать, брокерские фирмы используют модели рынка для принятия решений. В этой задаче рассматривается очень простая модель: допустим, нужно продать x акций определенного вида и имеется достаточно точная модель рынка: она предсказывает, что цена акций будет принимать значения $p_1, p_2, ..., p_n$ в ближайшие n дней. Кроме того, имеется функция $f(\cdot)$, прогнозирующая эффект масштабных продаж: если продать y акций в один день, цена с этого дня стабильно снижается на $f(y)$. Итак, при продаже y_1 акций в день 1 цена акций снижается до $p_1 - f(y_1)$ с суммарным доходом $y_1 \cdot (p_1 - f(y_1))$. После продажи y_1 акций в день 1 можно продать y_2 акций в день 2 по цене $p_2 - f(y_1) - f(y_2)$; при этом будет получен дополнительный доход в размере $y_2 \cdot (p_2 - f(y_1) - f(y_2))$. Процесс продолжается все n дней. (Обратите внимание: как и вычислениях для дня 2, уменьшение цены за предыдущие дни включается в цену на все последующие дни).

    Разработайте эффективный алгоритм, который получает цены $p_1, ..., p_n$ и функцию $f(\cdot)$ (записанную в виде списка значений $f(1), f(2), ..., f(x)$), и определяет лучший вариант продажи x акций ко дню n. Иначе говоря, найдите такие натуральные числа $y_1, y_2, ..., y_n$, что $x = y_1 + ... + y_n$ и продажа y_i акций в день i для $i = 1, 2, ..., n$ максимизирует суммарный доход. Стоимость акций p_i монотонно убывает, а $f(\cdot)$ монотонно возрастает; иначе говоря, продажа большего количества акций приводит к более значительному снижению цены. Время выполнения алгоритма может находиться в полиномиальной зависимости от n (количество дней), x (количество акций) и p_1 (пиковая цена акций).

    Пример. Допустим, $n = 3$; цены за три дня были равны 90, 80, 40; $f(y) = 1$ для $y \le 40\,000$ и $f(y) = 20$ для $y > 40\,000$. Предположим, вы начинаете с $x = 100\,000$ акций. При продаже всех акций в день 1 цена составит 70 за акцию, а общий доход — 7 000 000. Если же продать 40 000 акций в день 1 по цене 89 за акцию, а затем продать остальные 60 000 акций в день 2 по цене 59 за акцию, общий доход составит 7 100 000.

26. Вы руководите компанией, которая продает некий крупный товар (допустим, грузовики). Аналитики предоставляют вам прогнозы с предполагаемым объемом продаж за следующие n месяцев. Обозначим d_i предполагаемый объем

продаж в месяце i. Будем считать, что все продажи происходят в начале месяца, а непроданные грузовики хранятся на складе до начала следующего месяца. На складе помещается максимум S грузовиков, а затраты на хранение одного грузовика в течение месяца равны C. Чтобы получить партию грузовиков с завода, вы размещаете заказ. За каждое размещение заказа (независимо от количества заказанных грузовиков) взимается фиксированная плата K. Изначально на складах нет ни одного грузовика. Требуется разработать алгоритм, который решает, как разместить заказы для выполнения всех требований $\{d_i\}$ с минимальными издержками. Вкратце:

- расходы складываются из двух составляющих: 1) хранение — за каждый грузовик на складе, который не продан в этом месяце, приходится платить C; 2) размещение заказа — за каждый заказ приходится платить K;
- в каждом месяце необходимо иметь достаточно грузовиков, чтобы удовлетворить спрос d_i, но количество оставшихся после этого грузовиков не должно превысить емкость склада S.

Предложите алгоритм, решающий эту задачу за время, полиномиальное по n и S.

27. Владелец бензоколонки столкнулся со следующей проблемой: имеется большой подземный бак, в котором хранится бензин; емкость бака составляет L галлонов. Заказы обходятся дорого, поэтому они оформляются относительно редко. Для каждого заказа владелец должен заплатить фиксированную цену P (кроме стоимости заказанного бензина). Однако хранение галлона в течение дня требует затрат c, поэтому слишком большие запасы увеличивают расходы на хранение.

    Владелец бензоколонки собирается закрыть ее на неделю зимой. К моменту закрытия бак должен быть пустым. К счастью, на основании многолетнего опыта он может точно предсказать, сколько бензина будет расходоваться ежедневно до заданного момента. Допустим, что до закрытия осталось n дней и в день i ($i = 1, ..., n$) будет расходоваться g_i галлонов. Предполагается, что в конце дня 0 бак пуст. Предложите алгоритм, который решает, в какие дни следует размещать заказы и сколько бензина нужно заказывать для минимизации общих затрат.

28. Вспомните задачу планирования из раздела 4.2, в которой мы стремились к минимизации максимальной задержки. Существуют n заданий, каждое задание имеет предельное время d_i и необходимое время обработки t_i; все задания доступны для планирования, начиная с времени s. Чтобы задание i было выполнено, ему необходимо выделить период времени от $s_i \geq s$ до $f_i = s_i + t_i$, причем разным заданиям должны назначаться неперекрывающиеся интервалы. Такое распределение времени называется *расписанием*.

    В этой задаче рассматривается аналогичная ситуация, но оптимизируется другая цель. А именно, мы рассмотрим случай, при котором каждое задание должно быть либо выполнено к предельному времени, либо не выполнено вообще. Подмножество J называется *планируемым*, если для заданий в J существует расписание, при котором каждое задание будет завершено к предельному времени. Требуется выбрать планируемое подмножество максимального возможного

размера и найти для этого подмножества расписание, позволяющее завершить каждое задание к предельному времени.

- (a) Докажите, что существует оптимальное решение J (то есть планируемое множество максимального размера), в котором задания J планируются по возрастанию предельного времени.
- (b) Предположим, все значения предельного времени d_i и необходимого времени t_i являются целыми числами. Предложите алгоритм для поиска оптимального решения. Алгоритм должен выполняться за время, полиномиальное по количеству заданий n и максимальному предельному времени $D = \max_i d_i$.

29. Граф $G = (V, E)$ состоит из n узлов, каждая пара узлов соединена ребром. Каждому ребру (i, j) присвоен положительный вес w_{ij}; предполагается, что веса удовлетворяют *неравенству треугольника* $w_{ik} \leq w_{ij}+w_{jk}$. Для подмножества $V' \subseteq V$ запись $G[V']$ обозначает подграф (с весами ребер), построенный для узлов из V'.

    Дано множество $X \subseteq V$ из k терминальных узлов, которые должны быть соединены ребрами. *Деревом Штейнера* для X называется такое множество Z, что $X \subseteq Z \subseteq V$ в сочетании с остовным поддеревом T подграфа $G[Z]$. *Весом* дерева Штейнера называется вес дерева T.

    Покажите, что существует $f(\cdot)$ и полиномиальная функция $p(\cdot)$, для которых задача нахождения дерева Штейнера с минимальным весом для X может быть решена за время $O(f(k)\cdot p(n))$.

Примечания и дополнительная литература

Первопроходцем в области систематического изучения динамического программирования был Ричард Беллман (Bellman, 1957); алгоритм сегментированной задачи наименьших квадратов, приведенный в этой главе, основан на одной из ранних работ Беллмана (Bellman, 1961). С тех пор динамическое программирование поднялось до уровня метода, широко используемого в компьютерной науке, исследовании операций, теории управления и ряде других областей. Значительная часть последних разработок по теме относится к стохастическому динамическому программированию: если в нашей формулировке задачи неявно предполагалось, что все входные данные известны с самого начала, многие задачи в области планирования, производства и складского учета и в других областях включают неопределенность, а алгоритмы динамического программирования для этих задач формулируют эту неопределенность по вероятностной формуле. Введение в стохастическое динамическое программирование приведено в книге Росса (Ross, 1983).

В области комбинаторной оптимизации изучались многие расширения и вариации на тему задачи о рюкзаке. Как упоминалось в этой главе, псевдополиномиальная граница, встречающаяся в области динамического программирования, может стать неприемлемой при большом размере входных данных; в таких случаях динамическое программирование часто объединяется с другими эвристиками для

практического решения больших экземпляров задачи о рюкзаке. Книга Мартелло и Тота (Martello, Toth, 1990) посвящена вычислительным методам решения разных вариантов задачи о рюкзаке.

Динамическое программирование заняло место среди базовых методов вычислительной биологии в начале 1970-х годов в результате интенсивного изучения задачи сравнения последовательностей. Санкофф (Sankoff, 2000) приводит интересные исторические данные о ранних работах этого периода. В книгах Уотермена (Waterman, 1995) и Гасфилда (Gusfield, 1997) приводится подробное описание алгоритмов выравнивания последовательностей (а также многих сопутствующих алгоритмов в вычислительной биологии); Мэтьюз и Цукер (Mathews, Zukor, 2004) обсуждают будущее методов решения задачи предсказания вторичной структуры РНК. Алгоритм выравнивания последовательностей, эффективных по затратам памяти, был предложен Хиршбергом (Hirschberg, 1975).

Алгоритм для задачи нахождения кратчайшего пути, описанный в этой главе, основан на исходной работе Беллмана (Bellman, 1958) и Форда (Ford, 1956). Этот базовый метод поиска кратчайших путей был дополнен множеством оптимизаций, обусловленных как теоретическими, так и экспериментальными соображениями; на сайте Эндрю Голдберга приведена новейшая версия кода, написанного им для решения этой задачи (а также ряда других) по материалам работы Черкасски, Голдберга и Радзика (Cherkassky, Goldberg, Radzik, 1994). Практическое применение методов нахождения кратчайших путей для маршрутизации в Интернете, а также достоинства и недостатки разных алгоритмов сетевых приложений описаны в книгах Бертсекаса и Галлагера (Bertekas, Gallager, 1992), Кешава (Keshav, 1997) и Стюарта (Stewart, 1998).

Примечания к упражнениям

Упражнение 5 было создано по мотивам обсуждения с Лилиан Ли; упражнение 6 основано на результатах Дональда Кнута; упражнение 25 основано на результатах Димитриса Бертсимаса и Эндрю Ло; упражнение 29 основано на результатах С. Дрейфуса и Р. Вагнера.

Глава 7
Нахождение потока в сети

В этой главе мы займемся семейством алгоритмических задач, которые в каком-то смысле происходят от одной из исходных задач, сформулированных в начале курса: *двудольных паросочетаний*.

Вспомните исходную формулировку задачи двудольных паросочетаний. *Двудольным графом* $G = (V, E)$ называется ненаправленный граф, множество узлов которого может быть разбито на подмножества $V = X \cup Y$, обладающее тем свойством, что один конец каждого ребра принадлежит X, а другой конец принадлежит Y. Двудольные графы часто изображаются так, как показано на рис. 7.1: узлы X выстраиваются в столбец слева, узлы Y — в столбец справа, а ребра переходят между левым и правым столбцом.

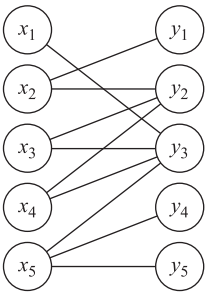


Рис. 7.1. Двудольный граф

Концепция *паросочетаний* уже неоднократно встречалась в этом учебном курсе: этим термином обозначаются совокупности пар в множествах, в которых ни один элемент множества не входит более чем в одну пару. (Вспомните мужчин и женщин из задачи устойчивых паросочетаний или символы из задачи выравнивания последовательностей.) В случае графа ребра образуют пары узлов, а мы соответственно говорим, что паросочетание в графе $G = (V, E)$ представляет собой множество ребер $M \subseteq E$ с тем свойством, что каждый узел присутствует не более чем в одном ребре M. Множество ребер M является *идеальным* паросочетанием, если каждый узел присутствует ровно в одном ребре M.

Паросочетания в двудольных графах используются для моделирования ситуаций, в которых одни объекты *назначаются* другим объектам. Вы уже видели

примеры таких ситуаций в предшествующих обсуждениях графов и двудольных графов. В одном из естественных такого рода узлы X представляют задания, узлы Y представляют машины, а ребро (x_i, y_j) означает, что машина y_j способна обработать задание x_i. В этом случае идеальное паросочетание определяет способ распределения заданий между машинами, способными их обработать, при котором каждой машине назначается ровно одно задание. Двудольные графы также могут использоваться для представления других отношений, возникающих между двумя разными множествами объектов: отношений между покупателями и магазинами; домами и обслуживающими их почтовыми отделениями; и т. д.

Задача нахождения наибольшего паросочетания в двудольном графе G относится к числу старейших задач комбинаторных алгоритмов. (Следует заметить, что граф G имеет идеальное паросочетание в том и только в том случае, если $|X| = |Y|$, и в нем существует паросочетание размера $|X|$.) Как выясняется, эта задача решается алгоритмом с полиномиальным временем, но разработка такого алгоритма требует применения идей, принципиально отличающихся от приемов, рассматривавшихся нами ранее.

Вместо того чтобы браться за разработку алгоритма напрямую, мы сначала сформулируем общий класс задач — *задач нахождения потока в сети*, особым случаем которого является задача двудольного паросочетания. Затем мы разработаем алгоритм с полиномиальным временем для общей *задачи о максимальном потоке* и покажем, как тем самым можно получить эффективный алгоритм для двудольного паросочетания. Хотя изначально задачи потока в сети изучались для анализа трафика в сетях, вы увидите, что они находят широкое применение в самых разных областях и ведут к построению эффективных алгоритмов не только для задачи двудольных паросочетаний, но и для многих других задач.

7.1. Задача о максимальном потоке и алгоритм Форда–Фалкерсона

Графы часто используются для моделирования *транспортных сетей* — сетей, по ребрам которых передается некоторый трафик, а узлы выполняют функции «пересадочных узлов» для передачи трафика по разным ребрам. Представьте дорожную сеть, в которой ребра представляют дороги, а узлы — развязки; или компьютерную сеть, в которой ребра представляют каналы, способные передавать пакеты, а узлы — коммутаторы; или сеть трубопроводов, в которой ребра представляют трубы, а узлы — места соединения труб. Для сетевых моделей такого типа характерны некоторые компоненты: *пропускные способности ребер*, обозначающие максимальную величину передаваемого трафика; узлы-*источники*, генерирующие трафик; узлы-*стоки* (или приемники), «поглощающие» поступающий трафик; и наконец, сам трафик, передаваемый по ребрам.

Потоковые сети

При рассмотрении графов этого вида трафик называется *потоком* — это абстрактная сущность, которая генерируется узлами-источниками, передается по ребрам и поглощается узлами-стоками. Формально *сеть движения потока* (или *потоковая сеть*) представляет собой направленный граф $G = (V, E)$ со следующими отличительными признаками:

- С каждым ребром e связывается пропускная способность — неотрицательное число, которое будет обозначаться c_e.
- Существует один узел-источник $s \in V$.
- Существует один узел-сток $t \in V$.

Узлы, отличные от s и t, будут называться *внутренними* узлами.

В отношении потоковых сетей, с которыми мы будем иметь дело, будут действовать два предположения: во-первых, ни одно ребро не входит в источник s, и ни одно ребро не выходит из стока t; во-вторых, у каждого узла существует хотя бы одно инцидентное ему ребро; в-третьих, все пропускные способности представляют собой целые числа. Эти предположения делают модель более стройной и устраняют ряд аномалий, сохраняя практически все интересующие нас свойства.

На рис. 7.2 изображена потоковая сеть с четырьмя узлами и пятью ребрами; рядом с каждым ребром указана его пропускная способность.

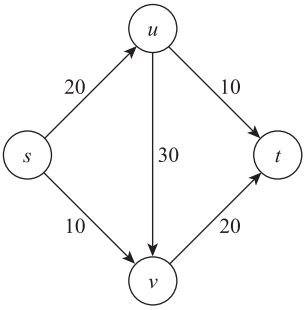


Рис. 7.2. Потоковая сеть с источником s и стоком t. Числа рядом с ребрами обозначают пропускные способности

Определение потока

На следующем шаге мы определим, что же понимается под передачей трафика (или потока) в нашей сети. Поток s–t представляет собой функцию f, которая связывает каждое ребро e с неотрицательным вещественным числом f: $E \rightarrow \mathbf{R}^+$; значение $f(e)$ представляет величину потока, передаваемого по ребру e. Поток f должен обладать следующими двумя свойствами[1]:

[1] Наше понятие потока моделирует трафик, проходящий в сети с постоянной скоростью: величина потока для ребра e обозначается одной переменной $f(e)$. Мы не будем моделировать *пульсирующий трафик*, при котором поток изменяется со временем.

(i) (Ограничения пропускной способности.) Для всех $e \in E$ выполняется условие $0 \le f(e) \le c_e$.

(ii) (Ограничения сохранения потока.) Для каждого узла v, отличного от s и t, выполняется условие

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e).$$

Здесь $\sum_{e \text{ into } v} f(e)$ — сумма значений потока по всем ребрам, входящим в узел v, а $\sum_{e \text{ out of } v} f(e)$ — сумма значений потока по всем ребрам, выходящим из v.

Таким образом, поток через ребро не может превышать его пропускную способность. Для каждого узла, отличного от источника и стока, величина входного потока должна быть равна величине выходного потока. У источника входящих ребер нет (согласно предположению), но поток из него выходит; иначе говоря, источник генерирует поток. Аналогичным образом сток может иметь входящий поток, хотя у него и нет выходящих ребер. *Величина* потока f, обозначаемая $v(f)$, определяется как сумма потоков, генерируемых в источнике:

$$v(f) = \sum_{e \text{ out of } s} f(e).$$

Чтобы сделать запись более компактной, мы определим $f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$ и $f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$. Определения можно расширить на множества вершин; если $S \subseteq V$, мы определим $f^{\text{out}}(S) = \sum_{e \text{ out of } S} f(e)$ и $f^{\text{in}}(S) = \sum_{e \text{ into } S} f(e)$. В этой терминологии ограничения сохранения потока для узлов $v \neq s,t$ принимают вид $f^{\text{in}}(v) = f^{\text{out}}(v)$; и мы можем записать $v(f) = f^{\text{out}}(s)$.

Задача о максимальном потоке

Естественная цель для заданной потоковой сети состоит в организации трафика, обеспечивающей как можно более эффективное использование доступной пропускной способности. Таким образом, основная алгоритмическая задача, которая будет рассматриваться в этой главе, формулируется так: для потоковой сети найти поток максимально возможной величины.

Размышляя над алгоритмом для этой задачи, необходимо учитывать, что структура потоковой сети устанавливает верхние границы для максимальной величины потока s–t. Основное «препятствие» для существования больших потоков заключается в следующем: предположим, узлы графа разделены на два множества A и B так, что $s \in A$ и $t \in B$. Интуитивно ясно, что любой поток, проходящий от s к t, должен в какой-то момент переходить из A в B — а следовательно, заполнять часть пропускной способности ребер из A в B. Из этого следует, что каждый такой «разрез» графа устанавливает ограничение на максимально возможную величину потока.

Алгоритм нахождения максимального потока, который мы разработаем в этом разделе, будет объединен с доказательством того, что величина максимального потока равна минимальной пропускной способности любого такого разбиения, называемого *минимальным разрезом*. Дополнительно наш граф также будет вычислять минимальный разрез. Вы увидите, что задача нахождения разрезов с минимальной пропускной способностью в потоковой сети полезна с точки зрения ее возможных практических применений — например, при нахождении максимального потока.

Разработка алгоритма

Итак, требуется найти максимальный поток в сети. Как взяться за решение этой задачи? Эксперименты показывают, что такие методы, как динамическое программирование, не работают — по крайней мере для задачи о нахождения максимального потока не существует алгоритма, который можно было бы естественно рассматривать как относящийся к парадигме динамического программирования. В отсутствии других идей можно поразмыслить над простыми жадными методами, чтобы понять, почему они не работают.

Допустим, мы начинаем с нулевого потока: $f(e) = 0$ для всех e. Очевидно, это состояние соблюдает ограничения пропускной способности и сохранения; проблема в том, что в нем величина потока равна 0. Попробуем увеличить значение f, «проталкивая» поток по пути из s в t, до границ, установленных пропускными способностями ребер. Таким образом, на рис. 7.3 можно было бы выбрать путь, состоящий из ребер $\{(s, u), (u, v), (v, t)\}$, увеличить поток по каждому из этих ребер до 20 и оставить $f(e) = 0$ для двух других. Это позволит выполнить ограничения пропускной способности (поток не превышает пропускной способности ребер) и ограничения сохранения потока (при увеличении потока для ребра, входящего во внутренний узел, мы также увеличиваем его для ребра, выходящего из узла). Теперь величина потока равна 20; возникает резонный вопрос: является ли она максимумом, возможным для изображенного графа? Если немного подумать, становится понятно, что в графе можно построить поток с величиной 30. Проблема в том, что мы «зашли в тупик» (не существует пути s–t, в который можно было бы напрямую протолкнуть поток без превышения пропускной способности), а максимальный поток еще не достигнут. Необходим более общий механизм проталкивания потока из s в t, который бы позволял увеличить величину текущего потока в подобных ситуациях.

Фактически нам хотелось бы выполнить операцию, обозначенную на рис. 7.3, *c* пунктирной линией. В ребро (s, v) проталкиваются 10 единиц потока; в результате в v входит слишком большой поток. Соответственно мы «отменяем» 10 единиц потока в (u, v); при этом восстанавливается ограничение сохранения потока для v, но зато из u выходит слишком малый поток. На последнем шаге 10 единиц потока проталкиваются в ребро (u, t), что приводит к восстановлению состояния сохранения для u. В результате в графе обнаруживается действительный поток величины 30. На рис. 7.3 темными ребрами обозначается поток перед операцией, а пунктирными — возможность его увеличения.

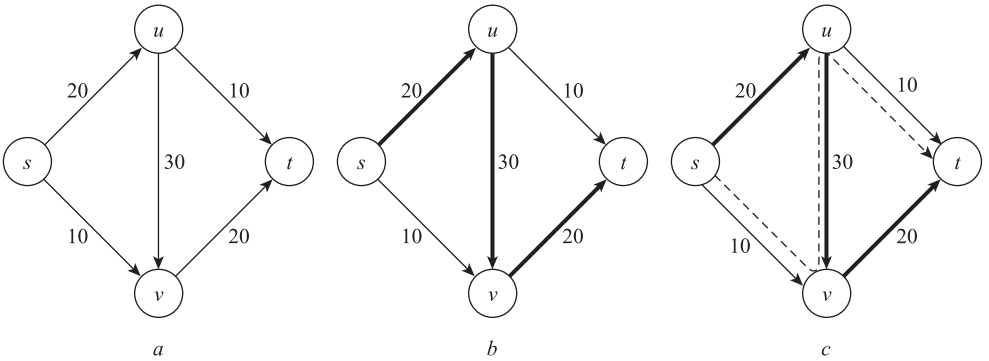


Рис. 7.3. (*a*) Сеть на рис. 7.2. (*b*) Проталкивание 20 единиц потока по пути s, u, v, t. (*c*) Новый увеличивающий путь использует ребро (u, v) в обратном направлении

В этом и заключается более общий механизм проталкивания потока: проталкивание ведется в прямом направлении для ребер со свободной пропускной способностью и в обратном направлении для ребер, по которым уже передается поток, для его перевода в другом направлении. Определим понятие *остаточного графа*, который предоставит систематический механизм поиска подобных операций.

Остаточный граф

Для заданной потоковой сети G и потока f в G остаточный граф G_f для G в отношении f определяется следующим образом (остаточный граф потока на рис. 7.3 после проталкивания 20 единиц потока по пути s, u, v, t изображен на рис. 7.4):

- Множество узлов G_f совпадает с множеством узлов G.
- Для каждого ребра $e = (u, v)$ графа G, для которого $f(e) < c_e$, имеются $c_e - f(e)$ «резервных» единиц пропускной способности, на которые можно попытаться нарастить поток.

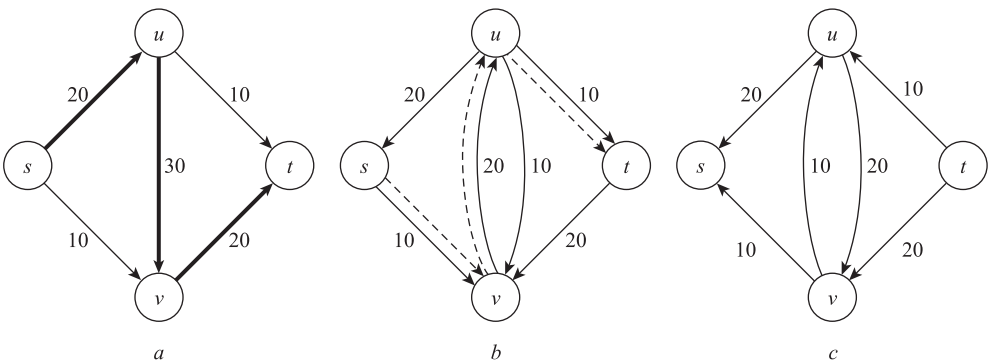


Рис. 7.4. (*a*) Граф G с путем s, u, v, t используется для проталкивания первых 20 единиц потока. (*b*) Остаточный граф полученного потока f; рядом с каждым ребром обозначена остаточная пропускная способность. Пунктирная линия обозначает новый увеличивающий путь. (*c*) Остаточный граф после проталкивания дополнительных 10 единиц потока по новому увеличивающему пути s, u, v, t

Соответственно в G_f включается ребро $e = (u, v)$ с пропускной способностью $c_e - f(e)$. Ребра, включаемые таким образом, называются *прямыми ребрами*.

- Для каждого ребра $e = (u, v)$ графа G, для которого $f(e) > 0$, имеются $f(e)$ единиц потока, которые можно при необходимости «отменить», направив поток в обратном направлении. Соответственно в G_f включается ребро $e' = (v, u)$ с пропускной способностью $f(e)$. Заметьте, что e' имеет те же концы, что и e, но проходит в обратном направлении; ребра, включаемые таким образом, называются *обратными ребрами*.

На этом определение остаточного графа G_f можно считать завершенным. Обратите внимание: каждое ребро e в G может породить одно или два ребра в G_f: если $0 < f(e) < c_e$, в G_f включается как прямое, так и обратное ребро. Следовательно, количество ребер в G_f может быть до двух раз больше, чем количество ребер в G. Пропускная способность ребра в остаточном графе иногда называется *остаточной пропускной способностью*, чтобы ее было проще отличить от пропускной способности соответствующего ребра в исходном потоковом графе G.

Увеличивающие пути в остаточном графе

На следующем шаге мы формализуем способ проталкивания потока из s в t для G_f. Пусть P является простым путем s–t в G_f — иначе говоря, P не посещает ни один узел более одного раза. Определим *критическую пропускную способность* `bottleneck`(P, f) как минимальную остаточную пропускную способность любого ребра P в отношении потока f. Затем определим операцию `augment`(f, P), которая дает новый поток f' в G.

```
augment(f, P)
   Присвоить b = bottleneck(P,f)
   Для каждого ребра (u, v) ∈ P
      Если e=(u, v) является прямым ребром
         Увеличить f(e) в G на b
      Иначе ((u, v) является обратным ребром, присвоить e=(v, u))
         Уменьшить f(e) в G на b
      Конец Если
   Конец цикла
   Вернуть (f)
```

Собственно, остаточный граф определялся исключительно для того, чтобы мы могли выполнить эту операцию. Любой путь s–t в остаточном графе часто называется *увеличивающим путем*.

Результат `augment`(f, P) представляет собой новый поток f' в G, полученный при увеличении и уменьшении величины потока для ребер P. Для начала убедимся в том, что f' действительно является потоком.

(7.1) f' является потоком в графе G.

Доказательство. Необходимо проверить ограничения пропускной способности и сохранения потока. Так как f' отличается от f только на ребрах P, необходимо проверить ограничения пропускной способности только на этих ребрах. Итак, пусть

(u, v) — ребро P. Неформально ограничение пропускной способности продолжает выполняться, потому что если $e = (u, v)$ является прямым ребром, мы сознательно предотвращаем увеличение потока на e выше c_e; а если (u, v) является обратным ребром, образовавшимся из ребра $e = (v, u) \in E$, мы сознательно предотвращаем снижение потока через e ниже 0. А если выражаться конкретнее, заметьте, что $bottleneck(P, f)$ не превышает остаточной емкости (u, v). Если $e = (u, v)$ — прямое ребро, то его остаточная пропускная способность равна $c_e - f(e)$; следовательно,

$$0 \le f(e) \le f'(e) = f(e) + \text{bootleneck}(P, f) \le f(e) + (c_e - f(e)) = c_e,$$

так что ограничение пропускной способности выполняется. Если (u, v) — обратное ребро, образовавшееся из ребра $e = (v, u) \in E$, то его пропускная способность равна $f(e)$, поэтому

$$c_e \ge f(e) \ge f'(e) = f(e) - \text{bootleneck}(P, f) \ge f(e) - f(e) = 0,$$

и снова ограничение пропускной способности выполняется.

Далее необходимо проверить ограничение сохранения потока для каждого внутреннего узла, лежащего на пути P. Пусть v является таким узлом; нетрудно убедиться в том, что изменение потока, входящего в v, равно изменению потока, выходящего из v; так как f удовлетворяет ограничению сохранения потока в v, для f' оно тоже должно выполняться.

Строго говоря, нужно проверить четыре случая в зависимости от того, является ли ребро P, входящее в v, прямым или обратным ребром, а также является ли ребро P, выходящее из v, прямым или обратным ребром. Однако все эти случаи анализируются достаточно легко, поэтому их проверку мы оставляем читателю. ■

В этой операции увеличения отражены типы прямого и обратного проталкивания потока, о котором говорилось ранее. Рассмотрим следующий алгоритм для вычисления потока s–t в G.

```
Max-Flow
   В исходном состоянии f(e) = 0 для всех e в G
   Пока существует путь s–t в остаточном графе G_f
      Пусть P – простой путь s-t в G_f
      f' = augment(f, P)
      Обновить f до f'
      Обновить остаточный граф G_f до G_f'
   Конец пока
Вернуть f
```

Этот алгоритм называется *алгоритмом Форда–Фалкерсона* в честь двух ученых, разработавших его в 1956 году. Пример выполнения алгоритма представлен на рис. 7.4. Алгоритм Форда–Фалкерсона в действительности достаточно прост; неочевидно в нем лишь то, завершится ли центральный цикл, и будет ли возвращаемый поток максимальным. Как выясняется, ответы на оба вопроса достаточно нетривиальны.

Анализ алгоритма: завершение и время выполнения

Начнем с доказательства некоторых свойств алгоритма; для этого мы воспользуемся методом индукции по количеству итераций цикла, предполагая, что все пропускные способности являются целыми числами.

(7.2) На каждой промежуточной стадии алгоритма Форда–Фалкерсона величины потока $\{f(e)\}$ и остаточные пропускные способности в G_f являются целыми числами.

Доказательство. Истинность этого утверждения до начала цикла очевидна. Теперь предположим, что оно истинно после j итераций. Поскольку все остаточные емкости в G_f являются целыми числами, значение bottleneck(P,f) для увеличивающего пути, найденного в итерации $j + 1$, будет целым числом. Из этого следует, что поток f' будет иметь целые значения — а значит, и пропускные способности нового остаточного графа будут целыми числами. ■

На основании этого свойства можно доказать, что алгоритм Форда–Фалкерсона завершается. Как и ранее в книге, мы будем искать метрику прогресса, из которой можно будет сделать вывод о завершении.

Сначала покажем, что величина потока строго возрастает при применении увеличения.

(7.3) Пусть f — поток в G, а P — простой путь s–t в G_f. Тогда $\nu(f') = \nu(f) +$ + bottleneck(P,f); а так как bottleneck$(P,f) > 0$, имеем $\nu(f') > \nu(f)$.

Доказательство. Первое ребро e в пути P должно выходить из s в остаточном графе G_f; поскольку путь является простым, он не возвращается к s. Так как в G нет ребер, входящих в s, ребро e должно быть прямым. Мы увеличиваем поток по этому ребру на bottleneck(P,f) и не изменяем поток через любое другое ребро, инцидентное s. Следовательно, значение f' превышает значение f на bottleneck(P,f). ■

Чтобы доказать завершение цикла, нам потребуется еще одно наблюдение: необходимо иметь возможность установить границу для максимально возможной величины потока. Граница может быть такой: если все ребра, выходящие из s, могут быть полностью насыщены потоком, то величина потока будет равна $\sum_{e \text{ out of } s} c_e$. Обозначим эту сумму C; можно утверждать, что $\nu(f) \leq C$ для всех потоков f для s–t. (C как оценка максимальной величины потока в G может быть сильно завышенной, но сейчас она для нас удобна как конечная и просто определяемая граница). Используя утверждение (7.3), мы теперь можем доказать завершение цикла.

(7.4) Предположим, как указано выше, что все пропускные способности в потоковой сети G являются целыми числами. Тогда алгоритм Форда–Фалкерсона завершается не более чем за C итераций цикла.

Доказательство. Ранее было замечено, что величина никакого потока в G не может превышать C (из-за ограничения пропускной способности для ребер, выходящих из i). Согласно (7.3) величина потока, которым управляет алгоритм Форда–Фалкерсона, возрастает при каждой итерации; следовательно, согласно (7.2),

при каждой итерации она увеличивается по крайней мере на 1. Так как значение начинается с 0 и не может превысить C, цикл в алгоритме Форда–Фалкерсона выполняется не более чем за C итераций. ■

Перейдем к рассмотрению времени выполнения алгоритма Форда–Фалкерсона. Пусть n — количество узлов в G, а m — количество ребер в G. Мы предполагали, что все узлы имеют хотя одно инцидентное ребро, поэтому $m \geq n/2$, и для упрощения границ можно использовать $O(m + n) = O(m)$.

(7.5) Предположим, все пропускные способности в потоковой сети G являются целыми числами. В этом случае алгоритм Форда–Фалкерсона может быть реализован с временем $O(mC)$.

Доказательство. Из (7.4) известно, что алгоритм завершается максимум за C итераций цикла. Рассмотрим объем работы, задействованной в одной итерации при текущем потоке f.

Остаточный граф G_f содержит не более $2m$ ребер, так как каждое ребро G порождает не более двух ребер в остаточном графе. Для хранения G_f будет использоваться представление списка смежности; для каждого узла v будут храниться два связанных списка: с ребрами, входящими в v, и с ребрами, выходящими из v. Чтобы найти путь s–t в G_f, можно воспользоваться поиском в ширину или в глубину, работающим за время $O(m + n)$; согласно предположению о том, что $m \geq n/2$, $O(m + n)$ — то же, что $O(m)$. Процедура `augment`(f, P) выполняется за время $O(n)$, так как путь P содержит максимум $n - 1$ ребро. Для известного нового потока f' новый остаточный граф можно построить за время $O(m)$: для каждого ребра e из G строятся правильные ребра (прямое и обратное) в $G_{f'}$.

Несколько более эффективная версия алгоритма будет хранить связанные списки ребер остаточного графа G_f в процедуре, изменяющей поток f посредством увеличения.

7.2. Максимальные потоки и минимальные разрезы

Перейдем к анализу алгоритма Форда–Фалкерсона, который займет целый раздел. Этот анализ даст много полезной информации не только об алгоритме, но и о задаче о максимальном потоке.

Анализ алгоритма: потоки и разрезы

Наша следующая цель — показать, что поток, возвращаемый алгоритмом Форда–Фалкерсона, имеет максимальную возможную величину для любого потока в G. Для этого мы вернемся к теме, поднятой в разделе 7.1: верхним границам для максимальной величины потока s–t, обусловленным структурой потоковой сети. Одна из таких границ уже приводилась: величина $v(f)$ любого потока s–t не превы-

шает $C = \sum_{e \text{ out of } s} c_e$. Иногда эта граница приносит пользу, но иногда оказывается очень слабой. Понятие *разреза* поможет нам разработать более общий механизм установления верхних границ для величины максимального потока.

Рассмотрим разбиение узлов графа на два множества A и B, для которых $s \in A$ и $t \in B$. Как упоминалось в разделе 7.1, любое такое разбиение устанавливает верхнюю границу для максимально возможного потока, потому что весь поток должен где-то переходить из A в B. Формально *разрезом s–t* называется разбиение (A, B) множества вершин V, при котором $s \in A$ и $t \in B$. *Пропускная способность* разреза (A, B), которую мы будем обозначать $c(A, B)$, представляет собой обычную сумму пропускных способностей всех ребер, выходящих из A: $c(A, B) = \sum_{e \text{ out of A}} c_e$.

Как выясняется, разрезы устанавливают верхнюю границу для величины потока — очень естественную и хорошо согласующуюся с нашими интуитивными представлениями. Сейчас эти рассуждения будут формализованы в серию фактов.

(7.6) Пусть f — произвольный поток s–t, а (A, B) — произвольный разрез s–t. В этом случае $v(f) = f^{\text{out}}(A) - f^{\text{in}}(A)$.

На самом деле это утверждение намного сильнее простой верхней границы. Оно говорит, что по величине потока f, передаваемой через разрез, можно точно измерить величину потока: это общая величина, выходящая из A, за вычетом величины, которая «втекает обратно» в A. Утверждение выглядит вполне естественно, хотя чтобы доказать его, придется немного потрудиться с суммами.

Доказательство. По определению $v(f) = f^{\text{out}}(s)$. Из предположения следует $f^{\text{in}}(s) = 0$, так как источник s не имеет входных ребер, и мы можем записать $v(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$. Кроме s, все узлы v в A являются внутренними, и мы знаем, что $f^{\text{out}}(v) - f^{\text{in}}(v) = 0$ для всех таких узлов. Следовательно,

$$v(f) = \sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v)),$$

так как у единственного ненулевого слагаемого в этой сумме v содержит s.

Попробуем переписать правую часть суммы. Если у ребра e оба конца принадлежат A, то $f(e)$ один раз входит в сумму со знаком «+», и один раз со знаком «–»; эти два слагаемых компенсируются. Если у e в A входит только начальный узел, то $f(e)$ входит в сумму только один раз со знаком «+». Если у e в A входит только конечный узел, то $f(e)$ входит в сумму только один раз со знаком «–». Наконец, если у e ни один из концов в A не входит, то $f(e)$ вообще не встречается в сумме. С учетом этого факта имеем

$$\sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v)) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A).$$

Объединяя эти два уравнения, приходим к формуле из (7.6). ■

Если $A = \{s\}$, то $f^{\text{out}}(A) = f^{\text{out}}(s)$, и $f^{\text{in}}(A) = 0$, так как по предположению никакие ребра не входят в источник. Следовательно, утверждение для этого множества $A = \{s\}$ в точности совпадает с определением величины потока $v(f)$.

Для разреза (A, B) ребра, входящие в B, в точности совпадают с ребрами, выходящими из A. Аналогичным образом ребра, выходящие из B, в точности совпадают с ребрами, входящими в A. Следовательно, $f^{out}(A) = f^{in}(B)$ и $f^{in}(A) = f^{out}(B)$ просто из сравнения определений этих двух выражений. Это позволяет переформулировать (7.6) следующим образом:

(7.7) Пусть f — произвольный поток s–t, а (A,B) — произвольный разрез s–t. В этом случае $v(f) = f^{in}(B) - f^{out}(B)$.

Если установить $A = V - \{t\}$ и $B = \{t\}$ в (7.7), получаем $v(f) = f^{in}(B) - f^{out}(B) = f^{in}(t) - f^{out}(t)$. По предположению сток не имеет исходящих ребер, поэтому $f^{out}(t) = 0$. Это означает, что *величину* потока также можно было бы определить в контексте стока t: это $f^{in}(t)$, величина потока, поступающего в сток.

Очень полезным следствием (7.6) является следующая верхняя граница.

(7.8) Пусть f — любой поток s–t, а (A, B) — любой разрез s–t. В этом случае $v(f) \le c(A, B)$.

Доказательство.

$$v(f) = f^{out}(A) - f^{in}(A) \le f^{out}(A) = \sum_{e \text{ out of } A} f(e) \le \sum_{e \text{ out of } A} c_e = c(A, B).$$

Первая строка — это просто (7.6); переходим от первой строки ко второй, так как $f^{in}(A) \ge 0$, а от третьей к четвертой — применяя ограничения пропускной способности к каждому из слагаемых суммы. ■

В некотором смысле утверждение (7.8) выглядит слабее, чем (7.6), так как в нем содержится неравенство вместо равенства. Тем не менее оно чрезвычайно полезно для нас, так как его правая часть не зависит ни от какого конкретного потока f. Фактически (7.8) говорит, что *величина любого потока ограничена сверху емкостью любого разреза*. Другими словами, рассматривая любой разрез s–t в G с некоторой величиной c^*, мы немедленно знаем из (7.8), что в G не может быть потока s–t с величиной, превышающей c^*. И наоборот, рассматривая любой поток s–t в G с величиной v^*, можно сразу утверждать по (7.8), что в s–t не может быть разреза с величиной менее v^*.

Анализ алгоритма: максимальный поток равен минимальному разрезу

Обозначим f поток, возвращаемый алгоритмом Форда–Фалкерсона. Требуется показать, что f имеет максимальную возможную величину среди всех потоков в G; для этого мы воспользуемся методом, упоминавшимся выше: предоставим разрез s–t (A^*, B^*), для которого $v(f) = c(A^*, B^*)$. Тем самым немедленно устанавливается, что f имеет максимальную величину среди всех потоков и что (A^*, B^*) имеет минимальную пропускную способность по всем разрезам s–t.

Алгоритм Форда–Фалкерсона завершается, когда поток f не имеет пути s–t в остаточном графе G_f. Как выясняется, это единственное свойство, необходимое для доказательства его максимальности.

(7.9) Если f — такой поток s–t, для которого не существует пути s–t в остаточном графе G_f, то в G существует разрез s–t (A^*, B^*), для которого $v(f) = c(A^*, B^*)$. Соответственно f имеет максимальную величину среди всех потоков в G, а (A^*, B^*) имеет минимальную емкость по всем разрезам s–t в G.

Доказательство. Это утверждение заявляет о существовании разреза, обладающего неким желательным свойством; теперь нужно найти такой разрез. Обозначим A^* множество всех узлов v в G, для которых в G_f существует путь s–v. Множество всех остальных узлов обозначается B^* : $B^* = V - A^*$.

Сначала установим, что (A^*, B^*) действительно является разрезом s–t. Безусловно, это разбиение V. Источник s принадлежит A^*, потому что путь из s в s всегда существует. Кроме того, $t \notin A^*$ по предположению об отсутствии пути s–t в остаточном графе; следовательно, $t \in B^*$, как и требуется.

Затем предположим, что $e = (u, v)$ является ребром в G, для которого $u \in A^*$ и $v \in B^*$, как показано на рис. 7.5. Можно утверждать, что $f(e) = c_e$. Если бы это было не так, то e было бы прямым ребром в остаточном графе G_f, а поскольку $u \in A^*$, в G_f существует путь s–u; присоединяя e к этому пути, мы получаем путь s–v в G_f, что противоречит предположению о $v \in B^*$.

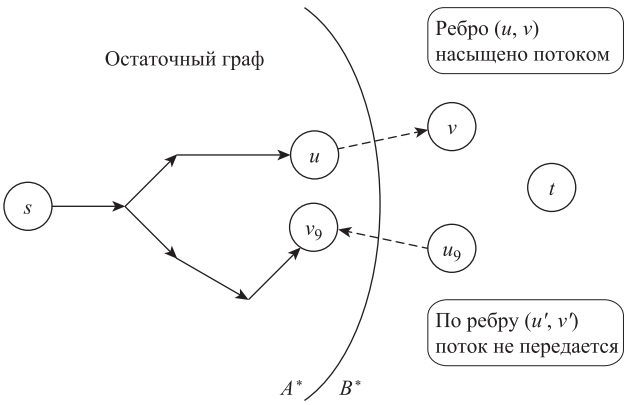


Рис. 7.5. Разрез (A^*, B^*) в доказательстве (7.9).

Теперь предположим, что $e' = (u', v')$ является ребром в G, для которого $u' \in B^*$, а $v' \in A^*$. Можно утверждать, что $f(e') = 0$. Если бы это было не так, то ребро e' породило бы обратное ребро $e'' = (v', u')$ в остаточном графе G_f, а поскольку $v' \in A^*$, в G_f существует путь s–v'; присоединяя e'' к этому пути, мы получаем путь s–u' в G_f, что противоречит предположению о $u' \in B^*$.

Итак, все ребра из A^* полностью насыщены потоком, а все ребра, направленные в A^*, совершенно не используются. Теперь мы можем воспользоваться (7.6), чтобы прийти к нужному выводу:

$$v(f) = f^{\text{out}}(A^*) - f^{\text{in}}(A^*) = \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) = \sum_{e \text{ out of } A^*} c_e - 0 = c(A^*, B^*).$$

Оглядываясь назад, мы видим, почему два типа остаточных ребер — прямые и обратные — критичны для анализа двух слагаемых в выражении из (7.6).

Так как алгоритм Форда–Фалкерсона завершается при отсутствии пути s–t в остаточном графе, из (7.6) немедленно следует его оптимальность.

(7.10) Поток f, возвращаемый алгоритмом Форда–Фалкерсона, является максимальным.

Также заметим, что наш алгоритм легко расширяется для вычисления минимального разреза s–t (A^*, B^*).

(7.11) Для заданного потока f с максимальной величиной разрез s–t с минимальной пропускной способностью вычисляется за время $O(m)$.

Доказательство. Просто последуем за построением доказательства (7.9). Мы строим остаточный граф G_f и проводим поиск в ширину или поиск в глубину для определения множества A^* всех узлов, доступных из s. После этого мы определяем $B^* = V - A^*$ и возвращаем разрез (A^*, B^*). ■

Обратите внимание: в графе G может быть много разрезов с минимальной пропускной способностью; процедура в доказательстве (7.11) просто находит один из этих разрезов, начиная с максимального потока f.

Дополнительно анализ алгоритма выявил следующий поразительный факт:

(7.12) В каждой потоковой сети существует поток f и разрез (A, B), для которых $v(f) = c(A, B)$.

Суть в том, что f в (7.12) должен быть максимальным потоком s–t; если бы существовал поток f' с большей величиной, то значение f' превысило бы пропускную способность (A, B), а это противоречит (7.8). Аналогичным образом можно сделать вывод о том, что (A, B) в (7.12) является минимальным разрезом (никакой другой разрез не может иметь меньшую пропускную способность), потому что если бы существовал разрез (A', B') с меньшей пропускной способностью, он был бы меньше величины f, а это снова противоречит (7.8). Из-за этих следствий утверждение (7.12) часто называется *теоремой о максимальном потоке и минимальном разрезе*, и формулируется следующим образом.

(7.13) В любой потоковой сети максимальная величина потока s–t равна минимальной пропускной способности разреза s–t.

Дальнейший анализ: целочисленные потоки

Среди многочисленных следствий нашего анализа алгоритма Форда–Фалкерсона есть одно особенно важное. Согласно (7.2) в любой момент времени величина потока остается целочисленной, а согласно (7.9) в итоге формируется максимальный поток. Следовательно:

(7.14) Если все пропускные способности потоковой сети являются целыми числами, то существует максимальный поток, для которого каждая величина потока $f(e)$ также является целым числом.

Обратите внимание: (7.14) не утверждает, что каждый максимальный поток является целочисленным, а лишь то, что некоторый максимальный поток обладает этим свойством. Интересно, что в (7.14) алгоритм Форда–Фалкерсона никак не упоминается, но наш алгоритмический подход предоставляет, пожалуй, самый простой способ доказательства этого утверждения.

Вещественные числа как пропускные способности

Наконец, прежде чем двигаться дальше, зададимся вопросом, насколько критичным было предположение о целочисленности пропускных способностей (речь не идет о (7.4), (7.5) и (7.14), где оно было очевидно необходимо). Для начала стоит заметить, что если разрешить использование рациональных чисел в качестве пропускных способностей, ситуация не станет более общей, потому что мы можем определить наименьшее общее кратное всех пропускных способностей и умножить их все на это значение, чтобы получить эквивалентную задачу с целочисленными пропускными способностями.

А если пропускные способности будут задаваться вещественными числами? Где в доказательстве мы использовали тот факт, что пропускные способности являются целыми числами? Да, использовали, и он был весьма критичен: утверждение (7.2) использовалось для доказательства того, что в (7.4) величина потока увеличивается по крайней мере на 1 при каждом шаге. С вещественными пропускными способностями следует учесть, что величина потока может возрастать со все более малыми приращениями; следовательно, пропадает гарантия конечности количества итераций цикла. И эта проблема оказывается очень серьезной, потому что *при аномальном выборе увеличивающего пути алгоритм Форда–Фалкерсона с вещественными пропускными способностями может выполняться бесконечно*.

Однако можно доказать, что теорема о максимальном потоке и минимальном разрезе (7.12) остается истинной даже в том случае, если пропускные способности являются вещественными числами. Утверждение (7.9) предполагало лишь то, что поток f не имеет пути s–t в остаточном графе G_f, чтобы сделать вывод о существовании разреза s–t с равной величиной. Очевидно, для любого потока f с максимальной величиной остаточный граф не содержит пути s–t; в противном случае величину потока можно было бы увеличить. Следовательно, чтобы доказать (7.12) для случая вещественных пропускных способностей, достаточно показать, что в каждой потоковой сети существует максимальный поток.

Конечно, при любом практическом применении потоков пропускные способности будут целыми или вещественными числами. Однако проблема аномального выбора увеличивающего пути может проявиться даже с целыми пропускными способностями: из-за нее алгоритм Форда–Фалкерсона может потребовать огромного количества итераций.

В следующем разделе показано, как выбрать увеличивающие пути так, чтобы избежать потенциально нежелательного поведения алгоритма.

7.3. Выбор хороших увеличивающих путей

В предыдущем разделе было показано, что любой выбор увеличивающего пути приводит к возрастанию величины потока; так мы пришли к границе C для количества увеличений, где $C = \sum_{e \text{ out of } s} c_e$. При относительно небольших C такая граница выглядит разумно; однако при больших C она становится слишком слабой.

Чтобы понять, насколько плохо может сложиться ситуация, рассмотрим граф на рис. 7.2, но со следующими пропускными способностями: ребра (s, v), (s, u), (v, t) и (u, t) имеют пропускную способность 100, а у ребра (u, v) пропускная способность равна 1 (рис. 7.6). Легко увидеть, что максимальный поток равен 200: $f(e) = 100$ для ребер (s, v), (s, u), (v, t) и (u, t), а для ребра (u, v) он равен 0. Этот поток может быть получен в результате серии из двух увеличений, использующих пути с узлами s, u, t и s, v, t. Но подумайте, насколько плохо поведет себя алгоритм Форда–Фалкерсона при аномальном выборе увеличивающих путей. Допустим, мы начинаем с увеличивающего пути P_1, содержащего узлы s, u, v, t в указанном порядке (рис. 7.6). Для этого пути $\texttt{bottleneck}(P_1, f) = 1$. После увеличения имеем $f(e) = 1$ для ребра $e = (u, v)$, поэтому в остаточном графе появляется обратное ребро. Для следующего увеличивающего пути выбирается путь P_2 путь P_2 из узлов s, v, u, t в таком порядке. При втором увеличении мы также получаем $\texttt{bottleneck}(P_2, f) = 1$. После второго увеличения имеем $f(e) = 0$ для ребра $e = (u, v)$, поэтому ребро снова находится в остаточном графе. Далее P_1 и P_2 поочередно выбираются для увеличения. В этом случае для того, чтобы добраться до желаемого потока с величиной 200, потребуется 200 увеличений. Именно эта граница была доказана в (7.4), так как в данном примере $C = 200$.

Разработка ускоренного алгоритма потока

В этом разделе мы покажем, что при более разумном выборе путей эту границу можно существенно улучшить. В области поиска хороших вариантов выбора увеличивающих путей в задаче о максимальном потоке, минимизирующих количество итераций, была проведена значительные исследования. Сейчас мы сосредоточимся на одном из самых естественных методов, а в конце раздела будут кратко упомянуты и другие методы. Вспомните, что увеличение наращивает величину максимального потока на критическую пропускную способность выбранного пути; следовательно, если выбирать пути с большой критической пропускной способностью, ситуация значительно улучшится. Естественно будет выбрать путь с наибольшей критической пропускной способностью. С другой стороны, необходимость поиска таких путей замедлит каждую отдельную итерацию. Чтобы избежать замедления, мы не будем беспокоиться о выборе пути, который бы имел *в точности* наибольшую критическую пропускную способность. Вместо этого мы введем *масштабирующий параметр* Δ и будем искать пути с критической пропускной способностью не менее Δ.

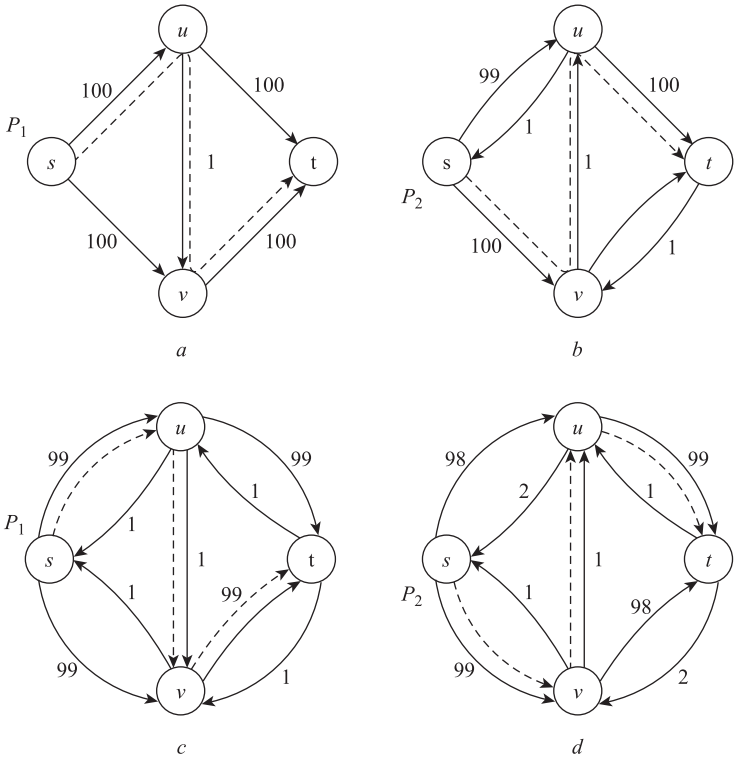


Рис. 7.6. Четыре итерации алгоритма Форда–Фалкерсона с неудачным выбором увеличивающих путей: увеличения чередуются между путем P_1 с узлами s, u, v, t (в указанном порядке) и путем P_2 с узлами s, u, v, t (в указанном порядке)

Пусть $G_f(\Delta)$ — подмножество остаточного графа, состоящее только из ребер с пропускной способностью не менее Δ. Мы будем работать со значениями Δ, которые являются степенью 2. Ниже приведен алгоритм.

```
Scaling Max-Flow
   В исходном состоянии f(e) = 0 для всех e в G
   Инициализировать Δ наибольшей степенью 2, не превышающей максимальную
         пропускную способность из s: Δ ≤ max_{e out of s} c_e
      Пока Δ ≥ 1
         Пока существует путь s–t в графе G_f(Δ)
            Присвоить P простой путь s–t в G_f(Δ)
            f' = augment(f, P)
            Обновить f до f' и обновить G_f(Δ)
         Конец Пока
         Δ = Δ/2
      Конец Пока
Вернуть f
```

Анализ алгоритма

Для начала заметим, что новый алгоритм максимального потока с масштабированием представляет собой простую реализацию исходного алгоритма Форда–Фалкерсона. Новые циклы, значение Δ и сокращенный остаточный граф $G_f(\Delta)$ нужны только для того, чтобы управлять выбором остаточного пути — чтобы ребра с большой остаточной пропускной способностью использовались как можно дольше. Следовательно, все свойства, доказанные для исходного алгоритма Форда–Фалкерсона истинны и для новой версии; поток остается целочисленным в этом алгоритме, а следовательно, все остаточные пропускные способности также являются целочисленными.

(7.15) Если пропускные способности являются целочисленными, то в алгоритме максимального потока с масштабированием поток и остаточные пропускные способности также остаются целочисленными. Из этого следует, что при $\Delta = 1$, $G_f(\Delta)$ не отличается от G_f — а следовательно, когда алгоритм завершает построение потока, f имеет максимальную величину.

Теперь рассмотрим время выполнения. Назовем итерацию внешнего цикла (с фиксированным значением Δ) *фазой* Δ-*масштабирования*. Легко дать верхнюю границу для количества разных фаз Δ-масштабирования в контексте значения $C = \sum_{e \text{ out of } s} c_e$, которое также использовалось в предыдущем разделе. Начальное значение Δ не превышает C, уменьшается вдвое и никогда не опускается ниже 1. Следовательно,

(7.16) Количество итераций внешнего цикла не превышает $1 + \lceil \log_2 C \rceil$.

Сложнее ограничить число увеличений, выполняемых в каждой фазе масштабирования. Идея заключается в том, что мы используем пути, значительно увеличивающие поток, поэтому увеличений должно быть относительно немного. Во время фазы Δ-масштабирования используются только ребра с остаточной пропускной способностью не менее Δ. Используя (7.3), получаем

(7.17) В фазе Δ-масштабирования каждое увеличение повышает величину потока минимум на Δ.

Здесь принципиально то, что в конце фазы Δ-масштабирования поток f не может слишком сильно отличаться от максимального возможного значения.

(7.18) Пусть f — поток в конце фазы Δ-масштабирования. В G существует разрез s–t (A, B), для которого $c(A, B) \leq \nu(f) + m\Delta$, где m — количество ребер в графе G. Соответственно, величина максимального потока в сети не превышает $\nu(f) + m\Delta$.

Доказательство. Это доказательство аналогично доказательству (7.9), которое устанавливало максимальную величину потока, возвращаемого исходным алгоритмом максимального потока.

Как и в том доказательстве, необходимо найти разрез (A, B) с нужным свойством. Обозначим A множество всех узлов v в G, для которых в $G_f(\Delta)$ существует путь s–v. Множество всех остальных узлов обозначается $B : B = V - A$. Мы видим, что (A, B) действительно является разрезом s–t, в противном случае фаза бы не закончилась.

Теперь рассмотрим ребро $e = (u, v)$ в G, для которого $u \in A$ и $v \in B$. Утверждается, что $c_e < f(e) + \Delta$. В противном случае e было бы прямым ребром в графе $G_f(\Delta)$, а так как $u \in A$, существует путь s–u в $Gf(\Delta)$; присоединяя e к этому пути, мы получаем путь s–v в $G_f(\Delta)$, что противоречит предположению о $v \in B$. Аналогично можно утверждать, что для любого ребра $e' = (u', v')$ в G, для которого $u' \in B$ и $v' \in A$, выполняется $f(e) < \Delta$. Если бы $f(e') \geq \Delta$, то ребро e' породило бы обратное ребро $e'' = (v', u')$ в графе $G_f(\Delta)$, а поскольку $v' \in A$, в $G_f(\Delta)$ существует путь s–v'; присоединяя e'' к этому пути, мы получаем путь s–u' в $G_f(\Delta)$, что противоречит предположению о $u' \in B$.

Итак, все ребра e из A почти насыщены — они удовлетворяют условию $c_e < f(e) + \Delta$ — все ребра, направленные в A, почти пусты — они удовлетворяют условию $f(e) < \Delta$. Теперь мы можем воспользоваться (7.6), чтобы придти к нужному выводу:

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \geq \sum_{e \text{ out of } A} (c_e - \Delta) - \sum_{e \text{ into } A} \Delta \\ &= \sum_{e \text{ out of } A} c_e - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ into } A} \Delta \geq c(A, B) - m\Delta. \end{aligned}$$

Первое неравенство следует из ограничений на величину потока в ребрах, пересекающих разрез, а второе — из того простого факта, что граф содержит только m ребер.

Величина максимального потока ограничивается пропускной способностью любого разреза по (7.8). Мы используем разрез (A, B) для получения границы, заявленной во втором утверждении. ■

(7.19) Количество увеличений в фазе масштабирования не превышает $2m$.

Доказательство. Истинность утверждения в первой фазе масштабирования очевидна; в этой фазе каждое из ребер, выходящих из s, может использоваться не более чем для одного увеличения. Теперь рассмотрим более позднюю фазу масштабирования Δ; обозначим f_p поток в конце *предыдущей* фазы масштабирования. В этой фазе в качестве параметра использовалось значение $\Delta' = 2\Delta$. Согласно (7.18) максимальный поток f^* не превышает $v(f^*) \leq v(f_p) + m\Delta' = v(f_p) + 2m\Delta$. В фазе Δ-масштабирования каждое увеличение повышает поток минимум на Δ, а следовательно, увеличений не может быть более $2m$. ■

Одно увеличение выполняется за время $O(m)$, включая время, необходимое для подготовки графа и нахождения соответствующего пути. При этом используется максимум $1 + \lceil \log_2 C \rceil$ фаз масштабирования, а в каждой фазе выполняется не более $2m$ увеличений. Соответственно мы приходим к следующему результату.

(7.20) Масштабирующий алгоритм нахождения максимального потока в графе с m ребрами и целочисленными пропускными способностями находит максимальный поток не более чем за $2m(1 + \lceil \log_2 C \rceil)$ увеличений. Он может быть реализован для выполнения за максимальное время $O(m^2 \log_2 C)$.

При больших C эта граница намного лучше границы $O(mC)$, действующей для произвольной реализации алгоритма Форда–Фалкерсона. В примере, при-

веденном в начале раздела, пропускные способности были равны 100, но с таким же успехом они могут быть равны 2^{100}; в этом случае обобщенный алгоритм Форда–Фалкерсона может занять время, пропорциональное 2^{100}, тогда как масштабирующий алгоритм будет выполнен за время, пропорциональное $\log_2(2^{100}) = 100$.

Различие можно рассматривать так: обобщенный алгоритм Форда–Фалкерсона требует времени, пропорционального *значению* пропускных способностей, тогда как масштабирующему алгоритму достаточно времени, пропорциональному количеству *битов*, необходимых для представления пропускных способностей во входных данных. В результате масштабирующие алгоритмы выполняются за время, полиномиальное по размеру входных данных (то есть количеству ребер и числовому представлению пропускных способностей); тем самым достигается традиционная цель — обеспечение полиномиального времени выполнения алгоритма. Плохие реализации алгоритма Форда–Фалкерсона, требующие около C итераций, этого стандарта полиномиальности не достигают. (Вспомните, что в разделе 6.4 для описания таких алгоритмов, полиномиальных по величине входных значений, но не по количеству битов, необходимых для их представления, использовался термин «псевдополиномиальность»).

Расширения: сильные полиномиальные алгоритмы

Возможно ли добиться результата, который бы качественно превосходил результат масштабирующего алгоритма? Одна из возможных целей могла бы выглядеть так: граф из нашего примера (рис. 7.6) состоит из четырех узлов и пяти ребер; хотелось бы использовать количество итераций, полиномиальное по числам 4 и 5, и полностью независимое от значений пропускных способностей. Такой алгоритм, полиномиальный только по $|V|$ и $|E|$, и работающий с числами, имеющими полиномиальное количество битов, называется *сильно полиномиальным*. На самом деле существует простая и естественная реализация алгоритма Форда–Фалкерсона, которая ведет к такой сильно полиномиальной границе: каждая итерация выбирает увеличивающий путь с минимальным количеством ребер. Диниц, и независимо от него Эдмондс и Карп, доказали, что при таком выборе алгоритм завершается не более чем за $O(mn)$ итераций. Это были первые полиномиальные алгоритмы для задачи нахождения максимального потока. С тех пор был проделан значительный объем работы, направленной на улучшение времени выполнения алгоритмов нахождения максимального потока. В настоящее время известны алгоритмы с временем выполнения $O(mn \log n)$, $O(n^3)$ и $O(\min(n^{2/3}, m^{1/2})$ $m \log n \log U)$, причем последняя граница предполагает, что все пропускные способности задаются целыми числами, не превышающими U. В следующем разделе рассматривается сильно полиномиальный алгоритм нахождения максимального потока, основанный на другом принципе.

7.4.* Алгоритм проталкивания предпотока

С самого начала наше обсуждение задачи о максимальном потоке строилось вокруг идеи увеличивающего пути в остаточном графе. Однако существуют и другие эффективные методы нахождения максимального потока, не имеющие прямого отношения к увеличивающим путям. В этом разделе мы рассмотрим один из таких методов: *алгоритм проталкивания предпотока*.

Разработка алгоритма

Алгоритмы, основанные на увеличивающих путях, хранят состояние потока f и используют процедуру увеличения для наращивания величины потока. С другой стороны, *алгоритм проталкивания предпотока* по сути увеличивает поток на уровне отдельных ребер. Изменение потока для одного ребра обычно нарушает ограничение сохранения потока, поэтому алгоритм в ходе своей работы должен хранить нечто менее «правильное», чем поток — нечто, не соблюдающее ограничение сохранения.

Предпотоки

Предпотоком s–t (или сокращенно предпотоком) называется функция f, связывающая каждое ребро e с неотрицательным вещественным числом $f: E \to R^+$. Предпоток f должен удовлетворять ограничениям пропускной способности:

(i) Для всех $e \in E$ выполняется условие $0 \le f(e) \le c_e$.

Вместо жестких ограничений сохранения потока обязательны только неравенства: для каждого узла, кроме s, входной поток должен быть по крайней мере не меньше выходного.

(ii) Для всех узлов v, кроме источника s, выполняется условие

$$\sum_{e \text{ into } v} f(e) \ge \sum_{e \text{ out of } v} f(e).$$

Разность

$$e_f(v) \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e)$$

называется *избыточным потоком* предпотока в узле v. Обратите внимание: предпоток, в котором все узлы, кроме s и t, имеют нулевой избыточный поток, является потоком, и величина потока равна в точности $e_f(t) = -e_f(s)$. Для предпотока f можно определить концепцию остаточного графа G_f — точно так же, как это делалось для потока. Алгоритм «проталкивает» поток по ребрам остаточного графа (с использованием как прямых, так и обратных ребер).

Предпотоки и разметка

Алгоритм проталкивания предпотока работает с предпотоком и стремится преобразовать его в поток. Алгоритм основан на интуитивном представлении о том, что поток естественным образом стекает «сверху вниз». «Возвышенностями» для этого интуитивного представления являются метки $h(v)$ для всех узлов v, определяемых и поддерживаемых алгоритмом (рис. 7.7). Мы будем проталкивать поток от узлов с большими метками к узлам с меньшими метками (в соответствии с аналогией о жидкости, стекающей сверху вниз). В более точной формулировке *разметка* представляет собой функцию $h: V \to Z \geq 0$, отображающую узлы на неотрицательные целые числа. Метки также будут называться *высотами* узлов. Разметка h и предпоток s–t будут называться *совместимыми*, если

- (i) (ограничения источника и стока) $h(t) = 0$ и $h(s) = n$,
- (ii) (ограничения крутизны) для всех ребер $(v, w) \in E_f$ в остаточном графе $h(v) \leq h(w) + 1$.

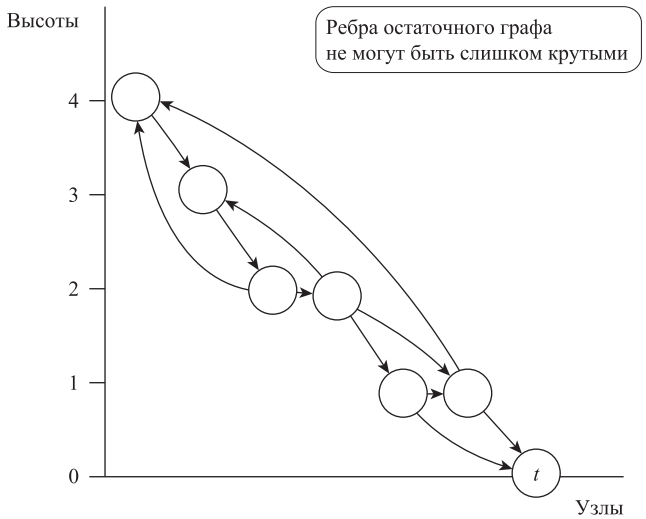


Рис. 7.7. Остаточный граф и совместимая разметка. Никакое ребро в остаточном графе не может быть слишком «крутым» — высота начального узла не может превышать высоту конечного узла более чем на 1. У источника s высота должна быть равна $h(s) = n$; на схеме она не показана

На уровне здравого смысла понятно: разность высот n между источником и стоком должна гарантировать, что начальная высота достаточна для перетекания потока от s к стоку t, а ограничение крутизны делает его спуск достаточно плавным, чтобы он дошел до стока.

Ключевое свойство совместимого предпотока и разметки заключается в том, что в остаточном графе не может существовать путь s–t.

(7.21) Если предпоток s–t совместим с разметкой h, то в остаточном графе G_f не существует путь s–t.

Доказательство. Утверждение доказывается от обратного. Пусть P — простой путь s–t в остаточном графе G. Обозначим узлы P: s, v_1, ..., $v_k = t$. По определению разметки, совместимой с предпотоком f, имеем $h(s) = n$. Ребро (s, v_1) входит в остаточный граф, а следовательно, $h(v_1) \geq h(s) - 1 = n - 1$. Используя индукцию по i и ограничение крутизны для ребра $(v_i - 1, v_i)$, получаем, что для всех узлов v_i в пути P высота равна минимум $h(v_i) \geq n - i$. Обратите внимание: последним узлом пути является $v_k = t$; следовательно, мы приходим к тому, что $h(t) \geq n - k$. Однако $h(t) = 0$ по определению, а $k < n$, так как путь P является простым. Это противоречие доказывает исходное утверждение. ■

Вспомните из (7.9), что если в остаточном графе G_f потока f не существует пути s–t, то поток имеет максимальную величину. Из этого вытекает следующее следствие.

(7.22) Если поток s–t совместим с разметкой h, то f является потоком с максимальной величиной.

Следует заметить, что (7.21) относится к предпотокам, а утверждение (7.22) более ограничено в том отношении, что оно применимо только к потокам. Таким образом, алгоритм проталкивания предпотока ведет предпоток f и разметку h, совместимую с f, и работает над модификацией f и h, чтобы превратить f в поток. Когда f действительно станет потоком, мы можем при помощи (7.22) сделать вывод о том, что это поток с максимальной величиной. В свете сказанного алгоритм проталкивания предпотока можно рассматривать как механизм, ортогональный алгоритму Форда–Фалкерсона. Алгоритм Форда–Фалкерсона поддерживает допустимый поток, постепенно изменяя его в направлении оптимальности. Алгоритм проталкивания предпотока, напротив, поддерживает условие, из которого следует оптимальность предпотока f, чтобы он был допустимым потоком, а алгоритм постепенно преобразует предпоток f в поток.

Для запуска алгоритма необходимо определить исходный предпоток f и совместимую с ним разметку h. Мы будем использовать исходную разметку $h(v) = 0$ для всех $v \neq s$, и $h(s) = n$. Чтобы предпоток f был совместимым с этой разметкой, следует убедиться в том, что ни одно ребро, выходящее из s, не присутствует в остаточном графе (так как эти ребра не удовлетворяют ограничению крутизны). Для этого мы определяем исходный предпоток $f(e) = c_e$ для всех ребер $e = (s, v)$, выходящих из источника, и $f(e) = 0$ для всех остальных ребер.

(7.23) Исходный предпоток f и разметка h совместимы.

Проталкивание и изменение разметки

Теперь посмотрим, какие действия выполняет алгоритм для преобразования предпотока f в допустимый поток, с сохранением его совместимости с некоторой разметкой h. Рассмотрим любой узел v с избыточным потоком (то есть $e_f(v) > 0$). Если в остаточном графе G_f имеется ребро e, которое выходит из v и переходит к узлу w на меньшей высоте (следует учитывать, что $h(v)$ превышает $h(w)$ не более чем на 1 из-за ограничения крутизны), то f можно изменить, перемещая часть избыточного потока из v в w.

push(f,h,v,w)
  Применяется, если $e_f(v) > 0$, $h(w) < h(v)$ и $(v,w) \in E_f$
  Если $e = (v,w)$ - прямое ребро
    Присвоить $\delta = \min(e_f(v), c_e - f(e))$ и
    увеличить $f(e)$ на δ
  Если (v,w) - обратное ребро
    Присвоить $e = (w,v)$, $\delta = \min(e_f(v), f(e))$ и
    уменьшить $f(e)$ на δ
  Вернуть (f,h)

Если протолкнуть избыточный поток v по любому ребру, выходящему из v, не удается, значит, необходимо поднять высоту v.

relabel(f,h,v)
  Применяется, если $e_f(v) > 0$, и
    для всех ребер $(v,w) \in E_f$ выполняется условие $h(w) \geq h(v)$
  Увеличить $h(v)$ на 1
  Вернуть (f,h)

Полный алгоритм проталкивания предпотока

Ниже приведена полная формулировка алгоритма проталкивания предпотока.

Preflow-Push
  В исходном состоянии $h(v) = 0$ для всех $v \neq s$, $h(s) = n$ и
  $f(e) = c_e$ для всех $e = (s,v)$ и $f(e) = 0$ для всех остальных ребер
  Пока существует узел $v \neq t$ с избыточным потоком $e_f(v) > 0$
    Пусть v - такой узел с избыточным потоком
    Если существует узел w, к которому можно применить *push*(f,h,v,w),
      push(f,h,v,w)
    Иначе
      relabel(f,h,v)
  Конец Пока
  Вернуть (f)

Анализ алгоритма

Как обычно, алгоритм нельзя назвать полностью определенным. Для его реализации нужно определить, какой узел с избыточным потоком следует выбрать и как эффективно выбрать ребро для проталкивания. Однако ясно, что каждая итерация этого алгоритма может быть реализована за полиномиальное время. (Позднее мы обсудим, как реализовать ее с достаточной эффективностью.) Более того, нетрудно увидеть, что предпоток f и разметка h остаются совместимыми на протяжении работы алгоритма. Если алгоритм завершается (что из его описания далеко не очевидно), то узлов с положительным избыточным потоком не остается (кроме t), а следовательно, предпоток f действительно является потоком. Затем из (7.22) следует, что f будет максимальным потоком при завершении.

Приведем несколько простых наблюдений относительно алгоритма.

(7.24) Во время выполнения алгоритма проталкивания предпотока:

(i) метки высот являются неотрицательными целыми числами;

(ii) если пропускные способности являются целыми числами, то предпоток f является целым числом;

(iii) предпоток f и разметка h совместимы.

Если алгоритм возвращает предпоток f, то f является потоком с максимальной величиной.

Доказательство. Согласно (7.23), исходный предпоток f и разметка h совместимы. Докажем посредством индукции по количеству операций `push` и `relabel`, что f и h обладают свойствами из утверждения. Операция проталкивания изменяет предпоток f, но границы δ гарантируют, что возвращаемый поток f удовлетворяет ограничениям пропускной способности и что все избыточные потоки остаются неотрицательными, так что f остается предпотоком. Чтобы убедиться в совместимости предпотока f и разметки h, заметим, что операция `push`(f, h, v, w) может добавить одно ребро в остаточный граф — обратное ребро (v, w), и это ребро удовлетворяет ограничению крутизны. Операция `relabel` увеличивает метку v, а следовательно, повышает крутизну всех ребер, выходящих из v. Однако она применяется только в том случае, если никакое ребро, выходящее из v в остаточном графе, не направлено вниз — а значит, предпоток f и разметка h совместимы после изменения метки.

Алгоритм завершается, когда ни один узел, кроме s или t, не имеет избыточного потока. В этом случае f является потоком по определению; а поскольку предпоток f и разметка h остаются совместимыми на протяжении работы алгоритма, из (7.22) следует, что f является потоком с максимальной величиной. ■

Теперь рассмотрим количество операций `push` и `relabel`. Сначала докажем ограничение на количество операций `relabel` — это поможет доказать предел максимального числа возможных операций проталкивания. Алгоритм никогда не изменяет метку s (так как у источника не может быть положительного избыточного потока). У всех остальных узлов v изначально $h(v) = 0$, а метка возрастает на 1 при каждом изменении. Следовательно, нужно просто установить предел для максимально возможного значения метки. Узел v становится кандидатом для `relabel` только в том случае, если у v есть избыточный поток. Единственный источником потока в сети является s; интуитивно ясно, что избыточный поток в v должен происходить из s. Следующее следствие из этого факта играет ключевую роль в ограничении меток.

(7.25) Имеется предпоток f. Если в узле v имеется избыточный поток, то существует путь в G_f из v к источнику s.

Доказательство. Пусть A обозначает множество всех узлов w, для которых существует путь из w в s в остаточном графе G_f, а $B = V - A$. Требуется показать, что все узлы с избыточным потоком принадлежат A.

Отметим, что $s \in A$. Кроме того, никакие ребра $e = (x, y)$, выходящие из A, не могут иметь положительный поток, так как ребро с $f(e) > 0$ породило бы обратное ребро (y, x) в остаточном графе, и тогда узел y находился бы в A. Теперь рассмотрим

сумму избыточных потоков в множестве B и вспомним, что каждый узел в B имеет неотрицательный избыточный поток, так как $s \notin B$.

$$0 \le \sum_{v \in B} e_f(v) = \sum_{v \in B} (f^{in}(v) - f^{out}(v))$$

Изменим запись суммы в правой части. Если оба конца ребра e принадлежат B, то $f(e)$ один раз включается в сумму со знаком «+» и один раз со знаком «–»; два слагаемых компенсируются. Если у e только конечный узел принадлежит B, то e выходит из A, а как было показано выше, у всех ребер, выходящих из A, $f(e) = 0$. Если у e только начальный узел принадлежит B, то $f(e)$ включается в сумму только один раз со знаком «–». Таким образом, получаем

$$0 \le \sum_{v \in B} e_f(v) = -f^{out}(B).$$

Так как потоки неотрицательны, мы видим, что сумма избыточных потоков в B равна нулю; так как каждый отдельный избыточный поток в B неотрицателен, это означает, что все они должны быть равны 0. ■

Теперь можно доказать, что в процессе выполнения алгоритма метки не изменяются слишком значительно. Вспомните, что n обозначает число узлов в V.

(7.26) Во время выполнения алгоритма для всех узлов $h(v) \le 2n - 1$.

Доказательство. Исходные метки $h(t) = 0$ и $h(s) = n$ не изменяются во время выполнения алгоритма. Рассмотрим произвольный узел $v \ne s, t$. Алгоритм изменяет метку v только при применении операции `relabel`; пусть f и h — предпоток и разметка, возвращенные операцией `relabel`(f, h, v). Согласно (7.25) существует путь P в остаточном графе G_f из v в s. Обозначим $|P|$ количество ребер в P; заметим, что $|P| \le n - 1$. Ограничение крутизны подразумевает, что высоты узлов могут убывать не более чем на 1 с каждым ребром P; следовательно, $h(v) - h(s) \le |P|$, что доказывает утверждение. ■

Метки монотонно возрастают во время выполнения алгоритма, поэтому это утверждение немедленно устанавливает предел для количества операций изменения меток.

(7.27) В процессе выполнения алгоритма метка каждого узла изменяется не более $2n - 1$ раз, поэтому общее количество операций `relabel` меньше $2n^2$.

Затем будет установлена граница для количества операций проталкивания. Будем различать два вида таких операций: операция `push`(f, h, v, w) называется *насыщающей*, если либо $e = (v, w)$ является прямым ребром в E_f и $\delta = c_e - f(e)$, либо (v, w) является обратным ребром с $e = (w, v)$ и $\delta = f(e)$. Иначе говоря, проталкивание является насыщающим, если после него ребро (v, w) уходит из остаточного графа. Все остальные операции проталкивания будут называться *ненасыщающими*.

(7.28) В процессе выполнения алгоритма количество насыщающих операций *push* не превышает $2nm$.

Доказательство. Рассмотрим ребро (v, w) в остаточном графе. После насыщающей операции проталкивания `push`(f, h, v, w) имеем $h(v) = h(w) + 1$, а ребро (v,w) по-

кидает остаточный граф G_f, как показано на рис. 7.8. Прежде чем мы снова сможем выполнять проталкивание по этому ребру, сначала необходимо протолкнуть из w в v, чтобы ребро (v, w) появилось в остаточном графе. Но для проталкивания из w в v сначала метка w должна увеличиться минимум на 2 (чтобы узел w находился выше v). Метка w может увеличиться на 2 не более $n - 1$ раза, так что насыщающее проталкивание из v в w может происходить не более n раз. Каждое ребро $e \in E$ может породить до двух ребер в остаточном графе, поэтому общее количество насыщающих проталкиваний не превышает $2nm$. ■

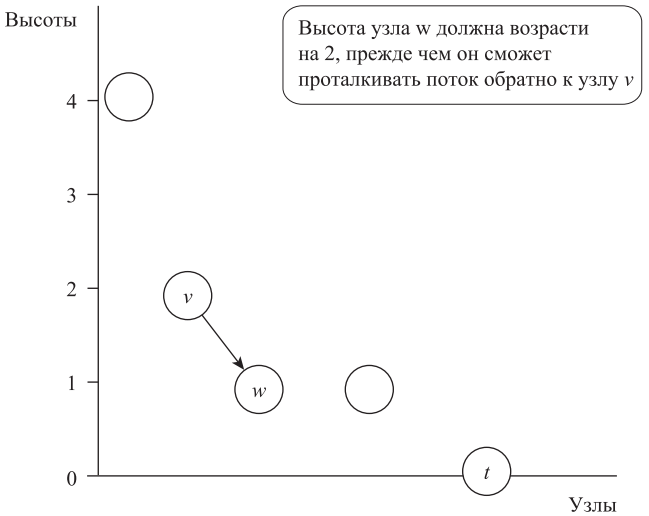


Рис. 7.8. После насыщающего проталкивания push(f, h, v, w) высота v превышает высоту w на 1

Самая сложная часть анализа — доказательство границы для количества ненасыщающих проталкиваний. Она же становится критическим фактором в теоретической границе времени выполнения.

(7.29) В процессе выполнения алгоритма количество ненасыщающих операций push не превышает $2n^2m$.

Доказательство. В этом доказательстве мы воспользуемся так называемым *методом потенциальных функций*. Для предпотока f и совместимой разметки h мы определяем

$$\Phi(f,h) = \sum_{v \neq s,\, e_f(v)>0} h(v)$$

как сумму высот всех узлов с положительным избыточным потоком. (Функция Φ часто называется *потенциальной*, потому что она напоминает «потенциальную энергию» всех узлов с положительным избыточным потоком).

В исходном предпотоке и разметке все узлы с положительным избыточным потоком имеют высоту 0, поэтому $\Phi(f,h) = 0$ остается неотрицательной во время выполнения алгоритма. Ненасыщающая операция push(f, h, v, w) уменьшает $\Phi(f,h)$

как минимум на 1, потому что после проталкивания узел v не имеет избыточного потока, а w — единственный узел, получающий новый избыточный поток от операции — имеет высоту на 1 меньше, чем v. Однако каждая насыщающая операция проталкивания и каждая операция `relabel` может увеличить $\Phi(f,h)$. Операция `relabel` увеличивает $\Phi(f,h)$ ровно на 1. Выполняется не более $2n^2$ операций `relabel`, поэтому общее возрастание $\Phi(f,h)$, обусловленное операциями `relabel`, составит $2n^2$. Насыщающая операция `push`(f, h, v, w) не изменяет метки, но она может увеличить $\Phi(f,h)$, потому что узел w может неожиданно получить положительный избыточный поток после проталкивания. Это приведет к увеличению $\Phi(f,h)$ на высоту w, которая не превышает $2n - 1$. Существуют максимум $2nm$ насыщающих операций проталкивания, поэтому общее возрастание $\Phi(f,h)$ из-за операций проталкивания не превышает $2mn(2n - 1)$. Следовательно, под действием двух факторов $\Phi(f,h)$ может возрасти максимум на $4mn^2$ во время выполнения алгоритма.

Но так как Φ остается неотрицательным и убывает минимум на 1 при каждой ненасыщающей операции проталкивания, из этого следует, что количество ненасыщающих операций проталкивания не может превышать $4mn^2$. ■

Расширения: улучшенная версия алгоритма

В области правил выбора узлов для улучшения времени выполнения алгоритма проталкивания предпотока в худшем случае были проведены значительные исследования. Сейчас мы рассмотрим простое правило, которое приводит к улучшенной границе $O(n^3)$ для количества ненасыщающих операций проталкивания.

(7.30) Если на каждом шаге выбирается узел с избыточным потоком на максимальной высоте, количество ненасыщающих операций проталкивания в процессе выполнения алгоритма не превышает $4n^3$.

Доказательство. Рассмотрим максимальную высоту $H = \max_{v \in V, e_f(v)>0} h(v)$ любого узла с избыточным потоком во время выполнения алгоритма. В анализе будет использоваться максимальная высота H вместо потенциальной функции Φ из предыдущей границы $O(n^2m)$.

Максимальная высота H может увеличиться только из-за изменения меток (так как поток всегда проталкивается к узлам с меньшей высотой), поэтому общее возрастание H в алгоритме не может превышать $2n^2$ согласно (7.26). Значение H начинается с 0 и остается неотрицательным, поэтому количество изменений H не превышает $4n^2$.

Теперь рассмотрим поведение алгоритма в фазе времени, в которой H остается постоянной величиной. Утверждается, что каждый узел может иметь максимум одну ненасыщающую операцию `push` в этой фазе. Действительно, в этой фазе поток проталкивается от узлов на высоте H к узлам на высоте $H - 1$; и после ненасыщающей операции *push* от v он должен получить поток от узла на высоте $H + 1$, прежде чем от него снова можно будет проталкивать поток.

Так как количество ненасыщающих операций push между изменениями H не превышает n, а H изменяется не более $4n^2$ раз, общее количество ненасыщающих операций push не превышает $4n^3$. ■

В продолжение (7.30) интересно заметить, что по проведенным экспериментам вычислительным «узким местом» метода является количество операций relabel, и более эффективное время выполнения было получено для модификаций, которые увеличивают метки быстрее, чем по единице. Это обстоятельство будет рассмотрено подробнее в упражнениях.

Реализация алгоритма проталкивания предпотока

Наконец, необходимо кратко обсудить эффективную реализацию этого алгоритма. Несколько простых структур данных позволят эффективно реализовать каждую из операций алгоритма за постоянное время, так что общая реализация алгоритма займет время $O(mn)$ плюс количество ненасыщающих операций push. Следовательно, обобщенный алгоритм будет выполняться за время $O(mn^2)$, тогда как версия, которая всегда выбирает узел с максимальной высотой, будет выполняться за время $O(n^3)$.

Все узлы с избыточным потоком можно объединить в простой список, чтобы выбор узла с избыточным потоком происходил с постоянным временем. Что касается выбора узла с максимальной высотой H за постоянное время, придется действовать более осмотрительно. Для этого мы будем вести связанный список всех узлов с избыточным потоком на каждой возможной высоте. Обратите внимание: если узел v изменяет метку или сохраняет положительный избыточный поток после проталкивания, он остается узлом с максимальной высотой H. Следовательно, выбирать новый узел после проталкивания нужно только в том случае, когда текущий узел v уже не имеет положительного избыточного потока. Если узел v находился на высоте H, то новый узел на максимальной высоте также будет находиться на высоте H, или если ни один узел на высоте H не имеет избыточного потока, максимальная высота будет равна $H - 1$, поскольку предыдущая операция push из v протолкнула поток в узел на высоте $H - 1$.

Теперь предположим, что был выбран узел v и теперь нужно выбрать ребро (v, w) для применения push(f, h, v, w) (или relabel(f, h, v), если такого w не существует). Чтобы иметь возможность быстро выбрать ребро, мы будем использовать представление графа в виде списка смежности. А точнее, для каждого узла v будет вестись связанный список всех возможных ребер, выходящих из v в остаточном графе (как прямых, так и обратных), и с каждым ребром будет храниться его пропускная способность и величина потока. Следует заметить, что в этом случае в структуре данных будут храниться две копии каждого ребра: прямая и обратная. Эти две копии содержат указатели друг на друга, поэтому обновления, выполняемые с одной копией, будут переноситься на другую копию за время $O(1)$. Ребра, выходящие из узла v, будут выбираться для операций push в порядке их следования в списке узла v. Чтобы упростить этот выбор, мы будем поддерживать для каждого

узла указатель current(v) на последнее ребро в списке, которое рассматривалось для операции push. Следовательно, если ребро v не имеет избыточного потока после насыщающей операции push из узла v, указатель current(v) останется на этом ребре, и это же ребро будет использоваться для следующей операции push из v. После насыщающей операции push из узла v указатель current(v) перемещается к следующему ребру в списке.

Здесь принципиально то, что после перемещения указателя current(v) от ребра (v, w) операция push не должна повторно применяться к этому ребру, пока не будет изменена метка v.

(7.31) После того, как указатель current(v) переместится от ребра (v, w), применение push к этому ребру невозможно, пока не будет изменена метка v.

Доказательство. После перемещения current(v) от ребра (v, w) вполне логично, что операции push не могут применяться к этому ребру. Либо $h(w) \geq h(v)$, либо ребро не входит в остаточный граф. В первом случае очевидно, что перед применением push к этому ребру необходимо изменить метку v. Во втором случае необходимо применить push к обратному ребру (w, v), чтобы ребро (v, w) снова вошло в остаточный граф. Однако если применить push к ребру (w, v), то w окажется выше v, а следовательно, метку v необходимо изменить, прежде чем поток можно будет снова проталкивать от v к w. ■

Так как перед изменением метки ребра не нужно снова рассматривать для применения push, приходим к следующему результату.

(7.32) Когда указатель current(v) достигает конца списка ребер для v, к узлу v можно применить операцию relabel.

После изменения метки узла v указатель current(v) возвращается к первому ребру в списке, после чего ребра снова рассматриваются в порядке их следования в списке v.

(7.33) Время выполнения алгоритма проталкивания предпотока, реализованного с использованием описанных структур данных, равно $O(mn)$ плюс $O(1)$ для каждой ненасыщающей операции push. В частности, обобщенный алгоритм проталкивания предпотока выполняется за время $O(n^2m)$, а версия, в которой всегда выбирается узел с максимальной высотой, выполняется за время $O(n^3)$.

Доказательство. Исходный поток и процедура изменения меток настраиваются за время $O(m)$. Операции push и relabel (после того, как операция будет выбрана) реализуются за время $O(1)$. Рассмотрим узел v. Мы знаем, что метка v может быть изменена не более $2n$ раз на протяжении алгоритма. Рассмотрим общее время, потраченное алгоритмом на поиск правильного ребра для проталкивания потока из узла v между двумя изменениями метки v. Если узел v имеет d_v смежных ребер, то согласно (7.32) на перемещение указателя current(v) между последовательными изменениями метки v будет потрачено время $O(d_v)$. Следовательно, общее время, потраченное на продвижение указателей current, по алгоритму составляет $O(\sum_{v \in V} nd_v) = O(mn)$, как и утверждалось. ■

7.5. Первое применение: задача о двудольном паросочетании

Разработав несколько эффективных алгоритмов для задачи нахождения максимального потока, мы обратимся к практическим применениям максимальных потоков и минимальных разрезов в графах. Начнем с двух фундаментальных задач. В этом разделе рассматривается задача о двудольном паросочетании, упоминавшаяся в начале главы. В следующем разделе мы обратимся к более общей задаче непересекающихся путей.

Задача

Одной из первоначальных целей для разработки алгоритма нахождения максимального потока было решение задачи о двудольном паросочетании; сейчас вы увидите, как это делается. Напомним, что *двудольным графом* $G = (V, E)$ называется ненаправленный граф, множество узлов которого можно разбить на подмножества $V = X \cup Y$, обладающее тем свойством, что один конец каждого ребра $e \in E$ принадлежит X, а другой конец принадлежит Y. *Паросочетанием* M в G называется такое подмножество ребер $M \subseteq E$, что каждый узел не более чем в одно ребро из M.

Задача о двудольном паросочетании заключается в нахождении в G паросочетания наибольшего возможного размера.

Разработка алгоритма

Граф, определяющий задачу паросочетаний, не направлен, тогда как потоковые сети являются направленными; тем не менее алгоритм задачи нахождения максимального потока легко адаптируется для нахождения максимального паросочетания.

Начиная с графа G в экземпляре задачи о двудольном паросочетании, мы построим потоковую сеть G' так, как показано на рис. 7.9. Сначала все ребра из G будут направлены из X в Y. Затем мы добавим узел s и ребро (s, x) из s в каждый узел X. Далее добавляется узел t и ребро (y, t) из каждого узла из Y в t. Наконец, каждому ребру в G' назначается пропускная способность 1.

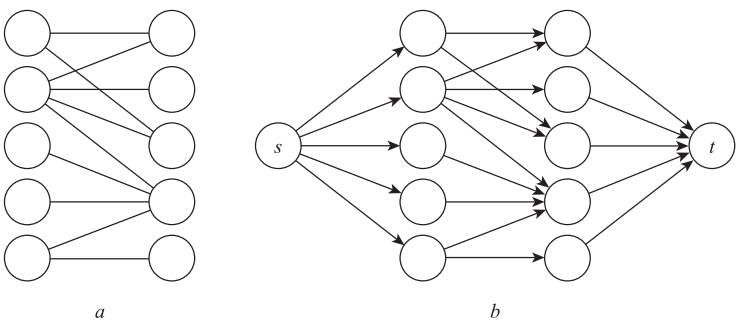


Рис. 7.9. *a* — двудольный граф; *b* — соответствующая потоковая сеть, в которой все пропускные способности ребер равны 1

После этого вычисляется максимальный поток s–t в этой сети G'. Как выясняется, величина этого максимального потока равна размеру максимального паросочетания в G. Кроме того, из анализа становится видно, как использовать сам поток для получения паросочетания.

Анализ алгоритма

В основе анализа лежит идея о том, что целочисленные потоки G' достаточно прозрачно кодируют паросочетания из G. Сначала предположим, что в G существует паросочетание, состоящее из k ребер $(x_{i_1}, y_{i_1}), \ldots, (x_{i_k}, y_{i_k})$. Затем рассмотрим поток f, отправляющий одну единицу потока по каждому пути в форме x_{i_j}, y_{i_j} — то есть $f(e) = 1$ для каждого ребра на одном из этих путей. Легко убедиться в том, что ограничения пропускной способности и сохранения потока действительно выполняются, а f является потоком s–t с величиной k.

И наоборот, предположим, что в G' существует поток f' с величиной k. Согласно теореме целочисленности максимальных потоков известно, что существует целочисленный поток f с величиной k; а поскольку все пропускные способности равны 1, это означает, что для каждого ребра e значение $f(e)$ равно 0 или 1. Теперь рассмотрим множество M' всех ребер в форме (x, y), для которых величина потока равна 1.

Множество M' обладает тремя простыми свойствами.

(7.34) M' состоит из k ребер.

Доказательство. Чтобы доказать это утверждение, рассмотрим разрез (A, B) в G' с $A = \{s\} \cup X$. Величина потока вычисляется как общий поток, выходящий из A, за вычетом общего потока, входящего в A. Первое из этих слагаемых попросту равно мощности M', так как ребра, выходящие из A, несут поток, причем каждое ребро несет ровно одну единицу потока. Второе слагаемое равно 0, так как в A нет входящих ребер. Из этого следует, что M' содержит k ребер. ■

(7.35) Каждый узел в X является начальным не более чем для одного ребра в M'.

Доказательство. Чтобы доказать это утверждение, предположим, что $x \in X$ является начальным узлом для минимум двух ребер в M'. Так как поток является целочисленным, это означает, что из x выходят как минимум две единицы потока. По ограничению сохранения потока в x должны входить минимум две единицы потока — но это невозможно, поскольку в x входит только одно ребро с пропускной способностью 1. А следовательно, x является начальным узлом не более чем для одного ребра в M'.

Рассуждая аналогичным образом, можно показать, что

(7.36) Каждый узел в Y является конечным не более чем для одного ребра в M'.

Объединяя все эти свойства, мы видим, что при рассмотрении M' как множества ребер в исходном двудольном графе G будет получено паросочетание с размером k. В итоге нам удалось доказать следующий факт.

(7.37) Размер максимального паросочетания в G равен величине максимального потока в G'; а ребрами такого паросочетания в G являются ребра, передающие поток от X к Y в G'.

Обратите внимание на важную роль, отведенную теореме целочисленности (7.14) в этом построении: мы должны были знать, существует ли в G' максимальный поток, состоящий только из величин 0 и 1.

Граница времени выполнения

Теперь посмотрим, насколько быстро можно вычислить максимальное паросочетание в G. Пусть $n = |X| = |Y|$, а m — количество ребер в G. Будем предполагать, что существует хотя бы одно ребро, инцидентное каждому узлу из исходной задачи — а следовательно, $m \geq n/2$. Во времени вычисления максимального паросочетания доминирующим фактором является время вычисления целочисленного максимального потока в G', так как преобразование его в паросочетание в G выполняется просто. Для этой задачи потока имеем $C = \sum_{e \text{ out of } s} c_e = |X| = n$, так как s содержит ребро с пропускной способностью 1 для каждого узла X. Используя границу $O(mC)$ из (7.5), получаем:

(7.38) Алгоритм Форда–Фалкерсона может использоваться для нахождения максимального паросочетания в двудольном графе за время $O(mn)$.

Интересно, что при использовании «улучшенных» границ $O(m^2 \log_2 C)$ или $O(n^3)$, разработанных в предыдущих разделах, мы бы получили для этой задачи ухудшенное время $O(m^2 \log n)$ или $O(n^3)$. В этом нет никакого противоречия. Эти границы проектировались как хорошие для любых ситуаций, даже при очень больших C относительно m и n. Но для задачи о двудольном паросочетании $C = n$, так что затраты, связанные с усложнением, в этом случае оказываются излишними.

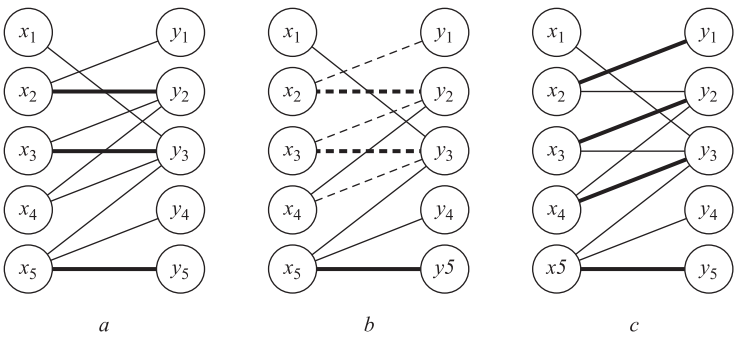


Рис. 7.10. *a* — двудольный граф с паросочетанием M; *b* — увеличивающий путь в соответствующем остаточном графе; *c* — паросочетание, полученное в результате увеличения

Также стоит задуматься над смыслом улучшающих путей в сети G'. Рассмотрим паросочетание M, состоящее из ребер (x_2, y_2), (x_3, y_3) и (x_5, y_5) в двудольном графе на рис. 7.1 (также см. рис. 7.10). Обозначим f соответствующий поток в G'. Это па-

росочетание не является максимальным, поэтому f не является максимальным потоком s–t, а следовательно, существует увеличивающий путь в остаточном графе G'_f. Один из таких увеличивающих путей помечен на рис. 7.10, *b*. Обратите внимание: ребра (x_2, y_2) и (x_3, y_3) используются в обратном направлении, а все остальные ребра — в прямом. Все увеличивающие пути должны чередоваться между ребрами, используемыми в обратном и прямом направлении, так как все ребра графа G' переходят из X в Y. По этой причине в контексте поиска максимального паросочетания увеличивающие пути называются *чередующимися путями*. Увеличение направлено на то, чтобы вывести из паросочетания ребра, идущие в обратном направлении, и заменить их ребрами, идущими в прямом направлении. Так как увеличивающий путь идет от s к t, прямых ребер на 1 больше, чем обратных; следовательно, размер паросочетания увеличивается на 1.

Расширения: структура двудольных графов без идеального паросочетания

С алгоритмической точки зрения мы уже знаем, как найти идеальное паросочетание: нужно использовать приведенный выше алгоритм для нахождения максимального паросочетания, а потом проверить, является ли это паросочетание идеальным.

Но пока зададимся чуть менее алгоритмическим вопросом. Не во всех двудольных графах существуют идеальные паросочетания. Как выглядит двудольный граф без идеального паросочетания? Существует ли простой способ определить, что двудольный граф не имеет идеального паросочетания — или по крайней мере убедить кого-то в том, что граф не имеет идеального паросочетания, после выполнения алгоритма? А если говорить конкретнее, было бы хорошо, если бы алгоритм после принятия решения об отсутствии идеального паросочетания мог выдать короткий «сертификат», подтверждающий его отсутствие. С таким сертификатом любой желающий мог бы быстро убедиться в отсутствии идеального паросочетания без отслеживания всего хода выполнения алгоритма.

Чтобы понять идею такого сертификата, на происходящее можно было бы взглянуть так, чтобы решить, имеет ли граф G идеальное паросочетание, мы могли бы проверить, что величина максимального потока в графе G' не ниже n. Согласно теореме о максимальном потоке и минимальном разрезе, разрез s–t с пропускной способностью ниже n существует в том случае, если величина максимального потока в G' ниже n. Итак, разрез с пропускной способностью ниже n мог бы послужить таким сертификатом. Однако нам нужен сертификат, обладающий естественным смыслом в контексте исходного графа G.

Как мог бы выглядеть такой сертификат? Например, если существуют узлы $x_1, x_2 \in X$, каждый из которых имеет только одно инцидентное ребро, а другим концом каждого ребра является один и тот же узел y, очевидно, такой граф не имеет идеального паросочетания: и x_1 и x_2 должны будут оказаться в паре с одним узлом y. На более общем уровне рассмотрим подмножество узлов $A \subseteq X$; пусть $\Gamma(A) \subseteq Y$ обозначает множество всех узлов, смежных с узлами в A. Если граф имеет

идеальное паросочетание, то все узлы в A должен находиться в парах с разными узлами в $\Gamma(A)$, так что множество $\Gamma(A)$ должно быть по крайней мере не меньше A. Мы приходим к следующему факту.

(7.39) Если двудольный граф $G = (V, E)$ с двумя сторонами X и Y имеет идеальное паросочетание, то для всех $A \subseteq X$ должно выполняться условие $|\Gamma(A)| \geq |A|$.

Это утверждение подсказывает, какой сертификат мог бы демонстрировать отсутствие идеального паросочетания у графа: множество $A \subseteq X$, для которого $|\Gamma(A)| < |A|$. Но будет ли истинным также условие, обратное (7.39)? Действительно ли в любой ситуации при отсутствии идеального паросочетания существует подобное множество A, которое это доказывает? Ответ на этот вопрос оказывается положительным, если добавить очевидное условие $|X| = |Y|$ (без которого идеального паросочетания, конечно, быть не может). Это утверждение известно в литературе как *теорема Холла*, хотя его разновидности были независимо открыты несколькими учеными (вероятно, первым был Кёниг) в начале 1900-х годов. Из доказательства утверждения также вытекает способ нахождения такого подмножества A за полиномиальное время.

(7.40) Предположим, в двудольном графе $G = (V,E)$ имеются две стороны X и Y, для которых $|X| = |Y|$. В этом случае граф G либо имеет идеальное паросочетание, либо существует такое подмножество $A \subseteq X$, что $|\Gamma(A)| < |A|$. Идеальное паросочетание или соответствующее подмножество A могут быть найдены за время $O(mn)$.

Доказательство. Воспользуемся тем же графом G', что и в (7.37). Будем считать, что $|X| = |Y| = n$. Согласно (7.37) граф G имеет максимальное паросочетание в том и только в том случае, если величина максимального потока в G' равна n.

Необходимо показать, что если величина максимального потока меньше n, то существует подмножество A, для которого $|\Gamma(A)| < |A|$, как указано в утверждении. По теореме о максимальном потоке и минимально разрезе (7.12), если величина максимального потока меньше n, то существует разрез (A', B') с пропускной способностью менее n в G'. Множество A' содержит s, и может содержать узлы как из X, так и из Y, как показано на рис. 7.11. Утверждается, что множество $A = X \cap A'$ обладает заявленным свойством. Если нам удастся это доказать, то тем самым будут доказаны обе части утверждения, так как в (7.11) было показано, что минимальный разрез (A', B') также может быть найден выполнением алгоритма Форда–Фалкерсона.

Начнем с того, что минимальный разрез (A', B') можно изменить так, чтобы гарантировать выполнение условия $\Gamma(A) \subseteq A'$ (как и прежде, $A = X \cap A'$). Для этого рассмотрим узел $y \in \Gamma(A)$, принадлежащий B', как показано на рис. 7.11, *а*. Утверждается, что перемещение y из B' в A' не увеличивает емкость разреза. Действительно, что происходит при перемещении y из B' в A'? Ребро (y, t) теперь пересекает разрез, увеличивая пропускную способность на 1. Но ранее было по крайней мере одно ребро (x, y) с $x \in A$, так как $y \in \Gamma(A)$; разрез пересекали все ребра из A и y, а теперь ситуация изменилась, поэтому общая пропускная способность разреза не может возрасти. (Обратите внимание: нам не нужно беспокоиться об узлах $x \in X$, не входящих в A. Два конца ребра (x, y) будут находиться на разных сторонах разреза, но само ребро не увеличивает пропускную способность разреза, потому что проходит из B' в A'.)

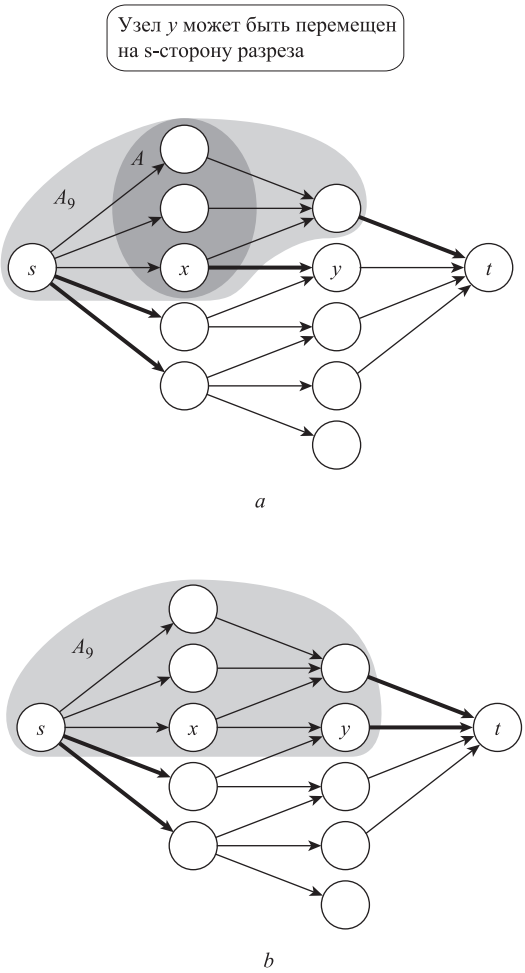


Рис. 7.11. *a* — минимальный разрез из доказательства (7.40); *b* — тот же разрез после перемещения узла y на сторону A'. Ребра, пересекающие разрез, выделены жирными линиями

Теперь рассмотрим пропускную способность минимального разреза (A', B'), у которого $\Gamma(A) \subseteq A'$ (рис. 7.11, *b*). Так как все соседи A принадлежат A', мы видим, что из A' выходят только ребра, либо выходящие из источника s, либо входящие в сток t. Следовательно, пропускная способность разреза равна в точности

$$c(A', B') = |X \cap B'| + |Y \cap A'|.$$

Однако $|X \cap B'| = n - |A|$, а $|Y \cap A'| \geq |\Gamma(A)|$. Из предположения о том, что $c(A', B') < n$, следует, что

$$n - |A| + |\Gamma(A)| \leq |X \cap B'| + |Y \cap A'| = c(A', B') < n.$$

Сравнивая первую и последнюю части, получаем требуемое неравенство $|A| > |\Gamma(A)|$.

7.6. Непересекающиеся пути в направленных и ненаправленных графах

В разделе 7.1 мы описывали поток f как своего рода сетевой «трафик». Однако наше определение потока выглядит статично: с каждым ребром e просто связывается число $f(e)$, которое определяет величину потока, проходящего через e. Посмотрим, нельзя ли перейти к более динамичной концепции, ориентированной на сетевой трафик, и формализовать концепцию «перемещения» потока из источника к стоку. Это динамическое представление потока приводит нас к *задаче о непересекающихся путях*.

Задача

Для точного определения задачи необходимо разобраться с двумя аспектами. Во-первых, мы дадим точное определение интуитивного соответствия между единицами потока, перемещающимися по путям, и понятием потока, использовавшимся до настоящего времени. Во-вторых, мы расширим задачу непересекающихся путей на ненаправленные графы. Вы увидите, что несмотря на то, что задача нахождения максимального потока определялась для направленных графов, она может естественным образом использоваться для решения взаимосвязанных задач для ненаправленных графов.

Множество путей является *непересекающимся по ребрам*, если их множества ребер не пересекаются, то есть никакие два пути не содержат ни одного общего ребра, даже если они проходят через некоторые общие узлы. Для заданного направленного графа $G = (V, E)$ с двумя выделенными узлами $s, t \in V$ задача *непересекающихся по направленным ребрам путей* заключается в нахождении максимального количества путей s–t, непересекающихся по направленным ребрам. Задача *непересекающихся по ненаправленным ребрам путей* заключается в нахождении максимального количества путей s–t в ненаправленном графе G. Родственный вопрос нахождения путей, непересекающихся не только ребрам, но и по узлам (разумеется, кроме узлов s и t), будет рассматриваться в упражнениях этой главы.

Разработка алгоритма

Как направленная, так и ненаправленная версия задачи исключительно естественно решаются с использованием потоков. Начнем с направленной задачи. Для графа $G = (V, E)$ с двумя выделенными узлами s и t мы определим потоковую сеть, в которой s и t — источник и сток соответственно, а каждое ребро имеет про-

пускную способность 1. Предположим, существуют k путей s–t, непересекающихся по ребрам. Мы можем сделать так, чтобы по каждому из таких путей передавалась одна единица потока: в каждом пути одному ребру e назначается $f(e) = 1$, а всем остальным ребрам — $f(e') = 0$; так определяется допустимый поток с величиной k.

(7.41) Если в направленном графе G от s к t существуют k путей, непересекающихся по ребрам, то величина максимального потока s–t в G не меньше k.

Предположим, мы также сможем продемонстрировать утверждение, обратное (7.41): если существует поток с величиной k, то существуют k путей s–t, непересекающихся по ребрам. Тогда мы просто сможем вычислить максимальный поток s–t в G, и объявить (обоснованно) это значение максимальным количеством непересекающихся по ребрам путей s–t.

Начнем с доказательства этого обратного утверждения, которое позволит убедиться в том, что метод с потоком действительно дает правильный ответ. В ходе анализа также будет получен способ извлечения k путей, непересекающихся по ребрам, из целочисленного потока, передающего k единиц из s в t. Следовательно, вычисление максимального потока в G даст не только максимальное *количество* путей, непересекающихся по ребрам, но и сами пути.

Анализ алгоритма

Доказательство (7.41) в обратном направлении лежит в основе анализа, так как оно немедленно устанавливает оптимальность алгоритма на базе потока для нахождения непересекающихся путей.

Чтобы доказать это, рассмотрим поток с величиной не менее k и построим k путей, непересекающихся по ребрам. Согласно (7.14) мы знаем, что существует максимальный поток f с целыми величинами потоков. Так как пропускная способность всех ребер ограничивается 1, а поток является целочисленным, каждое ребро, передающее поток в f, имеет ровно одну единицу потока. Следовательно, необходимо доказать следующее.

(7.42) Если f — поток с величиной ν, состоящий из 0 и 1, то множество ребер с величиной потока $f(e) = 1$ содержит множество из ν путей, непересекающихся по ребрам.

Доказательство. Воспользуемся индукцией по количеству ребер в f, передающих поток. Если $\nu = 0$, доказывать нечего. В противном случае должно существовать ребро (s, u), передающее одну единицу потока. Проверим путь из ребер, которые должны передавать поток: так как (s, u) передает единицу потока, из ограничения сохранения потока следует, что должно существовать ребро (v, w), по которому передается единица потока; и т. д. Если продолжать перебор, со временем произойдет одно из двух: либо мы достигнем t, либо узел v будет достигнут во второй раз.

В первом случае мы нашли путь из s в t, и этот путь будет использоваться как один из наших путей v. Обозначим f' поток, полученный уменьшением величины потока на ребрах из P до 0. Новый поток f' имеет величину $v - 1$, и в нем меньше

ребер, передающих поток. Применяя индукционную гипотезу для f', мы получаем $v - 1$ путей, непересекающихся по ребрам, которые вместе с путем P образуют заявленные v путей.

Если P достигает узла v во второй раз, значит, ситуация выглядит примерно так, как показано на рис. 7.12. (Все ребра на схеме передают одну единицу потока, а пунктирные ребра обозначают текущий пройденный путь, который только что достиг узла v во второй раз.) В этом случае можно действовать иначе.

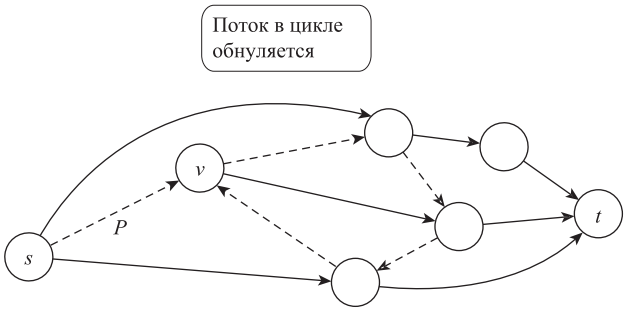


Рис. 7.12. Все ребра на схеме передают одну единицу потока. Путь P из пунктирных ребер — один из возможных путей в доказательстве (7.42)

Рассмотрим цикл C из ребер, посещенных между первым и вторым посещением v. Новый поток f' получается из f уменьшением величин потока на ребрах от C до 0. Новый поток f' имеет величину v, но меньшее количество ребер, передающих поток. Применяя индукционную гипотезу к f', мы получаем v путей, непересекающихся по ребрам, как и требовалось. ■

Объединение (7.41) и (7.42) дает следующий результат.

(7.43) В направленном графе G существуют k путей, непересекающихся по ребрам, в том и только в том случае, если величина максимального потока s–t в G не менее k.

Также обратите внимание на то, как доказательство (7.42) предоставляет процедуру построения k путей для заданного целочисленного максимального потока в графе G. Эта процедура иногда называется *декомпозицией потока*, потому что она «раскладывает» поток на множество путей. Соответственно мы показали, что наш алгоритм на базе потока находит максимальное количество путей s–t, непересекающихся по ребрам, а также предоставляет механизм построения самих путей.

Граница времени выполнения

Для этой задачи $C = \sum_{e \text{ out of } s} c_e \leq |V| = n$, так как существуют не более $|V|$ ребер из s, каждое из которых имеет пропускную способность 1. Следовательно, применяя границу $O(mC)$ из (7.5), мы получаем целочисленный максимальный поток за время $O(mn)$.

Процедуру декомпозиции из доказательства (7.42), позволяющую получить сами потоки, также можно привести к выполнению за время $O(mn)$. Для этого заметьте, что эта процедура при некоторых усилиях может построить один путь из s в t с максимум постоянной работой на ребро в графе — а следовательно, за время $O(m)$. Всего существуют максимум $n - 1$ путей из s в t, непересекающихся по ребрам (все они должны использовать разные ребра, выходящие из s), а следовательно, на построение всех путей потребуется время $O(mn)$.

Итак, мы показали следующее:

(7.44) Алгоритм Форда–Фалкерсона может использоваться для нахождения в направленном графе G максимального множества путей s–t, непересекающихся по ребрам, за время $O(mn)$.

Версия теоремы о максимальном потоке и минимальном разрезе для непересекающихся путей

При помощи теоремы о максимальном потоке и минимальном разрезе (7.13) можно получить оценку максимального количества путей s–t, непересекающихся по ребрам. Говорят, что множество ребер $F \subseteq E$ *отделяет* s от t, если после удаления ребер F из графа G в графе не остается ни одного пути s–t.

(7.45) В каждом направленном графе с узлами s и t максимальное количество путей s–t, непересекающихся по ребрам, равно минимальному количеству ребер, удаление которых приводит к отделению s от t.

Доказательство. Если удаление множества $F \subseteq E$ отделяет s от t, то каждый путь s–t должен использовать по крайней мере одно ребро из F, а следовательно, количество путей s–t, непересекающихся по ребрам, не превышает $|F|$.

Чтобы доказать другое направление, мы воспользуемся теоремой о максимальном потоке и минимальном разрезе (7.13). Согласно (7.43) максимальное количество путей, непересекающихся по ребрам, равно величине ν максимального потока s–t. Из (7.13) следует, что существует разрез s–t (A, B) с пропускной способностью ν. Обозначим F множество ребер, переходящих из A в B. Каждое ребро обладает пропускной способностью 1, так что $|F| = \nu$, и по определению разреза s–t удаление этих ν ребер из G приводит к отделению s от t. ■

Итак, этот результат может рассматриваться как естественный частный случай теоремы о максимальном потоке и минимальном разрезе, в котором пропускные способности всех ребер равны 1. Собственно, этот частный случай был доказан Менгером в 1927 году задолго до того, как полная теорема о максимальном потоке и минимальном разрезе была сформулирована и доказана; по этой причине (7.45) часто называется *теоремой Менгера*. Кстати говоря, в доказательстве теоремы Холла (7.40) для двудольных паросочетаний используется приведение к графу с единичными пропускными способностями ребер, поэтому для доказательства можно воспользоваться теоремой Менгера вместо общей теоремы о максимальном потоке и минимальном разрезе. Иначе говоря, теорема Холла является частным случаем теоремы Менгера, которая в свою очередь является частным случаем теоремы о максимальном потоке и минимальном разрезе. Этот порядок соответ-

ствует историческим событиям, потому что теоремы доказывались именно в таком порядке с интервалом в несколько десятилетий[1].

Расширения: непересекающиеся пути в ненаправленных графах

Наконец, рассмотрим задачу непересекающихся путей в ненаправленном графе G. Несмотря на тот факт, что граф G стал ненаправленным, для получения путей в G, непересекающихся по ребрам, можно воспользоваться алгоритмом максимального потока. Идея проста: каждое ненаправленное ребро (u, v) заменяется двумя направленными ребрами (u, v) и (v, u); так создается направленная версия G' графа G. (Ребра, входящие в s и выходящие из t, можно удалить, так как они бесполезны). Теперь к полученному направленному графу применяется алгоритм Форда–Фалкерсона. Тем не менее сначала необходимо разобраться с одним важным аспектом. Обратите внимание: два пути P_1 и P_2 могут быть непересекающимися по ребрам в направленном графе, но при этом использовать одно ребро в ненаправленном графе G: это происходит, когда P_1 использует направленное ребро (u, v), а P_2 использует ребро (v, u). Тем не менее, нетрудно убедиться в том, что всегда существует максимальный поток в любой сети, использующей не более одного из каждой пары противоположно направленных ребер.

(7.46) В любой потоковой сети существует максимальный поток f, в котором для всех противоположно направленных ребер $e = (u, v)$ и $e' = (v, u)$ либо $f(e) = 0$, либо $f(e') = 0$. Если пропускные способности потоковой сети являются целочисленными, то также существует такой целочисленный максимальный поток.

Доказательство. Рассмотрим максимальный поток f и изменим его так, чтобы он удовлетворял заявленному условию. Пусть $e = (u, v)$ и $e' = (v, u)$ — противоположно направленные ребра, $f(e) \neq 0$ и $f(e') \neq 0$. Обозначим δ меньшее из этих значений и изменим f, сократив величину потока в e и e' на δ. Полученный поток f' является допустимым; обладает такой же величиной, как f; а его величина на одном из ребер e и e' равна 0.

Теперь мы воспользуемся алгоритмом Форда–Фалкерсона и процедурой декомпозиции пути из (7.42) для получения путей, непересекающихся по ребрам, в ненаправленном графе G.

(7.47) В ненаправленном графе G существуют k путей из s в t, непересекающихся по ребрам, в том и только в том случае, если максимальная величина потока s–t в направленной версии G' графа G равна по крайней мере k. Кроме того, алгоритм

[1] В интересном очерке, написанном в 1981 году, Менгер излагает свою версию истории о том, как он объяснял свою теорему Кёнигу, одному из независимых первооткрывателей теоремы Холла. Казалось бы, Кёниг, много размышлявший над подобными задачами, должен был немедленно понять, почему обобщение его теоремы Менгером было истинным — и возможно, даже посчитать его очевидным. Но в действительности все было наоборот: Кёниг не поверил, что теорема истинна, и провел целую ночь в поисках контрпримера. На следующий день, выбившись из сил, он обратился к Менгеру за доказательством.

Форда–Фалкерсона может использоваться для нахождения максимального множества непересекающихся путей s–t в ненаправленном графе G за время $O(mn)$.

Ненаправленная аналогия (7.45) тоже истинна, так как в любом разрезе s–t не более чем одно из двух противоположно направленных ребер может переходить со стороны s на сторону t разреза (потому что если одно ребро следует в нужном направлении, то другое должно проходить со стороны t на сторону s).

(7.48) В каждом ненаправленном графе с узлами s и t максимальное количество путей s–t, непересекающихся по ребрам, равно минимальному количеству ребер, удаление которых приводит к отделению s от t.

7.7. Расширения задачи о максимальном потоке

Полезность задачи о максимальном потоке в основном не имеет никакого отношения к самому факту моделирования сетевого трафика. Скорее она заключается в том, что многие задачи с нетривиальной составляющей комбинаторного поиска могут быть решены за полиномиальное время, потому что их удается свести к задаче нахождения максимального потока или минимального разреза в направленном графе.

Двудольные паросочетания — первое естественное практическое применение задачи в этом направлении; в следующих разделах рассматриваются и другие применения. На первых порах мы сохраним представление о потоке как об абстрактном «трафике» и будем искать более общие условия, которым этот трафик может подчиняться. Как выясняется, эти более общие условия могут принести пользу в будущем.

В частности, особое внимание будет уделено двум обобщениям задачи о максимальном потоке. Вы увидите, что оба случая приводятся к базовой задаче о максимальном потоке.

Задача: циркуляция с потреблением

Одна из упрощающих особенностей исходной формулировки задачи о максимальном потоке заключалась в том, что графе содержал только один источник s и один сток t. Теперь предположим, что в графе присутствует множество S источников, генерирующих поток, и множество T стоков, поглощающих поток. Как и прежде, каждое ребро обладает целочисленной пропускной способностью.

С несколькими источниками и стоками не совсем понятно, как решить, какому из источников или стоков следует отдать предпочтение в задаче максимизации. Вместо того чтобы максимизировать величину потока, мы рассмотрим ситуацию, в которой источники имеют фиксированную величину поставки, а стоки — фиксированную величину потребления, а наша цель заключается в передаче потока

от узлов со свободной поставкой к узлам с заданным потреблением. Представьте, например, что сеть представление систему дорог или железнодорожных линий, которая используется для поставки товаров с фабрик (поставка) в торговые точки (потребление). В такой постановке задачи нас будет интересовать не максимизация конкретной величины, а удовлетворение всего спроса с использованием доступной поставки.

Следовательно, мы имеем потоковую сеть $G = (V, E)$ с пропускными способностями ребер. С каждым узлом $v \in V$ связывается уровень *потребления* d_v. Если $d_v > 0$, значит, узел v потребляет d_v единиц потока; узел выполняет функции стока и желает получать на d_v единиц больше потока, чем отправлять далее. Если $d_v < 0$, значит v имеет уровень потребления $-d_v$; он является источником, и хочет отправлять на $-d_v$ единиц потока больше, чем получает. Если $d_v = 0$, то узел v не является ни источником, ни стоком. Будем считать, что все пропускные способности и уровни потребления являются целыми числами.

Обозначим S множество всех узлов с отрицательным потреблением, и T — множество всех узлов с положительным потреблением. Хотя узел v в S хочет передавать больше потока, чем получает, он может иметь поток по входящим ребрам; он будет компенсироваться потоком из v по выходящим ребрам. Сказанное относится (в противоположном направлении) и к множеству T.

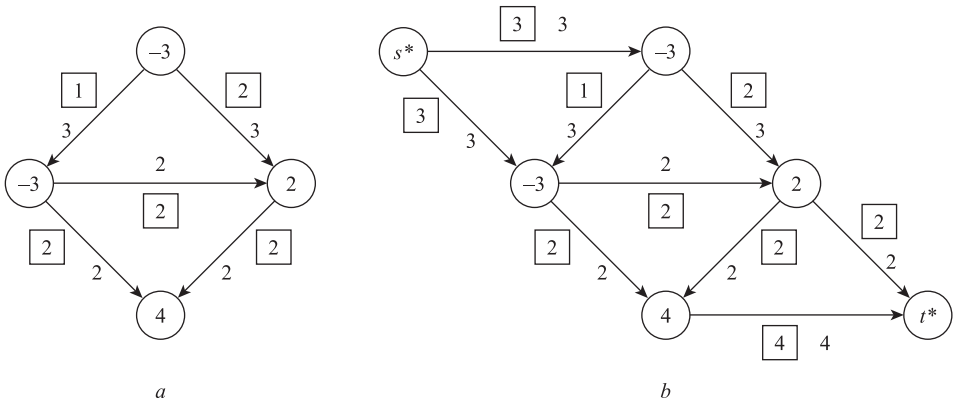


Рис. 7.13. *a* — экземпляр задачи о циркуляции с решением: в узлах указаны уровни потребления; на ребрах — пропускные способности и величины потока, а в прямоугольниках — величины потока; *b* — результат преобразования этого экземпляра в эквивалентный экземпляр задачи о максимальном потоке

В этой ситуации *циркуляцией* с потреблением $\{d_v\}$ называется функция f, которая связывает с каждым ребром неотрицательное вещественное число и удовлетворяет следующим двум условиям:

- (i) (Ограничения пропускной способности) Для всех $e \in E$ выполняется условие $0 \le f(e) \le c_e$.
- (ii) (Ограничения сохранения потока) Для каждого узла $v \in V$ выполняется условие $v, f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$.

Сейчас вместо задачи максимизации нас интересует *задача существования*: требуется узнать, *существует ли* действительная циркуляция (то есть циркуляция, удовлетворяющая условиям (i) и (ii)).

Для примера рассмотрим экземпляр задачи на рис. 7.13, *а*. На этой схеме два узла являются источниками (уровни потребления −3 и −3); два узла являются стоками, с уровнями потребления 2 и 4. Величины потока на схеме образуют действительную циркуляцию, так как все уровни потребления удовлетворяются в соответствии с пропускными способностями.

Если рассмотреть произвольный экземпляр задачи о циркуляции, существует простое условие, которое должно выполняться для существования действительной циркуляции: суммарная поставка должна быть равна суммарному потреблению.

(7.49) Если существует действительная циркуляция с потреблением $\{d_v\}$, то $\sum_v d_v = 0$.

Доказательство. Допустим, действительная циркуляция f существует. Тогда $\sum_v d_v = \sum_v f^{\text{in}}(v) - f^{\text{out}}(v)$. В последнем выражении величина $f(e)$ для каждого ребра $e = (u, v)$ подсчитывается дважды: в $f^{\text{out}}(u)$ и в $f^{\text{in}}(v)$. Эти два слагаемых компенсируются; а поскольку это происходит для всех величин $f(e)$, общая сумма равна 0. ■

Из (7.49) известно, что

$$\sum_{v:d_v>0} d_v = \sum_{v:d_v<0} -d_v.$$

Обозначим эту величину D.

Разработка и анализ алгоритма для циркуляций

Как выясняется, задача поиска действительной циркуляции с уровнями потребления $\{d_v\}$ сводится к задаче нахождения максимального потока s–t в другой сети, как показано на рис. 7.14.

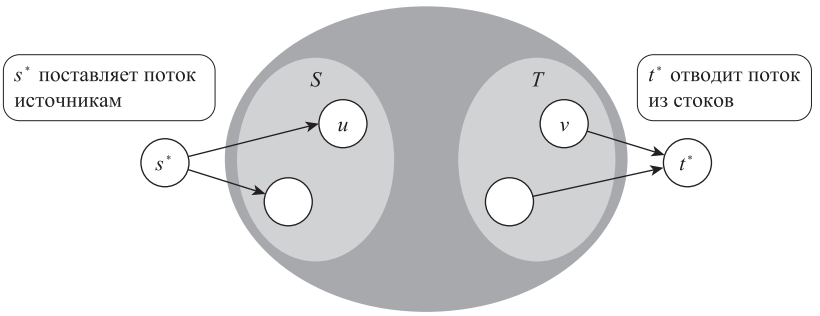


Рис. 7.14. Сведение задачи о циркуляции к задаче о максимальном потоке

Преобразование очень похоже на то, которое использовалось в задаче о двудольном паросочетании: каждый узел в S связывается с «супериcточником» s^*, а каждый узел в T — с «суперстоком» t. А конкретнее, граф G' строится на основе G добавлением в него новых узлов s^* и t^*. Для каждого узла $v \in T$ — то есть для каждого узла v с $d_v > 0$ — добавляется ребро (v,t^*) с пропускной способностью d_v. Для каждого узла $u \in S$ — то есть для каждого узла с $d_u < 0$ — добавляется ребро (s^*, u) с пропускной способностью $-d_u$. Остальная структура G переносится в G' без изменений.

В графе G' мы будем искать максимальный поток s^*–t^*. На интуитивном уровне это преобразование можно рассматривать как добавление узла s^*, «поставляющего» всем источникам их дополнительный поток, и узла t^*, «сливающего» лишний поток из стоков. Так, в части (b) рис. 7.13 показан результат применения этого преобразования к экземпляру из части (a).

Обратите внимание: в G' не может быть потока s^*–t^* с величиной больше D, поскольку разрез (A, B) с $A = \{s^*\}$ имеет пропускную способность D. Далее, если в G существует действительная циркуляция f с уровнями потребления $\{dv\}$, то отправляя величину потока $-d_v$ по каждому ребру (s^*, v) и величину потока d_v по каждому ребру (v,t^*), мы получим в G' поток s^*–t^* с величиной D, который является максимальным. И наоборот, предположим, что в G' существует (максимальный) поток s^*–t^* с величиной D. При этом каждое ребро, выходящее из s^*, и каждое ребро, входящее в t^*, полностью насыщено потоком. Таким образом, при удалении этих ребер мы получим циркуляцию в G с $f^{in}(v) - f^{out}(v) = d_v$ для каждого узла v. Кроме того, если в G' существует поток с величиной D, то этот поток содержит целочисленные значения.

Итак, мы доказали следующее:

(7.50) Действительная циркуляция с уровнями потребления $\{d_v\}$ в G существует в том и только в том случае, если максимальный поток s^*–t^* в G' имеет величину D. Если все пропускные способности и уровни потребления в G заданы целыми числами и существует действительная циркуляция, то эта действительная циркуляция является целочисленной.

В конце раздела 7.5 теорема о максимальном потоке и минимальном разрезе использовалась для получения характеристики (7.40) двудольных графов, не имеющих идеального паросочетания. Аналогичную характеристику можно определить и для графов, не имеющих действительной циркуляции. В этой характеристике используется понятие разреза, адаптированное для текущей ситуации. В контексте задачи циркуляции с потреблением разрезом (A, B) называется разбиение множества узлов V на два подмножества без каких-либо ограничений относительно того, на какой стороне разбиения окажутся источники и стоки. Мы приводим характеристику без доказательства.

(7.51) Граф G имеет действительную циркуляцию с уровнями потребления $\{d_v\}$ в том и только том случае, если для всех разрезов (A, B)

$$\sum_{v \in B} d_v \leq c(A,B).$$

Важно заметить, что в нашей сети существует только один «тип» потока. Хотя поток поступает из нескольких источников и поглощается несколькими стоками, мы не можем установить ограничения относительно того, какой источник будет поставлять поток тому или иному стоку; это решает алгоритм. В более сложной *задаче многопродуктового потока* сток t_i должен получать поток, поставляемый источником s_i для всех i. Эта тема более подробно рассматривается в главе 11.

Задача: циркуляция с потреблением и нижние границы

Наконец, немного обобщим приведенную задачу. Во многих ситуациях требуется не только обеспечить потребление в разных узлах, но и заставить поток использовать некоторые ребра. Для этого можно установить для ребер нижние границы наряду с обычными верхними границами, определяемыми пропускными способностями ребер.

Рассмотрим потоковую сеть $G = (V, E)$ с *пропускной способностью* c_e и *нижней границей* ℓ_e для каждого ребра e. Будем считать, что $0 \le \ell_e \le c_e$ для всех e. Как и прежде, каждому узлу v также назначен уровень потребление d_v, который может быть как положительным, так и отрицательным. Предполагается, что все уровни потребления, пропускные способности и нижние границы являются целыми числами.

Основные величины имеют тот же смысл, что и прежде, а нижняя граница ℓ_e означает, что величина потока через e должна быть не меньше ℓ_e. Таким образом, циркуляция в потоковой сети должна удовлетворять следующим двум условиям.

(i) (Ограничения пропускной способности.) Для всех $e \in E$ выполняется условие $\ell_e \le f(e) \le c_e$.

(ii) (Ограничения уровня потребления.) Для всех $v \in V$ выполняется условие $f^{in}(v) - f^{out}(v) = d_v$.

Как и прежде, требуется решить, существует ли *действительная циркуляция* — то есть циркуляция, удовлетворяющая этим требованиям.

Разработка и анализ алгоритма с нижними границами

Наша стратегия заключается в сведении этой задачи к задаче нахождения циркуляции с уровнями потребления, но без нижних границ. (Как вы уже видели, последняя задача может быть сведена к стандартной задаче о максимальном потоке.) Идея заключается в следующем: мы знаем, что по каждому ребру e необходимо передавать как минимум ℓ_e единиц потока. Предположим, исходная циркуляция f_0 определяется просто как $f_0(e) = \ell_e$. f_0 удовлетворяет всем ограничениям пропускной способности (для нижних и верхних границ), но, вероятно, не удовлетворяет всем ограничениям потребления. В частности,

$$f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{e \text{ into } v} \ell_e - \sum_{e \text{ out of } v} \ell_e.$$

Обозначим эту величину L_v. Если $L_v = d_v$, то ограничение потребления для v выполняется, а если нет — нужно наложить поверх f_0 циркуляцию f_1, которая исправит «дисбаланс» в v. Следовательно, потребуется $f_1^{\text{in}}(v) - f_1^{\text{out}}(v) = d_v - L_v$. И какой пропускной способностью мы для этого располагаем? По каждому ребру уже передаются ℓ_e единиц потока, поэтому для использования доступны еще $c_e - \ell_e$ единиц.

Эти соображения заложены в основу следующего построения. Пусть граф G' состоит из тех же узлов и ребер с пропускными способностями и уровнями потребления, но без нижних границ. Пропускная способность ребра e будет равна $c_e - \ell_e$. Уровень потребления узла v будет равен $d_v - L_v$.

Например, возьмем экземпляр графа на рис. 7.15(а). Перед вами тот же экземпляр, который был показан на рис. 7.13, но на этот раз одному из ребер назначена нижняя граница 2. В части *b* эта нижняя граница устраняется, что приводит к снижению верхней границы для ребра и изменению потребления на двух концах ребра. В процессе становится ясно, что действительной циркуляции не существует, так как после применения этой конструкции появляется узел с уровнем потребления –5 и только 4 единицами пропускной способности на выходящих ребрах.

Утверждается, что наша общая конструкция создает эквивалентный экземпляр задачи с уровнями потребности, но без нижних границ; к этой задаче можно применить общий алгоритм.

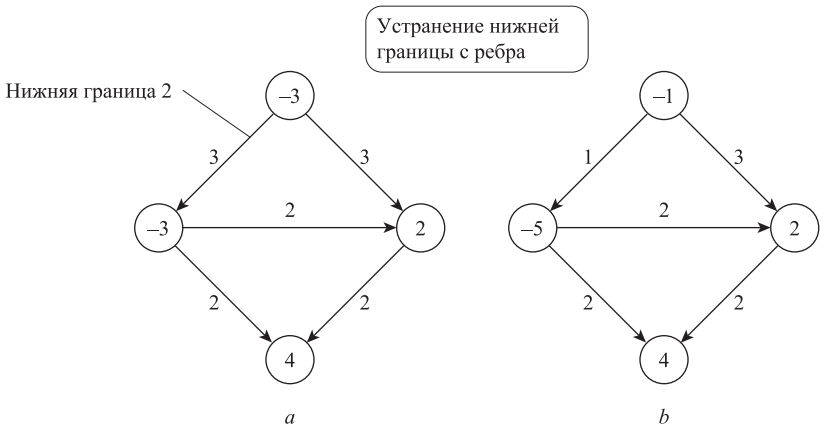


Рис. 7.15. *a* — экземпляр задачи о циркуляции с нижними границами: числа в узлах — уровни потребности, числа у ребер — пропускные способности. Одному из ребер назначается нижняя граница 2; *b* — результат преобразования этого элемента в эквивалентный экземпляр задачи о циркуляции без нижних границ

(7.52) Действительная циркуляция в G существует в том и только в том случае, если существует действительная циркуляция в G'. Если все уровни потребления,

пропускные способности и нижние границы в G являются целыми числами и существует действительная циркуляция, то эта действительная циркуляция является целочисленной.

Доказательство. Сначала предположим, что в G' существует циркуляция f'. Определим циркуляцию f в G по формуле $f(e) = f'(e) + \ell_e$. В этом случае f удовлетворяет ограничениям пропускной способности в G, и

$$f^{\text{in}}(v) - f^{\text{out}}(v) = \sum_{e \text{ into } v} (\ell_e + f'(e)) - \sum_{e \text{ out of } v} (\ell_e + f'(e)) = L_v + (d_v - L_v) = d_v,$$

поэтому также выполняются ограничения потребления в G.

И наоборот, предположим, что в G существует циркуляция f, и определим в G' циркуляцию f' по формуле $f'(e) = f(e) - \ell_e$. В этом случае f' удовлетворяет ограничениям пропускной способности в G' и

$$(f')^{\text{in}}(v) - (f')^{\text{out}}(v) = \sum_{e \text{ into } v} (f(e) - \ell_e) - \sum_{e \text{ out of } v} (f(e) - \ell_e) = d_v - L_v$$

поэтому также выполняются ограничения потребления в G'.

7.8. Планирование опроса

Многие практические задачи эффективно решаются сведением к задаче о максимальном потоке, но часто бывает трудно определить, когда такое сведение возможно. В нескольких ближайших разделах рассматриваются некоторые хрестоматийные примеры такого рода. В них мы постараемся показать, как обычно выглядят преобразования такого рода, и продемонстрировать наиболее распространенные способы применения потоков и разрезов при проектировании эффективных комбинаторных алгоритмов. Как выясняется, иногда решение базируется на вычислении максимального потока, а иногда на вычислении минимального разреза; и потоки, и разрезы являются чрезвычайно полезными алгоритмическими инструментами.

Начнем с несложного примера, который назовем *планированием опроса* — упрощенной версии задачи, с которой сталкиваются многие компании, желающие получить информацию о качестве обслуживания клиентов. На более общем уровне задача показывает, как конструкция, использовавшаяся для решения задачи о двудольном паросочетании, естественным образом возникает в любой среде, в которой требуется тщательно сбалансировать решения по набору вариантов — в данном случае при разработке анкеты с распределением вопросов по группе потребителей.

Задача

Одной из важных тем в бурно развивающейся области анализа данных является изучение закономерностей потребительских предпочтений. Допустим, компания продает k продуктов и ведет базу данных с историями покупок по большой группе клиентов. (Обладатели карт «Клуба покупателей» догадаются, как собираются такие данные.) Компания желает провести опрос и разослать индивидуальные анкеты в группе из n своих клиентов, чтобы определить, какие продукты больше нравятся покупателям.

Планирование опроса подчиняется некоторым правилам:

- Каждый клиент получает вопросы, относящиеся к определенному подмножеству продаваемых продуктов.
- Клиенту можно задавать вопросы только о тех продуктах, которые он покупал.
- Анкета не должна быть слишком длинной, чтобы не отбить у клиента желание участвовать в опросе, поэтому каждому клиенту задаются вопросы по продуктам из диапазона от c_i до c'_i.
- Наконец, для получения достаточного объема информации о продукте j необходимо опросить от p_j до p'_j разных клиентов.

В более формальном представлении входные данные задачи планирования опроса представляют собой двудольный граф G, узлы которого представляют клиентов и продукты, а ребро между клиентом i и продуктом j существует в том случае, если клиент покупал данный продукт. Кроме того, для каждого клиента $i = 1, ..., n$ установлены ограничения $c_i \le c'_i$ на количество продуктов, о которых его можно спрашивать; для каждого продукта $j = 1, ..., k$ установлены ограничения $p_j \le p'_j$ на количество разных клиентов, которым будут задаваться вопросы. Требуется решить, возможно ли построить анкеты, с которыми будут выполнены все указанные ограничения.

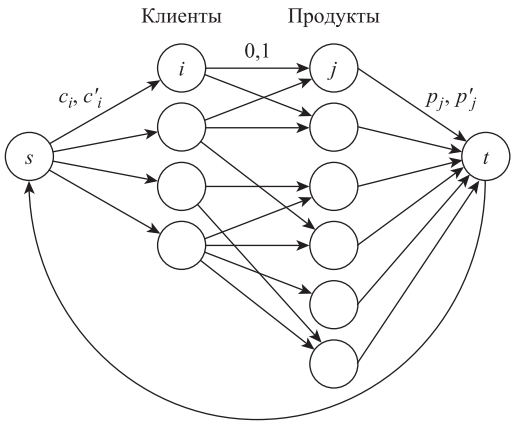


Рис. 7.16. Задача планирования опроса сводится к задаче нахождения действительной циркуляции: поток переходит от клиентов (ограничения пропускной способности определяют количество задаваемых вопросов) к продуктам (ограничения пропускной способности определяют количество вопросов, задаваемых по каждому продукту)

Разработка алгоритма

Мы решим эту задачу преобразованием к задаче циркуляции для потоковой сети G' с уровнями потребления и нижними границами, показанными на рис. 7.16. Чтобы получить граф G' из G, мы ориентируем ребра G от клиентов к продуктам, добавляем узлы s и t с ребрами (s, i) для каждого клиента $i = 1, ..., n$, ребрами (j, t) для каждого продукта $j = 1, ..., k$ и ребром (t, s). Циркуляция в такой сети соответствует распределению вопросов. Поток в ребре (s, i) определяет количество продуктов, включаемых в анкету для клиента i, поэтому ребро будет иметь пропускную способность c'_i и нижнюю границу c_i. Поток в ребре (j, t) соответствует количеству клиентов, которым задаются вопросы о продукте j, поэтому ребро будет иметь пропускную способность p'_j и нижнюю границу p_j. Каждое ребро (i, j), проходящее от клиента к купленному им продукту, имеет пропускную способность 1 с нижней границей 0. Поток, передаваемый ребром (t, s), соответствует общему количеству заданных вопросов. Мы можем присвоить этому ребру пропускную способность $\sum_i c'_i$ и нижнюю границу $\sum_i c_i$. У всех узлов уровень потребления равен 0.

Наш алгоритм должен просто построить сеть G' и проверить, существует ли в ней действительная циркуляция. Сформулированное ниже утверждение устанавливает правильность этого алгоритма.

Анализ алгоритма

(7.53) В графе G', построенном описанным способом, действительная циркуляция существует в том и только в том случае, если существует действительный вариант планирования опроса.

Доказательство. Приведенное описание подсказывает способ преобразования плана опроса в соответствующий поток. Ребро (i, j) передает одну единицу потока, если клиенту i задается вопрос о продукте j в опросе, и не передает поток в противном случае. Поток в ребрах (s, i) представляет количество вопросов, заданных клиенту i, поток в ребре (j, t) — количество клиентов, которым заданы вопросы о продукте j, и, наконец, поток в ребре (t, s) — общее количество заданных вопросов. Такой поток соответствует 0 потреблению, то есть в каждом узле происходит сохранение потока. Если опрос удовлетворяет этим правилам, то для соответствующего потока выполняются пропускные способности и нижние границы.

И наоборот, если задача циркуляции имеет действительное решение, то согласно (7.52) существует действительная целочисленная циркуляция, у которой имеется естественное соответствие в виде действительного плана опроса. Клиенту i будут заданы вопросы о продукте j в том и только том случае, если ребро (i, j) передает единицу потока.

7.9. Планирование авиаперелетов

Вычислительные задачи, с которыми сталкиваются крупные национальные авиакомпании, трудно себе даже представить. Им приходится ежедневно строить расписания тысяч маршрутов, эффективные в отношении амортизации оборудования, распределения экипажа, удовлетворения клиентов и множества других факторов — и все это с учетом таких непредсказуемых факторов, как погода и поломки. Неудивительно, что авиакомпании входят в число крупнейших пользователей мощных алгоритмических методов.

Рассмотрение таких вычислительных задач на любом сколько-нибудь реалистичном уровне слишком сильно уведет нас в сторону. Вместо этого мы обсудим «игрушечную» задачу, которая довольно четко отражает некоторые проблемы распределения ресурсов, возникающие в подобных контекстах. И как это уже не раз бывало в книге, «игрушечная» задача для наших целей окажется намного полезнее настоящей, поскольку ее решение основано на универсальных методах, которые могут применяться в самых разных ситуациях.

Задача

Допустим, вы отвечаете за управление самолетным парком и хотите построить расписание полетов. Ниже описана очень простая модель. В ходе маркетинговых исследований были определены m рейсовых сегментов, обслуживание которых могло бы принести высокую прибыль; сегмент j определяется четырьмя параметрами: аэропортом отправления, аэропортом прибытия, временем отправления и временем прибытия. На рис. 7.17, *a* изображен простой пример с шестью сегментами, которые вам хотелось бы обслуживать в течение одного дня:

(1) Бостон (BOS, отправление 6:00) — Вашингтон (DCA, прибытие 7:00)
(2) Филадельфия (PHL, отправление 7:00) — Питтсбург (PIT, прибытие 8:00)
(3) Вашингтон (DCA отправление 8:00) — Лос-Анджелес (LAS, прибытие 11:00)
(4) Филадельфия (PHL, отправление 11:00) — Сан-Франциско (SFO, прибытие 14:00)
(5) Сан-Франциско (SFO, отправление 14:15) — Сиэтл (SEA, прибытие 15:15)
(6) Лас-Вегас (LAX, отправление 17:00) — Сиэтл (SEA, прибытие 18:00)

В каждый сегмент включается время отправления и прибытия, а также аэропорты.

Чтобы один самолет мог использоваться для сегмента i, а затем позднее для сегмента j, если:

(a) аэропорт прибытия i совпадает с аэропортом отправления j, и времени между рейсами достаточно для технического обслуживания самолета; или

(b) между рейсами можно добавить сегмент, который переводит самолет из аэропорта прибытия i в аэропорт отправления j с достаточным промежуточным временем.

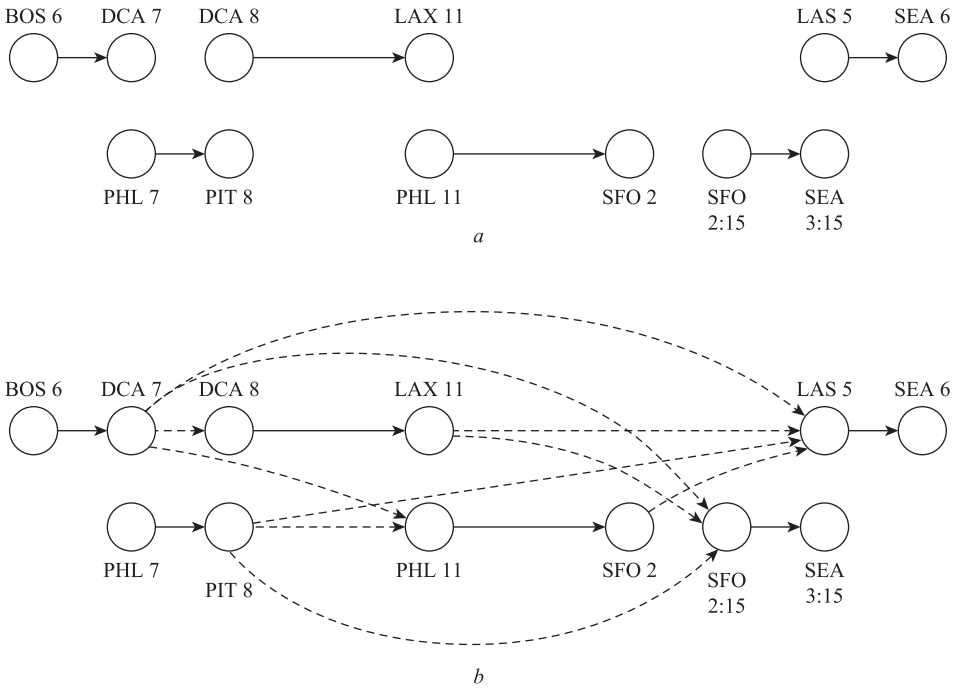


Рис. 7.17. *a* — простой экземпляр задачи планирования авиаперелетов; *b* — расширенный граф, показывающий возможности перехода между рейсами

Например, если время технического обслуживания составляет один час, один самолет может использоваться для полетов (1), (3) и (6), чтобы самолет находился в Вашингтоне между рейсами (1) и (3), а затем вставить перелет между рейсами (3) и (6).

Формулировка задачи

Ситуацию можно смоделировать на очень общем уровне, абстрагируясь от конкретных правил о времени технического обслуживания и промежуточных сегментов: мы просто говорим, что рейс *j доступен* из рейса *i*, если самолет, использовавшийся для рейса *i*, также может использоваться для полета *j*. Итак, по приведенным выше правилам (a) и (b) для каждой пары *i*,*j* можно легко определить, доступен ли рейс *j* из рейса *i*. (Конечно, правила доступности могут быть намного более сложными. Например, время технического обслуживания в случае (a) может зависеть от аэропорта, или случай (b) может потребовать, чтобы вставленный сегмент обеспечивал необходимую прибыль.) Здесь важно то, что наше определение позволяет обработать любой набор правил: входные данные задачи включают не только множество сегментов, но и спецификацию пар (i, j), для которых более поздний рейс *j* достижим из более раннего рейса *i*. Такие пары могут образовать произвольный ациклический граф.

В этой задаче требуется определить, возможно ли организовать все m рейсов из исходного списка с использованием не более k самолетов. Для этого необходимо найти способ эффективного повторного использования самолетов в разных рейсах. Например, вернемся к экземпляру на рис. 7.17 и предположим, что самолетный парк состоит из $k = 2$ самолетов. Если использовать один самолет для рейсов (1), (3) и (6), как предполагалось выше, другой самолет не сможет обслужить все остальные полеты (2), (4) и (5) (ему не хватит времени на техническое обслуживание в Сан-Франциско между рейсами (4) и (5)). Однако обслужить все шесть рейсов с двумя самолетами все же возможно, если воспользоваться другим решением: один самолет обслуживает рейсы (1), (3) и (5) (с включением промежуточного рейса LAX–SFO), а другой обслуживает рейсы (2), (4) и (6) (с включением промежуточных рейсов PIT–PHL и SFO–LAS).

Разработка алгоритма

В этом разделе рассматривается эффективный алгоритм, позволяющий решать произвольные экземпляры задачи планирования авиаперелетов на основании потока в сети. Вы увидите, что потоковые методы чрезвычайно естественно адаптируются к этой задаче.

Решение основано на следующей идее: единицы потока соответствуют самолетам. Для каждого рейса создается ребро, а верхняя и нижняя границы пропускной способности, равные 1, требуют, чтобы по ребру передавалась ровно одна единица потока. Другими словами, каждый рейс должен обслуживаться одним из самолетов. Если (u_i, v_i) — ребро, представляющее рейс i, а (u_j, v_j) — ребро, представляющее рейс j, и рейс j доступен из рейса i, то из v_i в u_j ведет ребро с пропускной способностью 1; при этом единица потока может переместиться по (u_i, v_i), а затем перейти прямо к (u_j, v_j). Такая конструкция ребер изображена на рис. 7.17, *b*.

Мы расширяем эту идею на поток в сети, включая в граф источник и сток; теперь можно рассмотреть всю конструкцию в подробностях. Множество узлов базового графа G определяется следующим образом:

- Для каждого рейса i граф содержит два узла u_i и v_i.
- G также содержит отдельный узел-источник s и узел-сток t.

Множество ребер G определяется следующим образом:

- Для всех i существует ребро (u_i, v_i) с нижней границей 1 и пропускной способностью 1. (Каждый рейс в списке должен обслуживаться.)
- Для всех i и j, для которых рейс j доступен от рейса i, существует ребро (v_i, u_j) с нижней границей 0 и пропускной способностью 1. (Один самолет может выполнять рейсы i и j.)
- Для всех i существует ребро (s, u_i) с нижней границей 0 и пропускной способностью 1. (Любой самолет может начать день с полета i.)
- Для всех j существует ребро (v_j, t) с нижней границей 0 и пропускной способностью 1. (Любой самолет может завершить день полетом j.)

- Существует ребро (s, t) с нижней границей 0 и пропускной способностью k. (Если имеются дополнительные самолеты, не обязательно использовать их в полетах.)

Наконец, узел s имеет уровень потребления $-k$, а узел t — уровень потребления k. У всех остальных узлов уровень потребления равен 0.

Наш алгоритм основан на построении сети G и поиска в ней действительной циркуляции. Докажем правильность этого алгоритма.

Анализ алгоритма

(7.54) Способ организации всех полетов с использованием не более k самолетов существует в том и только в том случае, если в сети G существует действительная циркуляция.

Доказательство. Сначала предположим, что все рейсы могут быть выполнены с использованием $k' \leq k$ самолетов.

Множество рейсов, выполняемых каждым отдельным самолетом, определяет путь P в сети G; по каждому такому пути P передается одна единица потока. Чтобы удовлетворить полные требования S и t, мы отправляем по ребру (s, t) $k - k'$ единиц потока. Полученная циркуляция удовлетворяет все ограничения по уровню потребления, пропускной способности и нижним границам.

И наоборот, рассмотрим действительную циркуляцию в сети G. Из (7.52) мы знаем, что в сети существует действительная циркуляция с целочисленными величинами потока. Предположим, что k' единиц потока передаются по ребрам, отличным от (s, t). Так как у каждого ребра пропускная способность ограничена значением 1, а циркуляция является целочисленной, каждое ребро, передающее поток, несет ровно одну единицу потока.

Преобразуем результат в расписание по схеме, похожей на приведенную в доказательстве (7.42), где поток преобразовывался в множество пар. На самом деле в данном случае ситуация проще, потому что граф не содержит циклов. Рассмотрим ребро (s, u_i), по которому передается одна единица потока. Из ограничения сохранения потока следует, что (u_i, v_i) передает одну единицу потока, и из v_i выходит единственное ребро, несущее одну единицу потока. Продолжая таким образом, мы строим путь P от s к t, в котором каждое ребро передает одну единицу потока. Процесс применяется к каждому ребру в форме (s, u_j), передающему одну единицу потока; таким образом будут получены k' путей от s к t, каждый из которых состоит из ребер, передающих одну единицу потока. С каждым путем P, построенным по этой схеме, связывается один самолет, который может выполнять все рейсы этого пути. ■

Расширения: моделирование других аспектов задачи

В реальной жизни планирование авиаперелетов поглощает бесчисленные часы процессорного времени. Впрочем, в самом начале мы упоминали о том, что наша

формулировка задачи является чисто учебной — в ней игнорируются некоторые очевидные факторы, которые должны учитываться в таких приложениях. Прежде всего в ней игнорируется тот факт, что любой самолет может отработать определенное количество часов, после чего временно выводится из эксплуатации для более серьезного технического обслуживания. Во-вторых, мы строим оптимальное расписание на один день (или по крайней мере на короткий промежуток времени) так, словно ни вчерашнего, ни завтрашнего дня не существует; однако в конце дня N самолеты должны быть оптимально расположены для начала дня N. В-третьих, все самолеты должны быть укомплектованы экипажами, и хотя экипажи тоже могут быть задействованы на нескольких рейсах, при этом действуют другие ограничения, так как люди и самолеты устают с разной скоростью. Причем все эти проблемы даже в первом приближении не учитывают тот факт, что обслуживание конкретного сегмента полета не является жестким ограничением; скорее, цель заключается в оптимизации прибыли, что позволяет выбрать несколько рейсов из многих возможных вариантов для достижения этой цели (не говоря уже об удобстве структуры перелетов для пассажиров).

Вероятно, из всего сказанного можно сказать следующее: потоковые методы приносят пользу при решении подобных задач, и они применяются на практике. Действительно, описанное решение представляет собой общий подход к организации эффективного повторного использования ограниченного набора ресурсов в разнообразных ситуациях. В то же время эффективная организация авиаперелетов в реальной жизни — исключительно сложная задача.

7.10. Сегментация изображений

Центральное место в компьютерной обработке изображений занимает *сегментация* изображения. Например, на изображении могут быть представлены три человека на сложном фоне. Естественная, но непростая задача заключается в идентификации каждого из трех людей как отдельного объекта.

Задача

Одна из основных задач этого направления — отделение переднего плана от фона: каждый пиксел изображения помечается как принадлежащий либо к переднему плану, либо к фону. Как выясняется, чрезвычайно естественная модель такого разбиения приводит к задаче, эффективно решаемой посредством вычисления минимального разреза.

Обозначим V множество *пикселов* анализируемого изображения. Некоторые пары пикселов объявляются *соседями*; пусть E — множество всех пар соседних пикселов. Таким образом будет получен ненаправленный граф $G = (V, E)$. Здесь мы намеренно не даем четкого определения того, что подразумевается под «пикселом» или «соседским» отношением. Для любого графа G будет получено эффективное решение задачи, поэтому мы можем определять эти концепции так, как считаем

нужным. Конечно, пикселы естественно представлять как точки, образующие сетку; при этом соседями считаются пикселы, занимающие смежные позиции в этой сетке, как показано на рис. 7.18, *a*.

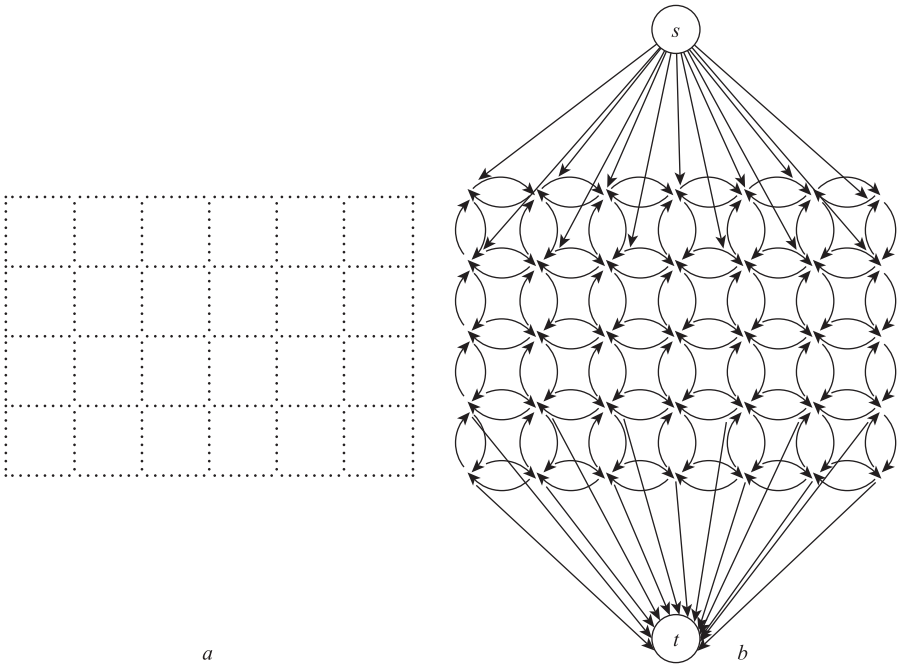


Рис. 7.18. *a* — матрица пикселов; *b* — схема соответствующего потокового графа; на схеме изображены не все ребра от источника к стоку

Для каждого пиксела i имеется степень *правдоподобия* a_i того, что он принадлежит переднему плану, и степень правдоподобия b_i того, что он принадлежит фону. Условно будем считать, что величины правдоподобия представляют собой произвольные неотрицательные числа, содержащиеся в постановке задачи, и они указывают, насколько желательно, чтобы пиксел i считался относящимся к переднему плану или фону. В остальном не так важно, какие физические свойства изображения оценивают эти характеристики и как они вычисляются.

Каждой отдельный пиксел i было бы разумно отнести к переднему плану, если $a_i > b_i$, или к фону в противном случае. Однако решение относительно i зависит от решений, принимаемых относительно соседей i. Например, если многие соседи i помечены как относящиеся к фону, то у нас появляется больше оснований отнести к фону i; это приводит к «сглаживанию» границ между фоном и передним планом, и снижению длины границы. Для каждой пары соседних пикселов (i, j) вводится *штраф за разделение* $p_{ij} \geq 0$, применяемый в том случае, если один пиксел пары относится к переднему плану, а другой к фону.

А теперь мы можем точно определить задачу сегментации в контексте параметров правдоподобия и штрафа за разделение. Требуется найти разбиение мно-

жества пикселов на подмножества A и B (передний план и фон соответственно), максимизирующее

$$q(A,B)=\sum_{i\in A}a_i+\sum_{j\in B}b_j-\sum_{\substack{(i,j)\in E\\ |A\cap\{i,j\}|=1}}p_{ij}.$$

Таким образом, высокие значения правдоподобия награждаются, а соседние пары (i, j), в которых один пиксел принадлежит A, а другой принадлежит B, штрафуются. Задача заключается в вычислении оптимальной разметки — разбиения (A, B), максимизирующей $q(A, B)$.

Разработка и анализ алгоритма

Нетрудно заметить сходство между задачей о минимальном разрезе и задачей нахождения оптимальной разметки. Тем не менее между этими задачами также существует ряд важных различий. Во-первых, целевая функция максимизируется, а не минимизируется. Во-вторых, в задаче разметки нет источника и стока; кроме того, нам приходится иметь дело со значениями a_i и b_i узлов. В-третьих, граф G является ненаправленным, тогда как в задаче о минимальном разрезе используется направленный граф. Все эти проблемы кратко рассмотрены ниже.

Первая проблема — тот факт, что задача сегментации направлена на максимизацию — решается следующим наблюдением. Пусть $Q=\sum_i(a_i+b_i)$. Сумма $\sum_{i\in A}a_i+\sum_{j\in B}b_j$ не отличается от суммы $Q-\sum_{i\in A}b_i+\sum_{j\in B}a_j$, что позволяет записать

$$q(A,B)=Q-\sum_{i\in A}b_i+\sum_{j\in B}a_j-\sum_{\substack{(i,j)\in E\\ |A\cap\{i,j\}|=1}}p_{ij}.$$

Таким образом, мы видим, что задача максимизации $q(A, B)$ эквивалентна задаче максимизации величины

$$q'(A,B)=\sum_{i\in A}b_i+\sum_{j\in B}a_j+\sum_{\substack{(i,j)\in E\\ |A\cap\{i,j\}|=1}}p_{ij}.$$

Что касается отсутствия источника и стока, мы поступим так же, как в других предшествующих построениях: создадим новый «супериcточник» s, представляющий передний план, и новый «суперсток», представляющий фон. Заодно это позволит разобраться со значениями a_i и b_i, находящимися в узлах (тогда как минимальные разрезы ограничиваются числами, связанными с ребрами). А именно, каждый из узлов s и t соединяется с каждым пикселом, а a_i и b_i используются для определения соответствующих пропускных способностей ребер между пикселом i и источником и стоком соответственно.

Наконец, чтобы разобраться с направленностью ребер, мы смоделируем каждую из соседних пар (i, j) двумя направленными ребрами (i, j) и (j, i), как было сделано в ненаправленной задаче о непересекающихся путях. Этот прием очень хорошо работает в данном случае, так как в любом разрезе s–t не более одного из двух противоположно направленных ребер может переходить со стороны s на сторону t разреза (потому что другое ребро в этом случае должно переходить со стороны t на сторону s).

Конкретнее, мы определяем следующую потоковую сеть $G' = (V', E')$, изображенную на рис. 7.18, *b*. Множество узлов V' состоит из множества пикселов V с двумя дополнительными узлами s и t. Для каждой соседней пары пикселов i и j добавляются направленные ребра (i, j) и (j, i), каждое из которых обладает пропускной способностью p_{ij}. Для каждого пиксела i добавляются ребро (s, i) с пропускной способностью a_i, и ребро (i, t) с пропускной способностью b_i.

Теперь разрез s–t (A, B) соответствует разбиению пикселов на множества A и B. Рассмотрим, как пропускная способность разреза $c(A, B)$ связана с величиной $q'(A, B)$, которую мы пытаемся минимизировать Ребра, пересекающие разрез (A, B), делятся на три естественные категории.

- Ребра (s, j), для которых $j \in B$; такое ребро вносит вклад a_j в пропускную способность разреза.
- Ребра (i, t), для которых $i \in A$; такое ребро вносит вклад b_i в пропускную способность разреза.
- Ребра (i, j), для которых $i \in A$ и $j \in B$; такое ребро вносит вклад p_{ij} в пропускную способность разреза.

На рис. 7.19 на примере с четырьмя пикселами показано, как выглядит каждый из трех видов ребер относительно разреза.

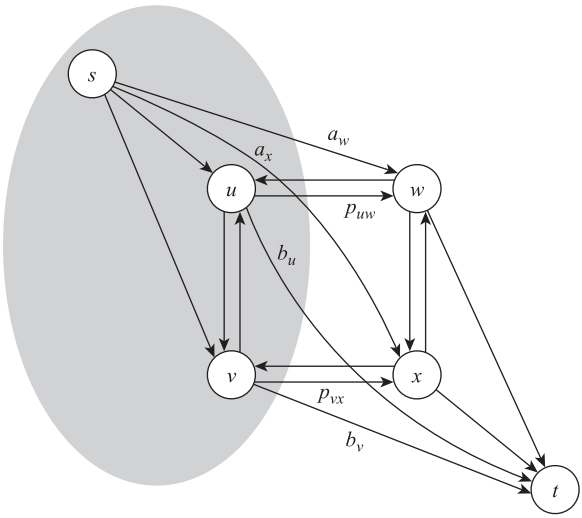


Рис. 7.19. Разрез s–t на графе, построенном для четырех пикселов. Обратите внимание на то, как в разрезе отражены три типа слагаемых в выражении $q'(A, B)$

Суммируя вклады этих трех типов ребер, получаем

$$c(A,B)=\sum_{i\in A}a_i b+\sum_{j\in B}a_j-\sum_{\substack{(i,j)\in E\\ |A\cap\{i,j\}|=1}}p_{ij}=q'(A,B).$$

Итак, все идеально складывается. Потоковая сеть устроена так, что пропускная способность разреза (A, B) в точности соответствует величине $q'(A, B)$: три типа ребер, пересекающих разрез (A, B), которые мы определили (ребра от источника, ребра к стоку и ребра, не связанные ни с источником, ни со стоком), соответствуют трем типам слагаемых в выражении $q'(A, B)$.

Следовательно, если мы хотим минимизировать $q'(A, B)$ (что, как было показано ранее, эквивалентно максимизации $q(A, B)$), достаточно найти разрез с минимальной пропускной способностью — а мы уже знаем, как эффективно решить эту задачу.

Таким образом, решение задачи о минимальном разрезе дает оптимальный алгоритм для решения задачи об отделении фона от переднего плана.

(7.55) Решение задачи сегментации может быть получено при помощи алгоритма нахождения минимального разреза в графе G', построенного ранее. Для минимального разреза (A', B') разбиение (A, B), полученное удалением s^* и t^*, максимизирует метрику сегментации $q(A, B)$.

7.11. Выбор проекта

Большим (и малым) компаниям постоянно приходится искать баланс между проектами, которые могут принести прибыль, и расходами, необходимыми для обеспечения этих проектов. Предположим, телекоммуникационный гигант CluNet анализирует плюсы и минусы проекта по обеспечению новой услуги высокоскоростного доступа к Интернету для рядовых потребителей. Маркетинговые исследования показывают, что новая услуга принесет неплохой доход, но ее запуск потребует немалых подготовительных затрат: наращивания пропускной способности оптоволоконного кабеля и приобретения скоростных маршрутизаторов последнего поколения.

Подобные решения особенно сложны из-за неочевидных взаимодействий между ними: возможно, сам по себе доход от новой услуги не компенсирует затрат на обновление маршрутизаторов; но с обновленными маршрутизаторами компания сможет предложить выгодный новый проект своим корпоративным клиентам, и этот дополнительный проект изменит баланс. Все эти взаимодействия вызывают цепную реакцию: корпоративный проект в действительности потребует новых расходов, но в свою очередь откроет путь к двум другим выгодным проектам — и т. д. В конечном итоге вопрос звучит так: какими проектами стоит заниматься, а от каких лучше отказаться? Это классическая задача сопоставления затрат с потенциальной выгодой.

Задача

Ниже описана предельно общая структура для моделирования подобных решений. Имеется множество P проектов, с каждым проектом $i \in P$ связывается доход p_i, который может быть как положительным, так и отрицательным. (Другими словами, каждая выгодная возможность и каждый дорогостоящий шаг по наращиванию инфраструктуры в приведенном выше примере будет рассматриваться как отдельный проект.) Реализация некоторых проектов является предусловием для других проектов; это обстоятельство моделируется направленным ациклическим графом $G = (P, E)$. Узлы G соответствуют проектам, а ребро (i, j) означает, что проект i может быть выбран только в том случае, если также будет выбран проект j. Учтите, что проект i может иметь несколько предусловий, и может быть много проектов, в предусловия которых входит j. Множество проектов $A \subseteq P$ называется *действительным*, если предусловие каждого проекта в A также принадлежит A: для каждого $i \in A$ и каждого ребра $(i, j) \in E$ также $j \in A$. Требования, представленные в этой форме, будут называться *ограничениями очередности*. Прибыль от множества проектов определяется по формуле

$$\text{profit}(A) = \sum_{i \in A} p_i .$$

Задача выбора проектов заключается в выборе действительного множества проектов с максимальной прибылью.

В начале 1960-х годов эта задача стала активно изучаться в литературе по анализу данных, где она получила название *задачи открытой добычи*. «Открытой добычей» называется способ добычи полезных ископаемых, при котором с поверхности берутся земляные блоки для извлечения содержащейся в них руды. Перед началом добычи вся площадь делится на множество P блоков, и производится оценка чистой стоимости p_i каждого блока (то есть цены руды за вычетом затрат на обработку для этого конкретного блока). Одни значения будут положительными, другие отрицательными. В полном множестве блоков действуют ограничения очередности, с которыми некоторые блоки могут извлекаться только после извлечения других блоков, находящихся над ними. Задача открытой добычи формулируется как определение самого выгодного множества извлекаемых блоков с учетом ограничений очередности. Эта формулировка задачи укладывается в структуру выбора проектов — каждый блок соответствует отдельному проекту.

Разработка алгоритма

Сейчас мы покажем, что задача выбора проектов решается посредством сведения к вычислению минимального разреза в расширенном графе G', который определяется по аналогии с графом, использовавшимся в разделе 7.10 для сегментации изображений. Идея заключается в построении G' на базе G таким образом, чтобы сторона источника у минимального разреза в G' соответствовала оптимальному множеству выбираемых проектов.

Чтобы построить граф G', мы добавим в граф G новый источник s и новый сток t, как показано на рис. 7.20. Для каждого узла $i \in P$ с $p_i > 0$ добавляется ребро (s, i) с пропускной способностью p_i. Для каждого узла $i \in P$ с $p_i < 0$ добавляется ребро (i, t) с пропускной способностью $-p_i$. Значения пропускных способностей ребер G мы зададим позднее. Однако уже сейчас видно, что пропускная способность разреза $(\{s\}, P \cup \{t\})$ равна $C = \sum_{i \in P; p_i > 0} p_i$, поэтому величина максимального потока в этой сети не превышает C.

Далее нужно гарантировать, что если (A', B') — минимальный разрез в этом графе, то $A = A' - \{s\}$ подчиняется ограничениям очередности; то есть если у узла $i \in A$ имеется ребро $(i, j) \in E$, то должно выполняться условие $j \in A$. Пожалуй, самым концептуально «чистым» способом будет назначение ребрам G пропускной способности ∞. Ранее понятие бесконечной пропускной способности не формализовалось, но сделать это нетрудно: оно означает лишь то, что такое ребро вообще не имеет верхней границы. Алгоритмы предшествующих разделов, а также теорема о максимальном потоке и минимальном разрезе переносятся для случая бесконечной емкости. Впрочем, введения понятия бесконечной емкости также можно избежать, присвоив каждому из этих ребер «фактически бесконечную» пропускную способность. В нашем контексте для этого достаточно назначить каждому такому ребру пропускную способность $C + 1$: величина максимального потока в G' не превышает C, поэтому никакой минимальный разрез не может содержать ребро, пропускная способность которого превышает C. Следующее описание работает при любом из этих двух вариантов.

Теперь можно изложить алгоритм: мы вычисляем минимальный разрез (A', B') в G' и объявляем $A' - \{s\}$ оптимальным множеством проектов. Переходим к доказательству того, что этот алгоритм действительно предоставляет оптимальное решение.

Анализ алгоритма

Начнем с рассмотрения множества проектов A, удовлетворяющего ограничениям очередности. Пусть $A' = A \cup \{s\}$ и $B' = (P - A) \cup \{t\}$; рассмотрим разрез s–t (A', B'). Если множество A удовлетворяет ограничениям очередности, то ни одно ребро $(i, j) \in E$ не пересекает этот разрез (рис. 7.20). Пропускная способность разреза выражается следующим образом:

(7.56) Пропускная способность разреза (A', B'), определяемая для множества проектов A с учетом ограничений очередности, равна

$$c(A', B') = C - \sum_{i \in A} p_i.$$

Доказательство. Ребра G' можно разделить на три категории: соответствующие множеству ребер E графа G, выходящие из источника s и входящие в сток t. Так как A удовлетворяет ограничениям очередности, ребра E не пересекают разрез (A', B'), а следовательно, не влияют на его пропускную способность. Ребра, входящие в сток t, вносят в пропускную способность разреза вклад

$$\sum_{i \in A \text{ and } p_i < 0} -p_i.$$

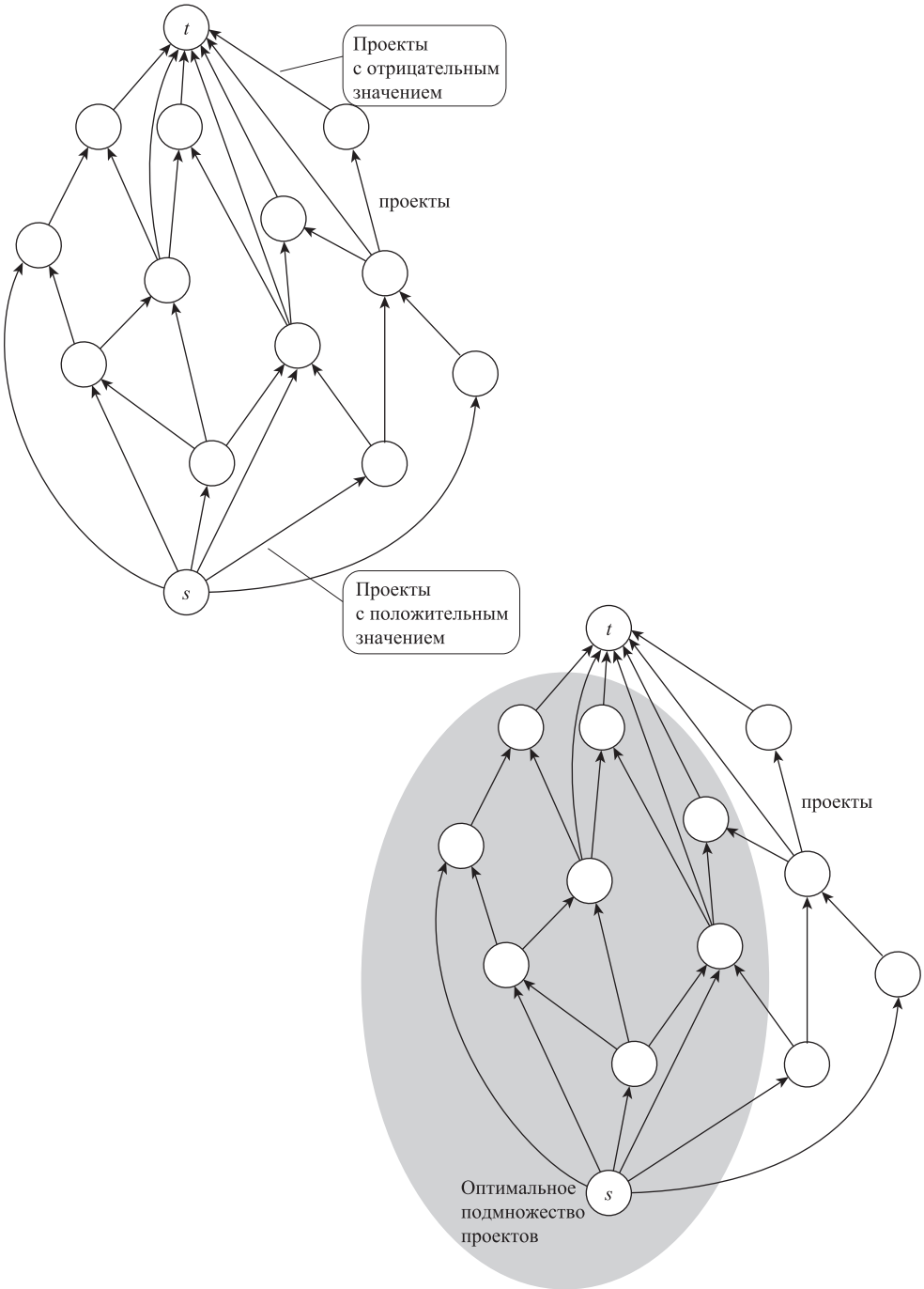


Рис. 7.20. Потоковый граф, используемый для решения задачи выбора проектов. Справа изображен возможный разрез с минимальной пропускной способностью

Вклад ребер, выходящих из источника s, равен

$$\sum_{i \notin A \text{ and } p_i > 0} p_i.$$

По определению C последнюю величину можно переписать в виде $C - \sum_{i \in A \text{ and } p_i > 0} p_i$. Пропускная способность разреза (A', B') равна сумме этих двух слагаемых, то есть

$$\sum_{i \in A \text{ and } p_i < 0} (-p_i) + \left(C - \sum_{i \in A \text{ and } p_i > 0} p_i \right) = C - \sum_{i \in A} p_i,$$

как и утверждалось в постановке задачи. ■

Вспомните, что пропускная способность ребер G превышает $C = \sum_{i \in P: p_i > 0} p_i$, а следовательно, такие ребра не могут пересекать разрез, пропускная способность которого не превышает C. Из этого следует, что такие разрезы определяют действительные множества проектов.

(7.57) Если (A', B') — разрез с пропускной способностью, не превышающей C, то множество $A = A' - \{s\}$ удовлетворяет ограничениям очередности.

Теперь мы можем доказать главную цель всего построения — что минимальный разрез в G' определяет оптимальное множество проектов. Объединяя эти два утверждения, мы видим, что разрезы (A', B'), пропускная способность которых не превышает C, однозначно соответствуют действительным множествам проектов $A = A' - \{s\}$. Пропускная способность такого разреза (A', B') составляет

$$c(A', B') = C - \text{profit}(A).$$

Величина пропускной способности C является константой, не зависящей от разреза (A', B'), поэтому разрез с минимальной пропускной способностью соответствует множествам проектов A с максимальной выгодой *profit*. Следовательно, нам удалось доказать следующее утверждение.

(7.58) Если (A', B') представляет собой минимальный разрез в G', то множество $A = A' - \{s\}$ является оптимальным решением задачи выбора проектов.

7.12. Выбывание в бейсболе

Продюсер в комментаторской говорит: «Видели на той неделе в газете про Эйнштейна?... Один репортер попросил его рассчитать результаты “гонки за флагом”. Знаете, одна команда выигрывает столько-то из оставшихся игр, остальные команды выигрывают столько или столько... Какие возможны варианты? У кого преимущество?

“А он в этом разбирается?”

“Похоже, не очень. Он решил, что «Доджерс» выиграют у «Гигантов» в пятницу”».

— Дон Делилло, «Изнанка мира»

Задача

Допустим, вы — репортер из «Алгоритмических спортивных новостей», и к концу сезона возникает следующая ситуация. Четыре бейсбольные команды пытаются занять первое место в Восточной подгруппе Американской лиги; назовем их «Нью-Йорк», «Балтимор», «Торонто» и «Бостон». На данный момент команды имеют следующее количество побед:

Нью-Йорк: 92, Балтимор: 91, Торонто: 91, Бостон: 90.

В сезоне осталось пять игр: все возможные пары из перечисленных команд, кроме Нью-Йорка и Бостона.

Вопрос заключается в следующем: сможет ли Бостон набрать побед не меньше, чем любая другая команда в подгруппе (то есть закончить на первом месте — возможно, разделив его с другой командой?)

Если немного подумать, становится очевидно, что ответ будет отрицательным. Один из аргументов выглядит так: очевидно, Бостон должен выиграть все оставшиеся игры, а Нью-Йорк должен проиграть обе оставшиеся игры. Но это означает, что и Балтимор, и Торонто выиграют у Нью-Йорка; следовательно, победитель игры Балтимор-Торонто будет иметь больше побед.

Следующий аргумент позволяет избежать подобного анализа. Бостон может набрать не более 92 побед. Совместно три другие команды имеют 274 победы, и в трех играх друг с другом будет ровно три победы, так что итоговая сумма составит 277. Но 277 побед для трех команд означают, что одна из них будет иметь более 92 победы.

Естественно задать некоторые вопросы: (i) Существует ли эффективный алгоритм для определения того, лишилась ли команда шансов на первое место? (ii) Когда команда лишается шансов на первое место, существует ли «усредняющий» алгоритм наподобие описанного выше, который доказывает это?

Перейдем к более конкретной формулировке: имеется множество S команд, для каждой команды $x \in S$ текущее количество побед равно w_x. Кроме того, две команды $x,y \in S$ еще должны сыграть g_{xy} партий друг с другом. Наконец, задана конкретная команда z.

Мы воспользуемся методами максимального потока для двух целей. Во-первых, будет представлен эффективный алгоритм для принятия решения о том, выбыла ли команда z из соревнования за первое место — другими словами, возможно ли выбрать результаты остальных игр так, чтобы команда z имела побед не меньше, чем любая другая команда в S. Во-вторых, будет доказана теорема об однозначной характеристике выбывания (то есть что всегда можно привести короткое «доказательство» того, что команда потеряла шансы на первое место).

(7.59) Допустим, команда z действительно выбыла из борьбы за первое место. Тогда существует «доказательство» этого факта в следующей форме:

- z может закончить игры с максимум m победами.
- Существует подмножество команд $T \subseteq S$, для которого

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{xy} > m|T|.$$

(А следовательно, одна из команд T неизбежно получит строго больше m побед.)

Следующий пример дает более сложную иллюстрацию того, как работает «усредняющий» аргумент (7.59). Предположим, остались те же четыре команды, но с другими количествами текущих побед:

Нью-Йорк: 90, Балтимор: 88, Торонто: 87, Бостон: 79.

У Бостона еще остаются четыре игры против каждой из трех остальных команд. У Балтимора остается еще по одной игре против Нью-Йорка и Торонто. И наконец, Нью-Йорку и Торонто осталось сыграть еще шесть игр друг с другом. Понятно, что для Бостона ситуация выглядит мрачно, но действительно ли он выбыл из борьбы за первое место?

Да, у Бостона действительно не осталось шансов на первое место. Чтобы убедиться в этом, заметьте, что у Бостона может быть не более 91 победы; а теперь рассмотрим множество команд T = {Нью-Йорк, Торонто}. Нью-Йорк и Торонто вместе уже имеют 177 побед; в шести оставшихся играх общее количество побед достигнет 183, а $183/2 > 91$. Это означает, что у одной команды неизбежно будет более 91 победы, поэтому Бостон не сможет занять первое место. Любопытно, что в данном экземпляре задачи для множества всех трех команд, опережающих Бостон, такое доказательство не подходит: все три команды имеют в сумме 265 побед, и между ними осталось еще 8 игр; в сумме получается 273, а $273/3 = 91$ — этого недостаточно, чтобы доказать, что Бостон не сможет разделить первое место с другими командами. Следовательно, для усредняющего аргумента должно быть выбрано множество T, которое состоит из Нью-Йорка и Торонто и не включает Балтимор.

Разработка и анализ алгоритма

Начнем с построения потоковой сети, предоставляющей эффективный алгоритм для проверки факта выбывания команды z. Затем, анализируя минимальный разрез в сети, мы докажем (7.59).

Очевидно, если для команды z существует вариант, приводящий ее к первому месту, следует рассмотреть вариант, в котором z выигрывает все оставшиеся игры. Допустим, при этом команда выходит с m победами. Теперь нужно тщательно распределить победы во всех остальных играх, чтобы ни одна команда не набрала более m побед. Такое распределение может быть получено посредством вычисления максимального потока на базе следующей идеи: имеется источник s, который генерирует все победы. Победа i может пройти через одну из двух команд, участвующих в i-й игре. Затем устанавливается ограничение пропускной способности, которое говорит, что через команду x могут проходить не более $m - w_x$.

Конкретнее строится потоковая сеть G, изображенная на рис. 7.21. Пусть $S' = S - \{z\}$ и $g^* = \sum_{x,y \in S'} g_{xy}$ — общее количество игр, оставшихся между всеми парами команд в S'. В сеть включаются узлы s и t, узел v_x для каждой команды $x \in S'$, и узел u_{xy} для каждой пары команд $x,y \in S'$ с ненулевым количеством игр, которые им осталось сыграть друг с другом. Ребра делятся на следующие категории:

- ребра (s, u_{xy}) (победы выходят из s);
- ребра (u_{xy}, v_x) и (u_{xy}, v_y) (только x или y может победить в игре, в которой они играют друг с другом);
- ребра (v_x, t) (победы поглощаются в t).

Какие же пропускные способности следует назначить этим ребрам? От s к u_{xy} должны передаваться g_{xy} побед с насыщением, поэтому ребру (s, u_{xy}) назначается пропускная способность g_{xy}. Команда x не должна выиграть более $m–w_x$ игр, поэтому ребру (v_x, t) назначается пропускная способность $m–w_x$. Наконец, ребро в форме (u_{xy}, v_y) должно иметь как минимум g_{xy} единиц пропускной способности, чтобы оно могло передать все победы от u_{xy} к v_x; на самом деле самое «чистое» решение — назначить ребру бесконечную пропускную способность. (Построение будет работать даже в том случае, если ребру назначить только g_{xy} единиц пропускной способности, но доказательство (7.59) при этом усложнится.)

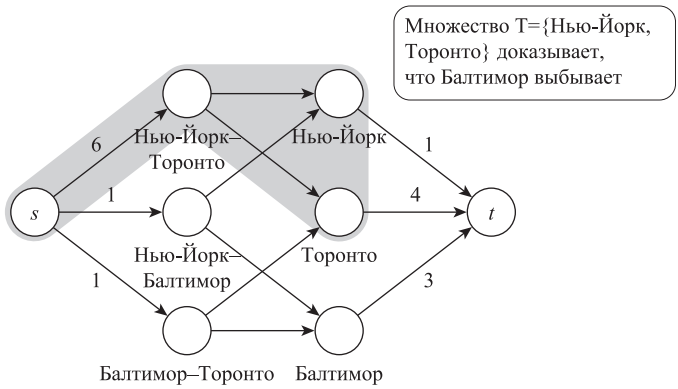


Рис. 7.21. Множество T={Нью-Йорк, Торонто} доказывает, что Балтимор выбывает

Если существует поток с величиной g^*, то результаты всех оставшихся игр могут создать ситуацию, при которой ни одна команда не имеет более m побед; а следовательно, если команда z выиграет все свои оставшиеся игры, она все равно может добиться хотя бы ничьей за первое место. И наоборот, если для оставшихся игр существуют результаты, с которыми z получает по крайней мере ничью, эти результаты могут использоваться для определения потока с величиной g^*. Например, на рис. 7.21, основанном на втором примере, обозначенный разрез показывает, что максимальный поток имеет величину не более 7, тогда как $g^* = 6 + 1 + 1 = 8$.

Итак, мы показали следующий результат:

(7.60) Команда z выбывает из борьбы за первое место в том и только в том случае, если величина максимального потока в G строго меньше g^*. Следовательно, проверка выбывания z может быть выполнена за полиномиальное время.

Характеристика выбывания команды

Представленная схема построения сетевого потока также может использоваться для доказательства (7.59). Идея заключается в том, что теорема о максимальном потоке и минимальном разрезе дает удобную характеристику вида «в том и только в том случае» для существования потока; интерпретируя эту характеристику в контексте приложения, мы получим столь же удобную характеристику. Тем самым демонстрируется общий подход к построению теорем характеризации для задач, сводимых к потоку в сети.

Доказательство (7.59). Предположим, команда z выбыла из борьбы за первое место. В этом случае максимальный поток s–t в G имеет величину $g' < g^*$; следовательно, существует разрез s–t (A, B) с пропускной способностью g', и (A, B) является минимальным разрезом. Пусть T — множество команд x, для которых $v_x \in A$. Теперь докажем, что T может использоваться как «усредняющий аргумент» в (7.59).

Сначала рассмотрим узел u_{xy} и предположим, что x или y не входит в T, но $u_{xy} \in A$. Тогда ребро (u_{xy}, v_x) переходит из A в B, а следовательно, разрез (A, B) обладает бесконечной пропускной способностью. Это противоречит предположению о том, что (A, B) является минимальным разрезом с пропускной способностью менее g^*. Итак, если одна из команд x или y не входит в T, то $u_{xy} \in B$. С другой стороны, предположим, что x и y принадлежат T, но $u_{xy} \in B$. Рассмотрим разрез (A', B'), который был бы получен добавлением u_{xy} в множество A и удалением его из множества B. Пропускная способность (A', B') просто равна пропускной способности (A, B) за вычетом пропускной способности g_{xy} ребра (s, u_{xy}) — потому что ребро (s, u_{xy}) прежде переходило от A к B, а теперь не переходит от A' к B'. Но так как $g_{xy} > 0$, это означает, (A', B') обладает меньшей пропускной способностью, чем (A, B), что снова противоречит предположению о минимальности разреза (A, B). Итак, если x и y принадлежат T, то обе команды $u_{xy} \in A$.

Таким образом, мы пришли к следующему заключению на основании того факта, что (A, B) является минимальным разрезом: $u_{xy} \in A$ в том и только в том случае, если обе команды $x, y \in T$.

А теперь остается лишь найти пропускную способность минимального разреза $c(A, B)$ в контексте пропускных способностей составляющих ребер. Из вывода предыдущего абзаца мы знаем, что ребра, переходящие из A в B, должны иметь одну из следующих двух форм:

- ребра в форме (v_x, t), где $x \in T$, и
- ребра в форме (s, u_{xy}), где по крайней мере один из узлов x или y не принадлежит T (другими словами, $\{x, y\} \not\subset T$).

Следовательно, имеем

$$c(A,B) = \sum_{x \in T}(m - w_x) + \sum_{\{x,y\} \not\subset T} g_{xy} = m|T| - \sum_{x \in T} w_x + (g^* - \sum_{x,y \in T} g_{xy}).$$

Так как мы знаем, что $c(A,B) = g' < g^*$, из последнего неравенства вытекает

$$m|T| - \sum_{x \in T} w_x - \sum_{x,y \in T} g_{xy} < 0,$$

а следовательно,

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{xy} > m|T|. \blacksquare$$

Например, применяя аргумент из доказательства (7.59) к экземпляру на рис. 7.21, мы видим, что узлы Нью-Йорка и Торонто расположены на стороне источника относительно минимального разреза, и как было показано ранее, эти две команды действительно доказывают, что Бостон выбыл из борьбы за первое место.

7.13.* Добавление стоимостей в задачу паросочетаний

Вернемся к первой задаче, упомянутой в этой главе, — двудольным паросочетаниям. Идеальные паросочетания в двудольном графе предоставили механизм моделирования парных отношений между разными объектами — заданий с машинами, например. Но во многих ситуациях в одном множестве объектов существует много возможных идеальных паросочетаний, и нам хотелось бы как-то выразить идею о том, что некоторые идеальные паросочетания могут быть «лучше» других.

Задача

Естественная формулировка задачи такого рода заключается во введении *стоимостей*. Например, выполнение некоторого задания на некоторой машине может требовать определенных затрат, или же нам хотелось бы связать задания с машинами так, чтобы свести к минимуму общую стоимость. Или представьте себе n пожарных машин, которые необходимо отправить в n разных домов; каждый дом находится на заданном расстоянии от пожарной части, и распределение должно минимизировать среднее расстояние перемещения каждой машины до точки назначения. Короче, алгоритм для поиска идеального паросочетания с *минимальной общей стоимостью* был бы чрезвычайно полезен.

Формально мы рассматриваем двудольный граф $G = (V, E)$, множество узлов которого, как обычно, разбито на подмножества $V = X \cup Y$ так, что один конец каждого ребра $e \in E$ принадлежит X, а другой конец принадлежит Y. Кроме того, с каждым ребром e связана неотрицательная стоимость c_e. Для паросочетания M стоимостью паросочетания называется общая стоимость всех ребер, входящих в M, то есть $cost(M) = \sum_{e \in M} c_e$. Задача нахождения идеального паросочетания с минимальной стоимостью предполагает, что $|X| = |Y| = n$, и целью является поиск идеального паросочетания с минимальной стоимостью.

Разработка и анализ алгоритма

В этом разделе описан эффективный алгоритм для решения этой задачи, основанный на идее увеличивающих путей, но адаптированный с учетом стоимостей. Таким образом, алгоритм итеративно строит паросочетания с использованием i ребер, для каждого значения i от 1 до n. Мы покажем, что при завершении алгоритма с паросочетанием размера n получается идеальное паросочетание с минимальной стоимостью. Высокоуровневая структура алгоритма проста: если имеется паросочетание с минимальной стоимостью с размером i, то мы ищем увеличивающий путь для получения паросочетания с размером i+1; и вместо любого увеличивающего пути (достаточного при отсутствии ребер) используется увеличивающий путь с минимальной стоимостью, чтобы самое большое паросочетание тоже имело минимальную стоимость.

Вспомните процедуру построения остаточного графа, использовавшегося для поиска увеличивающих путей. Имеется паросочетание M; в граф добавляются два новых узла s и t. Мы добавляем ребра (s, x) для всех непарных узлов $x \in X$ и ребра (y, t) для всех непарных узлов $y \in Y$. Ребро $e = (x, y) \in E$ ориентировано от x к y, если e не входит в паросочетание M, и от y к x, если $e \in M$. Этот остаточный граф будет обозначаться G_M. Обратите внимание: все ребра, проходящие из Y в X, входят в паросочетание M, тогда как ребра, проходящие из X в Y, в него не входят. Любой направленный путь s–t P в графе G_M соответствует паросочетанию, размер которого на 1 больше, чем у M, полученному заменой ребер из P — то есть ребра P, проходящие из X в Y, добавляются в M, а все ребра в P, проходящие из Y в X, удаляются из M. Как и прежде, путь P в G_M будет называться *увеличивающим путем*; соответственно паросочетание M *увеличивается* с использованием пути P.

Полученное паросочетание должно иметь как можно меньшую стоимость. Для этого мы будем искать увеличивающий путь, стоимость которого мала относительно следующих естественных стоимостей: ребра, выходящие из s и входящие в t, имеют стоимость 0; ребро, ориентированное от X к Y, имеет стоимость c_e (так как включение этого ребра в путь означает добавление ребра в M); и ребро e, ориентированное от Y к X, имеет стоимость $-c_e$ (так как включение этого ребра в путь означает удаление ребра из M). Стоимость пути P в G_M будет обозначаться $cost(P)$. Следующее утверждение кратко резюмирует суть построения.

(7.61) Пусть M — паросочетание, а P — путь в G_M от s к t; паросочетание M' получено из M улучшением относительно P. Тогда $|M'| = |M| + 1$, а $cost(M') = cost(M) + cost(P)$.

Приведенное утверждение подсказывает естественный алгоритм для нахождения идеального паросочетания с минимальной стоимостью: нужно проводить итеративный поиск путей с минимальной стоимостью в G_M и использовать пути для улучшения паросочетаний. Но как убедиться в том, что найденное идеальное паросочетание имеет минимальную стоимость? Или того хуже — что этот алгоритм вообще имеет смысл? Нахождение минимальных путей возможно только в том случае, если граф G_M гарантированно не содержит отрицательных циклов.

Анализ отрицательных циклов

В действительности понимание роли отрицательных циклов в G_M играет ключевую роль при анализе алгоритма. Сначала рассмотрим случай, в котором M является идеальным паросочетанием. Обратите внимание: в этом случае узел s не имеет выходящих ребер, а t не имеет входящих ребер в G_M (поскольку паросочетание является идеальным), а следовательно, никакой цикл в G_M не может содержать s или t.

(7.62) Имеется идеальное паросочетание M. Если в G_M существует направленный цикл C с отрицательной стоимостью, то стоимость M не минимальна.

Доказательство. Воспользуемся циклом C для увеличения — подобно тому, как направленные пути использовались для получения паросочетаний с большим размером. Увеличение M относительно C подразумевает добавление (и удаление) ребер из C в M. Полученное новое идеальное паросочетание M' имеет стоимость $cost(M') = cost(M) + cost(C)$; однако $cost(C) < 0$, а следовательно, стоимость M не минимальна. ■

Что еще важнее, обратное утверждение также истинно; таким образом, идеальное паросочетание M имеет минимальную стоимость тогда и только тогда, когда в G_M нет отрицательных циклов.

(7.63) Пусть M — идеальное паросочетание. Если в G_M нет направленных циклов с отрицательной стоимости C, то M является идеальным паросочетанием с минимальной стоимостью.

Доказательство. Предположим, утверждение ложно, и существует идеальное паросочетание M' с меньшей стоимостью. Рассмотрим множество ребер, входящих в одно из паросочетаний M и M' (но не в оба сразу). Заметим, что такое множество ребер соответствует множеству направленных циклов в G_M, непересекающихся по узлам. Стоимость множества направленных циклов равна в точности $cost(M') - cost(M)$. Если M' имеет меньшую стоимость, чем M, значит, по крайней мере один из таких циклов имеет отрицательную стоимость. ■

Итак, план заключается в переборе паросочетаний увеличивающегося размера, при котором граф G_M не содержит отрицательных циклов ни на одной итерации. В этом случае вычисление пути с минимальной стоимостью всегда будет однозначно определено; а при завершении с получением идеального паросочетания из (7.63) будет следовать, что паросочетание имеет минимальную стоимость.

Цены и узлы

Будет полезно представить, что с каждым узлом v связана числовая *цена* $p(v)$. Цены помогут не только лучше понять логику работы алгоритма, но и ускорить его реализацию. В частности, нам нужно следить за тем, чтобы граф G_M не содержат отрицательных циклов ни при какой итерации. Как узнать, что после увеличения в новом остаточном графе по-прежнему нет отрицательных циклов? Оказывается, компактное доказательство этого факта можно получить при помощи цен.

Чтобы понять, как работают цены в данном случае, полезно представить их экономическую интерпретацию. Рассмотрим следующий сценарий: допустим,

множество X представляет людей, которые назначаются на работы из множества Y. Для ребра $e = (x,y)$ стоимость c_e представляет затраты на выполнение человеком x работы y. Цену $p(x)$ можно сравнить с премией, которая выплачивается человеку x за участие в этой системе — своего рода «поощрительная премия». С учетом этого обстоятельства затраты на назначение человека x на работу y возрастают до $p(x) + c_e$. С другой стороны, цена $p(y)$ для узлов $y \in Y$ может рассматриваться как выгода от выполнения работы y (кто бы из работников X ее ни выполнял). В этом случае «чистые затраты» на назначение человека x на работу y принимают вид $p(x) + c_e - p(y)$: стоимость найма работника x с премией $p(x)$, выполнение им работы y с затратами c_e, с получением выгоды $p(y)$. Назовем эту величину *сокращенной стоимостью* ребра $e = (x, y)$, и обозначим ее $c^p{}_e = p(x) + c_e - p(y)$. Однако важно помнить, что в описание задачи входят только стоимости c_e; цены (премии и выгоды) всего лишь помогают анализировать решение.

Набор чисел $\{p(v) : v \in V\}$ образует множество *совместимых цен* по отношению к паросочетанию M, если:

(i) для всех непарных узлов $x \in X\, p(x) = 0$ (если человеку не предложена работа, то и платить ему не нужно);

(ii) для всех ребер $e = (x, y)\, p(x) + c_e \geq p(y)$ (каждое ребро имеет неотрицательную сокращенную стоимость); и

(iii) для всех ребер $e = (x, y) \in M\, p(x) + c_e = p(y)$ (каждое ребро, используемое при назначении, имеет сокращенную стоимость 0).

Чем полезна такая система цен? На интуитивном уровне совместимые цены предполагают, что паросочетание имеет малую стоимость: по парным ребрам премия равна стоимости а на всех остальных ребрах премия не превышает стоимость. Для частичного паросочетания из этого может не следовать, что паросочетание имеет минимальную возможную стоимость для своего размера (в нем могут быть задействованы дорогостоящие работы). Однако мы утверждаем, что если M является произвольным паросочетанием, для которого существует множество совместимых цен, то G_M не содержит отрицательных циклов. Для идеального паросочетания M из этого будет следовать, что M имеет минимальную стоимость согласно (7.63).

Чтобы понять, почему G_M не может содержать отрицательные циклы, мы расширим определение сокращенной стоимости на ребра остаточного графа с использованием того же выражения $c^p{}_e = p(x) + c_e - p(y)$ для любого ребра $e = (v, w)$. Заметим, что из определения совместимых цен следует, что все ребра в остаточном графе G_M имеют неотрицательные сокращенные стоимости. Далее для любого цикла C

$$cost(C) = \sum_{e \in C} c_e = \sum_{e \in C} c^p_e,$$

так как все слагаемые в правой части, соответствущие ценам, компенсируются. Известно, что каждое слагаемое в правой части неотрицательно, поэтому очевидно, что значение $cost(C)$ также неотрицательно.

Существует и другая, алгоритмическая причина для назначения цен узлам. Если в графе имеются ребра с отрицательной стоимостью, но нет отрицательных

циклов, кратчайшие пути вычисляются по алгоритму Беллмана-Форда за время $O(mn)$. Но если в графе нет ребер с отрицательной стоимостью, вместо него можно воспользоваться алгоритмом Дейкстры со временем только $O(m \log n)$ — ускорение почти в n раз.

В нашем случае наличие цен позволяет вычислять кратчайшие пути для неотрицательных сокращенных стоимостей $c^p{}_e$, получая эквивалентный ответ. Предположим, мы используем алгоритм Дейкстры для нахождения минимальной стоимости $d_{p,M}(v)$ направленного пути от s к каждому узлу $v \in X \cup Y$ в соответствии со стоимостями $c^p{}_e$. С заданными минимальными стоимостями $d_{p,M}(y)$ для непарного узла $y \in Y$ (несокращенная) стоимость пути от s к t через y составляет $d_{p,M}(y) + p(y)$, а минимальная стоимость находится за дополнительное время $O(n)$. Подводя итог, приходим к следующему факту.

(7.64) Пусть M — паросочетание, а p — совместимые цены. Путь от s к t с минимальной стоимостью может быть найден за один проход алгоритма Дейкстры и дополнительное время $O(n)$.

Обновление цен узлов

Мы воспользовались ценами для улучшения одной итерации алгоритма. Чтобы подготовиться к следующей итерации, необходимо знать не только путь с минимальной стоимостью (для получения следующего паросочетания), но и способ получения совместимых цен в отношении нового паросочетания.

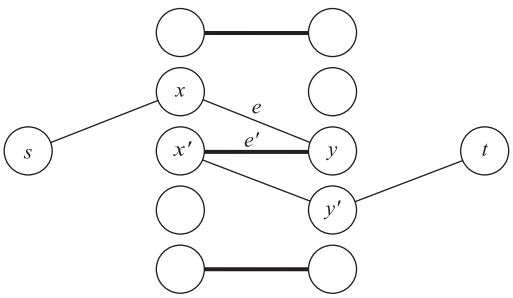


Рис. 7.22. Паросочетание M (темные ребра) и остаточный граф, используемый для увеличения размера паросочетания

Чтобы примерно представить, как это делается, рассмотрим непарный узел x в отношении паросочетания M и ребро $e = (x,y)$, показанное на рис. 7.22. Если новое паросочетание M' включает ребро e (то есть если e входит в увеличивающий путь, используемый для обновления паросочетания), то мы бы хотели, чтобы сокращенная стоимость этого ребра была равна 0. Однако цены p, использованные в паросочетании M, могут привести к сокращенной стоимости $c^p{}_e > 0$ — то есть назначение человека x на работу y в нашей экономической интерпретации не окупится. Нулевую сокращенную стоимость можно обеспечить либо повышением цены $p(y)$ (выгоды y) на $c^p{}_e$, либо уменьшением цены $p(x)$ на ту же величину. Чтобы цены оставались неотрицательными, мы увеличим $p(y)$. Однако узел y может быть

сопоставлен в паросочетании M с другим узлом x' по ребру $e' = (x', y)$, как показано на рис. 7.22. Увеличение выгоды $p(y)$ уменьшает сокращенную стоимость ребра e' до отрицательной, а следовательно, цены перестают быть совместимыми. Чтобы сохранить совместимость, можно увеличить $p(x')$ на ту же величину. Впрочем, такое изменение может создать проблемы с другими ребрами. Можно ли обновить все цены, сохранив совместимость паросочетания и цен по всем ребрам? Оказывается, эта задача легко решается использованием расстояний от s до всех остальных узлов, вычисленных алгоритмом Дейкстры.

(7.65) Пусть M — паросочетание, p — совместимые цены, а M' — паросочетание, полученное посредством увеличения по пути минимальной стоимости от s до t. Тогда $p'(v) = d_{p,M}(v) + p(v)$ является совместимым множеством цен для M'.

Доказательство. Чтобы доказать совместимость, сначала рассмотрим ребро $e = (x', y) \in M$. Единственным ребром, входящим в x', будет направленное ребро (y, x'), а следовательно, $d_{p,M}(x') = d_{p,M}(y) - c_e^p$, где $c_e^p = p(y) + c_e - p(x')$, и мы получаем искомое уравнение на всех ребрах. Затем рассмотрим ребра (x, y) в $M' - M$. Эти ребра расположены по пути минимальной стоимости от s до t, а следовательно, они удовлетворяют условию $d_{p,M}(y) = d_{p,M}(x) + c_e^p$, как и требовалось. Наконец, мы получаем необходимое неравенство для всех остальных ребер с учетом того факта, что все ребра $e = (x, y) \notin M$ должны удовлетворять условию $d_{p,M}(y) \le d_{p,M}(x) + c_e^p$. ■

Наконец, необходимо решить, как инициализировать алгоритм, чтобы начать его выполнение. Мы инициализируем M пустым множеством, определяем $p(x) = 0$ для всех $x \in X$ и определяем $p(y)$ для $y \in Y$ как минимальную стоимость ребра, входящего в y. Обратите внимание: эти цены совместимы в отношении $M = \varphi$.

Ниже приведено краткое описание алгоритма.

```
Инициализировать M пустым множеством
Определить p(x) = 0 для x ∈ X, и p(y) = mine into yc_e для y ∈ Y
Пока M не является идеальным паросочетанием
   Найти путь s-t P с минимальной стоимостью в G_M,
      используя (7.64) с ценами p
   Провести увеличение по P для получения нового паросочетания M'
   Найти множество совместимых цен в отношении M' согласно (7.65)
Конец Пока
```

Итоговое множество совместимых цен доказывает, что G_M не имеет отрицательных циклов; и с учетом (7.63) из этого следует, что M имеет минимальную стоимость.

(7.66) Идеальное паросочетание минимальной стоимости может быть найдено за время, необходимое для n вычислений кратчайшего пути с неотрицательной длиной ребер.

Расширения: экономическая интерпретация цен

В завершение нашего обсуждения идеального паросочетания минимальной стоимости мы немного разовьем экономическую интерпретацию цен. Рассмотрим

следующий сценарий: допустим, X — множество из n людей, каждый из которых желает купить дом, а Y — множество из n домов. Пусть $v(x, y)$ обозначает ценность дома y для покупателя x. Так как каждый покупатель хочет купить один из домов, можно предположить, что лучшим вариантом будет идеальное паросочетание M, максимизирующее $\sum_{(x,y)\in M} v(x, y)$. Такое идеальное паросочетание может быть найдено применением алгоритма идеального паросочетания минимальной стоимости со стоимостями $c_e = -v(x, y)$, если $e = (x, y)$.

А теперь зададимся следующим вопросом: можно ли убедить этих покупателей купить назначенный ему дом? Сам по себе каждый покупатель x хочет купить дом y, обладающий для него максимальным значением $v(x, y)$. Как убедить его вместо этого купить дом, выделенный нашим паросочетанием M? Мы будем использовать цены для стимуляции покупателей. Допустим, мы назначаем цену $P(y)$ для каждого дома y — то есть человек, покупающий дом y, должен заплатить $P(y)$. С учетом этих цен покупатель будет заинтересован в покупке дома с максимальной чистой стоимостью — то есть дом y, максимизирующий $v(x, y) - P(y)$. Мы говорим, что идеальное паросочетание M и цены на дома P *уравновешены*, если для всех ребер $(x, y) \in M$ и всех других домов y' выполняется

$$v(x, y) - P(y) \geq v(x, y') - P(y').$$

Но можно ли найти идеальное паросочетание и множество цен, достигающих такого состояния дел, при котором все покупатели остаются довольными? Оказывается, идеальное паросочетание с минимальной стоимостью и связанное с ним множество совместимых цен дает искомое.

(7.67) Пусть M — идеальное паросочетание минимальной стоимости, где $c_e = -v(x, y)$ для каждого ребра $e = (x, y)$, а p — совместимое множество цен. Тогда паросочетание M и множество цен $\{P(y) = -p(y) : y \in Y\}$ уравновешены.

Доказательство. Рассмотрим ребро $e = (x, y) \in M$; пусть $e' = (x, y')$. Так как M и p совместимы, имеем $p(x) + ce = p(y)$ и $p(x) + ce' \geq p(y')$. Вычитая эти два неравенства для исключения $p(x)$ и подставляя значения p и c, мы получаем нужное неравенство в определении уравновешенности. ■

Упражнения с решениями

Упражнение с решением 1

Имеется направленный граф $G = (V, E)$ с положительной целочисленной пропускной способностью c_e каждого ребра e, источником $s \in V$ и стоком $t \in V$. Также задан целочисленный максимальный поток s–t в G, определяемый величиной потока f_e по каждому ребру e.

Допустим, мы выбираем некоторое ребро $e \in E$ и увеличиваем его пропускную способность на одну единицу. Покажите, как найти максимальный поток в полученном графе за время $O(m + n)$, где m — количество ребер в G, а n — количество узлов.

Решение

Важно заметить, что времени $O(m + n)$ недостаточно для вычисления нового максимального потока «с нуля», поэтому нужно как-то использовать заданный поток f. Интуитивно понятно, что даже после увеличения пропускной способности ребра e на 1 поток f не может сильно отдалиться от максимума; в конце концов, сеть изменилась не так уж значительно.

На самом деле нетрудно показать, что максимальная величина потока может увеличиться не более чем на 1.

(7.68) Рассмотрим потоковую сеть G', полученную увеличением пропускной способности e на 1. Величина максимального потока в G' равна либо $v(f)$, либо $v(f) + 1$.

Доказательство. Величина максимального потока в G' равна минимум $v(f)$, так как f остается действительным потоком в сети. Кроме того, он еще и является целочисленным, поэтому достаточно показать, что величина максимального потока в G' не превышает $v(f) + 1$.

Согласно теореме о максимальном потоке и минимальном разрезе, в исходной потоковой сети G существует разрез s–t (A, B) с пропускной способностью $v(f)$. Зададимся вопросом: какова пропускная способность (A, B) в новой потоковой сети G'? Все ребра, пересекающие (A, B), имеют в G' ту же пропускную способность, что и G, с возможным исключением e (если e пересекает (A, B)). Но c_e возрастает только на 1, поэтому пропускная способность (A, B) в новой потоковой сети G' не превышает $v(f) + 1$. ■

Утверждение (7.68) предполагает естественный алгоритм. Начиная с действительного потока f в G', мы пытаемся найти один увеличивающий путь от s к t в остаточном графе G'_f. Поиск выполняется за время $O(m + n)$. Теперь возможен один из двух вариантов: может быть, найти увеличивающий путь не удастся, и тогда мы знаем, что f является максимальным потоком. В противном случае увеличение проходит успешно с поучением потока f' с величиной не менее $v(f) + 1$. В этом случае согласно (7.68) мы знаем, что f' является максимальным потоком. Итак, в любом случае после вычисления одного увеличивающего пути будет получен максимальный поток.

Упражнение с решением 2

Вы помогаете медицинской фирме «Врачи без выходных» составить рабочее расписание врачей в крупной больнице. Расписание на обычные дни в основном готово, но теперь нужно разобраться с особыми случаями — и в частности позаботиться о том, чтобы в праздники всегда был доступен хотя бы один врач.

Система работает так: определены k праздничных периодов (Рождественская неделя, выходные Дня независимости, выходные Дня благодарения, ...), каждый из которых занимает несколько смежных дней. Пусть D_j — множество дней, включенных в j-й праздничный период; объединение всех этих дней $\cup_j D_j$ будет называться множеством всех *праздничных дней*.

В больнице работают n врачей, за каждым врачом i закреплено множество праздничных дней S_i, когда он может выйти на работу. (В это множество может входить лишь часть дней из некоторых праздничных периодов; например, врач может работать в пятницу, субботу и воскресенье в период Дня благодарения, но не в четверг.)

Предложите алгоритм с полиномиальным временем, который получает эту информацию и определяет, возможно ли выбрать одного врача для работы в каждый праздничный день с учетом следующих ограничений.

- Для заданного параметра c каждый врач может быть назначен на работу не более чем в c праздничных дней, и только тогда, когда он доступен для выхода на работу.
- Для каждого праздничного периода j каждый врач может быть назначен на работу не более чем в один из дней множества D_j. (Например, хотя конкретный врач может работать в несколько праздничных дней в течение года, он не может работать более одного дня в выходные Дня благодарения, более одного дня в выходные Дня независимости и т. д.)

Алгоритм должен либо возвращать распределение, удовлетворяющее этим ограничениям, либо (обоснованно) сообщать о том, что такое распределение не существует.

Решение

Эта постановка задачи очень естественно подходит для применения алгоритмов сетевого потока, потому что на высоком уровне абстракции мы пытаемся сопоставить элементы одного множества (врачи) с элементами другого множества (праздничные дни). Сложности появляются с добавлением требования о том, что каждый врач может работать не более одного дня в каждый праздничный период.

Для начала посмотрим, как бы решалась эта задача без такого требования — в упрощенном случае, когда у каждого врача i имеется множество S_i дней, когда он может работать, и каждый врач должен работать не более c дней. Схема изображена на рис. 7.23(а). Узлы u_i представляют врачей, и связываются с узлами v_l, представляющими дни, когда они могут выйти на работу; ребро имеет пропускную способность 1. Суперисточник s связывается с каждым узлом врача u_i ребром с пропускной способностью c, и каждый узел дня v_l связывается с суперстоком t ребром, верхняя и нижняя границы которого равны 1. В этом случае назначенные дни «переходят» по врачам на дни, в которые те могут работать, а нижние границы ребер от дней до стока гарантируют, что каждый день будет обеспечен врачом. Наконец, предположим, что общее количество праздничных дней равно d; мы назначаем стоку уровень потребления $+d$, источнику — уровень потребления $-d$, а затем ищем действительную циркуляцию. (Вспомните, что после введения нижних границ для некоторых ребер алгоритмы формулируются в терминологии циркуляции с потреблением, а не максимального потока.)

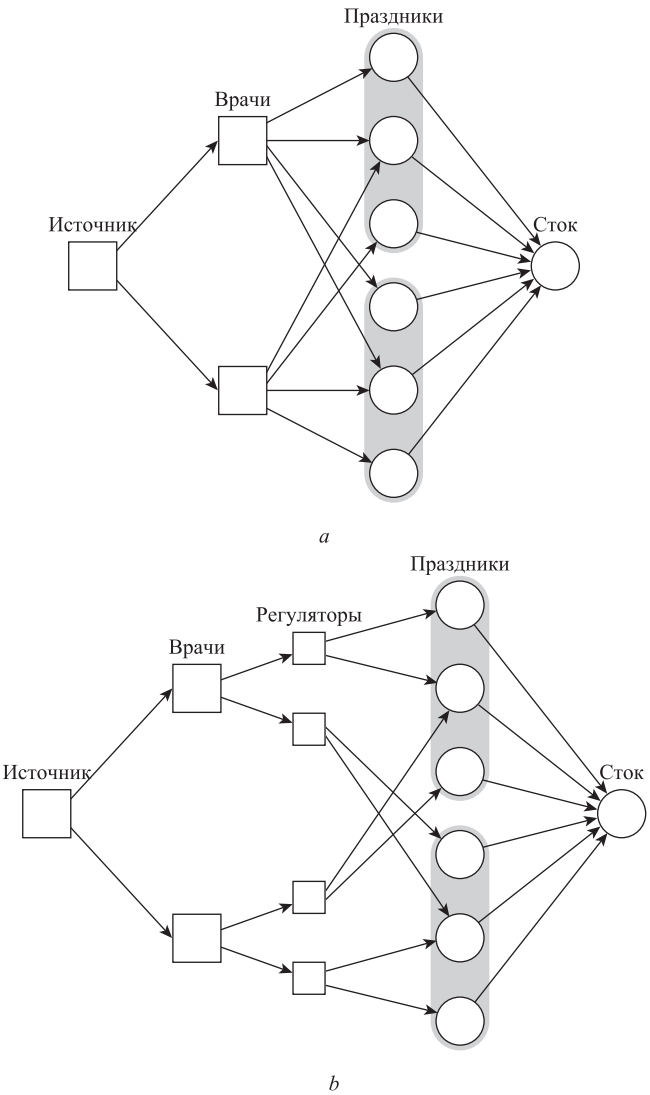


Рис. 7.23. *a* — врачи связываются с выходными днями без ограничения того, сколько дней за один праздничный период может отработать врач; *b* — потоковая сеть дополняется «регуляторами», не позволяющими врачу работать более одного дня в каждый праздничный период. Множества, выделенные серым цветом, соответствуют разным праздничным периодам

Но мы должны обеспечить выполнение дополнительного требования: что каждый врач может отработать не более одного дня за каждый праздничный период. Для этого берется каждая пара (i, j), состоящая из врача i и праздничного периода j, и добавляется «регулятор» — новый узел w_{ij} с входящим ребром с пропускной способностью 1 от узла врача u_i, и выходящими ребрами с пропускной способно-

стью 1 к каждому дню праздничного периода j, когда врач i доступен для работы. Регулятор «сужает» поток от u_i к дням праздничного периода j, чтобы к ним могла проходить максимум одна единица потока. Схема изображена на рис. 7.23, *b*. Как и прежде, мы назначаем для стока уровень потребления $+d$, для источника — уровень потребления $-d$, и ищем действительную циркуляцию. Общее время выполнения соответствует времени построения графа, которое равно $O(nd)$, плюс время поиска одной действительной циркуляции в этом графе.

Правильность алгоритма следует из следующего утверждения.

(7.69) Вариант распределения врачей по праздничным дням с соблюдением всех ограничений существует в том и только в том случае, если в построенной нами потоковой сети существует действительная циркуляция.

Доказательство. Во-первых, если вариант связывания врачей с праздничными днями с соблюдением всех ограничений существует, то мы можем построить следующую циркуляцию. Если врач i работает в день l праздничного периода j, мы передаем одну единицу потока по пути s, u_i, w_{ij}, v_l, t; это делается для всех таких пар (i, l). Так как распределение врачей удовлетворяет всем ограничениям, полученная циркуляция соблюдает все пропускные способности; кроме того, она отправляет d единиц потока из s и доставляет его в t, а следовательно, соблюдает уровни потребления.

И наоборот, предположим, что существует действительная циркуляция. В этом направлении доказательства мы покажем, как использовать циркуляцию для построения расписания для всех врачей. Прежде всего, согласно (7.52), существует действительная циркуляция, в которой все величины потоков являются целыми числами. Теперь мы построим расписание следующим образом: если ребро (w_{ij}, v_l) передает единицу потока, то врач i работает в день l. Из-за пропускных способностей в полученном расписании каждый врач работает не более c дней, не более одного в каждом праздничном периоде, и в каждый день работает один врач. ■

Упражнения

1. (a) Перечислите все минимальные разрезы s–t в потоковой сети, изображенной на рис. 7.24. Рядом с каждым ребром на схеме обозначена его пропускная способность.

   (b) Какова минимальная пропускная способность разреза s–t в потоковой сети на рис. 7.25? Как и в предыдущем случае, пропускная способность каждого ребра обозначена рядом с ним.

2. На рис. 7.26 изображена потоковая сеть с вычисленным потоком s–t. Рядом с каждым ребром на схеме обозначена его пропускная способность, а числа в прямоугольниках задают величину потока, передаваемого по каждому ребру. (Ребра без чисел в прямоугольниках — а именно четыре ребра с пропускной способностью 3 — не передают поток.)

(a) Какова величина этого потока? Является ли он максимальным потоком (s, t) в этом графе?

(b) Найдите минимальный разрез s–t в потоковой сети, изображенной на рис. 7.26. Укажите, чему равна его пропускная способность.

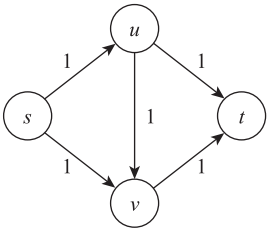


Рис. 7.24. Найдите минимальные разрезы s–t в этой потоковой сети

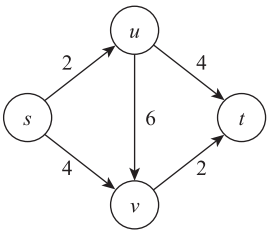


Рис. 7.25. Определите минимальную пропускную способность разреза s–t в этой потоковой сети

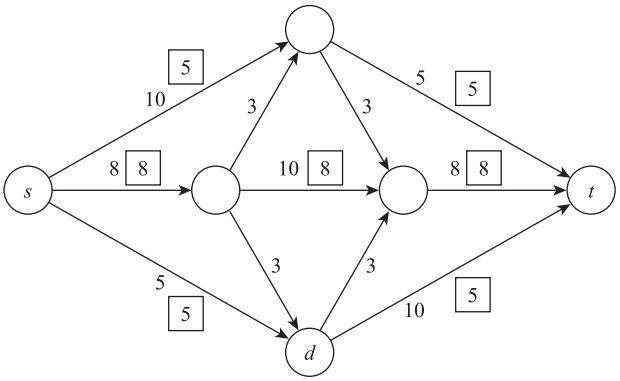


Рис. 7.26. Найдите величину изображенного потока. Является ли он максимальным? Найдите минимальный разрез

3. На рис. 7.27 изображена потоковая сеть с вычисленным потоком s–t. Рядом с каждым ребром на схеме обозначена его пропускная способность, а числа в прямоугольниках задают величину потока, передаваемого по каждому ребру. (Ребра без чисел в прямоугольниках не передают поток.)

(a) Какова величина этого потока? Является ли он максимальным потоком (s, t) в этом графе?

(b) Найдите минимальный разрез s–t в потоковой сети, изображенной на рис. 7.27. Укажите, чему равна его пропускная способность.

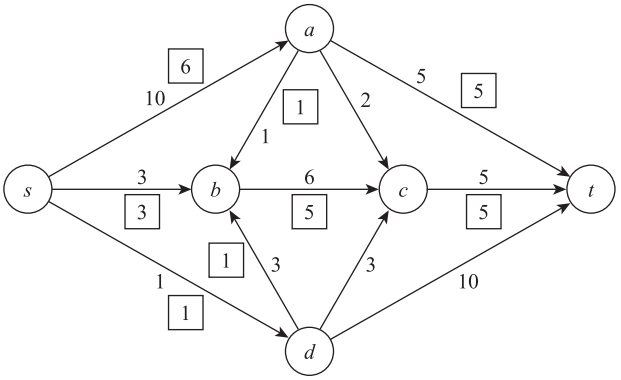


Рис. 7.27. Найдите величину изображенного потока. Является ли он максимальным? Найдите минимальный разрез

4. Решите, является ли следующее утверждение истинным или ложным. Если утверждение истинно, приведите краткое объяснение, а если ложно — приведите контрпример.

   Пусть G — произвольная потоковая сеть с источником s, стоком t и положительной целочисленной пропускной способностью c_e для каждого ребра e. Если f — максимальный поток s–t в G, то f насыщает каждое ребро, выходящее из s (то есть для всех ребер e, выходящих из s, выполняется условие $f(e) = c_e$).

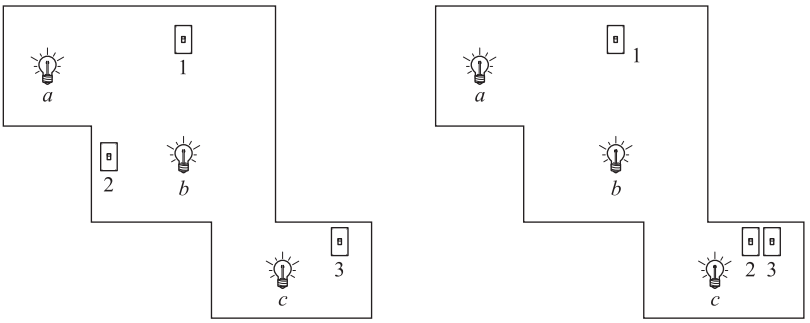
5. Решите, является ли следующее утверждение истинным или ложным. Если утверждение истинно, приведите краткое объяснение, а если ложно — приведите контрпример.

   Пусть G — произвольная потоковая сеть с источником s, стоком t и положительной целочисленной пропускной способностью c_e для каждого ребра e; пусть (A, B) — минимальный разрез s–t для этих пропускных способностей $\{c_e : e \in E\}$. Предположим, каждая пропускная способность увеличивается на 1; в этом случае (A, B) останется минимальным разрезом s–t для новых пропускных способностей $\{1 + c_e : e \in E\}$.

6. Представители Комиссии по архитектурной эргономике обратились к вам за консультацией.

   Архитекторы стремятся сделать свои дома как можно более удобными для их обитателей, но у них возникают большие проблемы с системой размещения светильников и выключателей в проектируемых домах. Для примера рассмотрим одноэтажный дом с n светильниками и n местами для настенных выключателей. Требуется связать каждый выключатель с одним светильником так, чтобы человек, стоящий у выключателя, *видел* управляемый им светильник.

Иногда это возможно, иногда нет. Возьмем два простых плана домов на рис. 7.28; на них размещены три светильника (a, b, c) и три выключателя (1, 2, 3). На рис. 7.28, *a* выключатели можно связать со светильниками так, чтобы от каждого выключателя проходила линия видимости к светильнику, но на рис. 7.28, *b* это невозможно.



a) Эргономичный план *b*) Неэргономичный план

Рис. 7.28. План (a) эргономичен, потому что выключатели можно связать со светильниками так, что каждый светильник будет виден от управляющего им выключателя. (Выключатель 1 связывается с a, выключатель 2 с b, а выключатель 3 с c). План (b) не эргономичен, потому что такое связывание невозможно

Назовем план с n позициями светильников и n позициями выключателей *эргономичным*, если каждый светильник можно связать с выключателем так, чтобы каждый светильник был виден от управляющего им выключателя. План представляется множеством из m горизонтальных или вертикальных отрезков на плоскости (стены), при этом i-я стена определяется конечными точками (x_i, y_i), (x'_i, y'_i). Положение каждого из n переключателей и каждого из n светильников задается его координатами на плоскости. Светильник *виден* от переключателя, если соединяющий их отрезок не пересекает ни одну стену.

Предложите алгоритм, который будет решать, является ли заданный план эргономичным. Время выполнения алгоритма должно быть полиномиальным по m и n. Считайте, что у вас имеется процедура с временем выполнения $O(1)$, которая получает два отрезка и проверяет, пересекаются ли они на плоскости.

7. Рассмотрим группу клиентов мобильных устройств, которые должны быть подключены к одной из нескольких возможных базовых станций. Всего клиентов n, позиция каждого клиента задается координатами (x, y) на плоскости. Также существуют k базовых станций; их позиции тоже заданы координатами (x, y). Требуется, чтобы каждый клиент подключался ровно к одной станции. Выбор вариантов подключения ограничивается парой факторов.

   Во-первых, существует параметр дистанции r — клиент может быть подключен только к базовой станции, находящейся от него в пределах расстояния r. Также существует параметр предельной нагрузки L — к одной базовой станции может быть подключено не более L клиентов.

Предложите алгоритм с полиномиальным временем для решения следующей задачи: для множества клиентов и множества базовых станций с заданными позициями, а также параметров дистанции и предельной нагрузки определить, возможно ли одновременное подключение всех клиентов к базовым станциям с соблюдением условий дистанции и предельной нагрузки из предыдущего абзаца.

8. По статистике наступление весны обычно сопровождается повышением численности несчастных случаев и потребности в срочной медицинской помощи, часто требующей переливания крови. Одна больница пытается оценить, достаточны ли ее запасы донорской крови.

   Как известно, в крови каждого человека присутствуют определенные антигены (своего рода молекулярная сигнатура); пациенту нельзя переливать кровь с определенным антигеном, если этот антиген отсутствует в его крови. А если говорить конкретнее, в соответствии с этим правилом кровь делится на четыре группы: A, B, AB и O. Кровь группы A содержит антиген A, кровь группы B — антиген B, кровь группы AB содержит оба антигена, и кровь группы O не содержит ни одного. Таким образом, пациентам с группой крови A при переливании могут получать только кровь групп A и O, пациентам с группой B — только B и O, пациентам с группой O — только O, и пациенты с группой крови AB могут получать кровь любой группы[1].

   (a) Пусть s_O, s_A, s_B и s_{AB} — запасы крови разных групп. Известны прогнозируемые потребности в крови всех четырех групп d_O, d_A, d_B и d_{AB} на следующую неделю. Предложите алгоритм с полиномиальным временем для определения достаточности имеющихся запасов при прогнозируемой потребности.

   (b) Рассмотрим следующий пример: ожидается, что за следующую неделю потребуется не более 100 единиц крови. Типичное распределение крови по группам среди пациентов в США выглядит так: примерно 45 % — группа O, 42 % — группа A, 10 % — группа B, и 3 % — группа AB. Больница хочет знать, хватит ли имеющегося запаса при поступлении 100 пациентов с ожидаемым распределением групп. Всего на складе доступно 105 единиц. В следующей таблице приведена сводка прогнозируемой потребности и имеющихся запасов.

Группа крови	Запас	Потребность
O	50	45
A	36	42
B	11	8
AB	8	3

[1] Австрийский ученый Карл Ландштейнер получил Нобелевскую премию в 1930 году за открытие групп крови A, B, AB и O.

Достаточно ли имеющихся 105 единиц для потребности в 100 единиц? Найдите распределение, удовлетворяющее максимально возможное количество пациентов. Чтобы показать, почему не все пациенты смогут получить донорскую кровь, воспользуйтесь аргументом, основанным на разрезе с минимальной пропускной способностью. Также приведите объяснение этого факта, понятное администрации больницы, не искушенной в алгоритмах (соответственно, в этом объяснении не должны встречаться слова «поток», «разрез» или «граф» в том смысле, в котором они использовались в книге).

9. Алгоритмы потоков в сети часто встречаются при организации работ в зоне природных катастроф и других экстренных ситуациях, так как крупные непредвиденные события часто требуют перемещения и эвакуации большого количества людей за короткое время.

   Рассмотрим следующую ситуацию: из-за серьезного наводнения спасатели обнаружили n пострадавших, которых необходимо срочно доставить в больницу. В этом регионе имеются k больниц, и каждый из n пострадавших должен быть доставлен в больницу, находящуюся не более чем в получасе езды от его текущего местонахождения (таким образом, у каждого пострадавшего имеется свой выбор больницы в зависимости от его текущего местонахождения).

   В то же время было бы нежелательно создавать чрезмерную нагрузку на отдельную больницу, направляя в нее слишком много пациентов. Спасатели общаются по мобильным телефонам, и хотят совместными усилиями подобрать больницы для пострадавших так, чтобы *сбалансировать* нагрузку: в каждую больницу должны поступить не более $\lceil n/k \rceil$ пострадавших.

   Предложите алгоритм с полиномиальным временем, который получает информацию о местонахождении пострадавших и определяет, возможно ли такое распределение.

10. Имеется направленный граф $G = (V, E)$ с положительной целочисленной пропускной способностью c_e каждого ребра e, источником $s \in V$ и стоком $t \in V$. Также задан максимальный поток s–t в G, определяемый величиной потока f_e по каждому ребру e. Поток f является *ациклическим*: в G не существует цикла, в котором по всем ребрам передается положительный поток. Поток f также является целочисленным.

    Допустим, мы выбираем некоторое ребро $e^* \in E$ и уменьшаем его пропускную способность на одну единицу. Покажите, как найти максимальный поток в полученном графе за время $O(m + n)$, где m — количество ребер в G, а n — количество узлов.

11. Ваши знакомые программисты написали очень быстрый код нахождения максимального потока, основанный на многократном нахождении увеличивающих путей (как описано в разделе 7.1). Однако просматривая результаты выполнения, вы обнаруживаете, что алгоритм не всегда находит поток с максимальной величиной. Ошибка обнаруживается очень быстро; ваши знакомые не разобрались в проблеме обратных ребер, и их реализация строит остаточный граф, который включает только прямые ребра. Другими словами, она ищет пути s–t

в графе $\tilde{G}_f$, состоящем только из ребер e, для которых $f(e) < c_e$, и завершается при отсутствии улучшающего пути, состоящего только из таких ребер. Назовем эту реализацию «алгоритмом с прямыми ребрами». (Обратите внимание: мы не пытаемся указывать, как этот алгоритм выбирает свои пути с прямыми ребрами; он может выбирать их так, как считает нужным, при условии, что его выполнение будет завершено только при отсутствии путей с прямыми ребрами.)

Вам будет нелегко убедить своих знакомых, что код придется переписывать. Они утверждают, что алгоритм не только быстро работает, но и никогда не возвращает поток, величина которого меньше фиксированной части оптимального. А вы этому верите? Суть их утверждений можно выразить следующим образом.

Существует такая абсолютная константа $b > 1$ (не зависящая от конкретной входной потоковой сети), что для каждого экземпляра задачи максимального потока алгоритм с прямыми ребрами гарантированно находит поток с величиной не менее $1/b$ величины максимального потока (независимо от того, как он выбирает свои пути).

Решите, является ли это утверждение истинным или ложным. Приведите доказательство или опровержение.

12. Имеется потоковая сеть с ребрами единичной пропусной способности: она состоит из направленного графа $G = (V, E)$, источника $s \in V$ и стока $t \in V$; $c_e = 1$ для всех $e \in E$. Также задан параметр k. Требуется удалить k ребер таким образом, чтобы максимальный поток s–t в G сократился как можно сильнее. Другими словами, нужно найти множество ребер $F \subseteq E$, для которого $|F| = k$, а максимальный поток s–t в $G' = (V, E-F)$ был как можно меньше.

    Предложите алгоритм с полиномиальным временем для решения этой задачи.

13. В стандартной задаче о максимальном потоке s–t предполагается, что ребра имеют пропускные способности, а величина потока, проходящего через узел, не ограничена. В этой задаче рассматривается разновидность задач о максимальном потоке и минимальном разрезе с пропускными способностями узлов.

    Пусть $G = (V, E)$ — направленный граф с источником $s \in V$, и стоком $t \in V$ и неотрицательными пропускными способностями узлов $\{c_v \geq 0\}$ для всех $v \in V$. Для потока f в этом графе поток через узел v определяется как $f^{\text{in}}(v)$. Поток называется *действительным*, если он удовлетворяет обычным ограничениям сохранения потока и ограничениям пропускной способности узлов: $f^{\text{in}}(v) \leq c_v$ для всех узлов.

    Предложите алгоритм с полиномиальным временем для нахождения максимального потока s–t в такой сети, определяющей пропускные способности для узлов. Определите разрез s–t для сетей с пропускными способностями узлов и покажите, что аналогия теоремы о максимальном потоке и минимальном разрезе остается истинной.

14. *Задача о выходе* определяется следующим образом. Имеется направленный граф $G = (V, E)$ (схема дорожной сети). Узлы некоторой совокупности $X \subset V$ называются *населенными*, а узлы другой совокупности $S \subset V$ называются *безопасными*. (Предполагается, что X и S не пересекаются.) При возникновении

чрезвычайной ситуации требуется найти эвакуационные пути из населенных узлов в безопасные. Множество эвакуационных путей определяется как множество таких путей в G, что (i) каждый узел в X является начальным для одного пути, (ii) последний узел каждого пути принадлежит S, и (iii) пути не имеют пересекающихся ребер. Такой набор путей позволяет «вывести» обитателей населенных узлов в S без чрезмерной перегрузки ребер G.

(a) Для заданных G, X и S покажите, как за полиномиальное время принять решение о том, существует ли множество эвакуационных путей.

(b) Предположим, поставлена та же задача, что в (a), но мы хотим обеспечить еще более сильную версию запрета на перегрузку (iii). Таким образом, формулировка (iii) приводится к виду «пути не имеют общих *узлов*».

Покажите, как с таким новым условием за полиномиальное время принять решение о том, существует ли множество эвакуационных путей.

Также приведите пример с теми же G, X и S, для которых на (a) дается положительный ответ, а на (b) — отрицательный.

15. Предположим, вы и ваша знакомая Аланис проживаете в студенческой коммуне с $n - 2$ другими людьми. За следующие n дней каждый жилец квартиры должен приготовить ужин ровно один раз, так что ежедневно кто-то занимается приготовлением пищи.

Конечно, у каждого жильца имеются свои планы на некоторые дни (экзамены, концерты и т. д.), поэтому решить, кто должен готовить в тот или иной вечер, будет непросто. Для конкретности обозначим жильцов $\{p_1, ..., p_n\}$, а дни $\{d_1, ..., d_n\}$; для человека p_i существует множество дней $S_i \subset \{d_1, ..., d_n\}$, в которые они *не могут* заниматься приготовлением ужина.

Действительным расписанием называется такое распределение жильцов коммуны по дням, чтобы каждый жилец готовил ровно в один день, ежедневно кто-то готовил, и если жилец p_i готовит в день d_j, то $d_j \notin S_i$.

(a) Опишите такой двудольный граф G, в котором идеальное паросочетание существует в том и только том случае, если существует действительное расписание приготовления обедов.

(b) Ваша знакомая Аланис берется за составление действительного расписания. После значительных усилий она строит то, что по ее мнению является действительным расписанием, и уходит на лекции. К сожалению, заглянув в составленное ей расписание, вы замечаете серьезную проблему. $n - 2$ жильцов распределены в разные дни, в которые они свободны; здесь проблем нет. Но внезапно выясняется, что для двух других людей p_i и p_j, и двух дней d_k и d_l, оба человека были назначены на день d_k, а в день d_l никто не готовит.

Вы хотите исправить ошибку Аланис — но так, чтобы расписание не пришлось строить заново. Покажите, что при наличии такого «почти правильного» расписания можно за время $O(n^2)$ решить, существует ли действительное расписание для текущих исходных данных. (А если расписание существует, выведите его.)

16. На заре существования Всемирной паутины часто говорили, что большая часть огромного потенциала таких компаний, как Yahoo!, кроется в «зрачках пользователя» — то есть в том простом факте, что миллионы людей ежегодно просматривают их страницы. Кроме того, убеждая пользователей сообщать личные данные при регистрации, такой сайт, как Yahoo!, может выводить целенаправленную рекламу при каждом посещении — возможность, недоступная для телевизионных сетей или журналов. Скажем, если пользователь сообщает Yahoo!, что он — 20-летний студент Корнелльского университета, то сайт выводит баннер с рекламой квартир в Итаке (штат Нью-Йорк); если же пользователь является 50-летним финансовым брокером из Гринвича (штат Коннектикут), сайт выводит рекламу автосалона по продаже «Линкольнов».

    Но чтобы решить, какую рекламу показывать тем или иным пользователям, необходимо провести достаточно серьезные вычисления. Предположим, менеджеры популярного сайта выявили k разных *демографических групп* $G_1, G_2, ..., G_k$. (Группы могут перекрываться; например, группа G_1 может состоять из всех жителей штата Нью-Йорк, а группа G_2 — из обладателей ученой степени по информатике.) Сайт заключил контракты с m разными *рекламодателями* на показ определенного количества копий рекламы на сайте. Контракт с i-м рекламодателем формулируется примерно так:

    - Имеется подмножество $X_i \subseteq \{G_1, ..., G_k\}$ демографических групп; рекламодатель i хочет, чтобы его реклама выводилась только у пользователей, принадлежащих хотя бы к одной из демографических групп множества X_i.
    - Для заданного числа r_i рекламодатель i хочет, чтобы его реклама выводилась минимум r_i пользователям в минуту.

    Рассмотрим задачу планирования хорошей *рекламной политики* — способа показа одной рекламы для одного пользователя сайта. Предположим, в некоторую минуту с сайтом работают n пользователей. Так как у нас имеется регистрационная информация о каждом пользователем, мы знаем, что пользователь j (для $j = 1, 2, ..., n$) принадлежит подмножеству $U_j \subseteq \{G_1, ..., G_k\}$ демографических групп. Задача формулируется так: существует ли способ вывода одной рекламы для каждого пользователя, чтобы выполнить контракты сайта с m рекламодателями за данную минуту (то есть для всех $i = 1, 2, ..., m$ можно ли показать по крайней мере r_i из n пользователей, каждый из которых принадлежит по крайней мере одной демографической группе из X_i, рекламу, предоставленную рекламодателем i?)

    Предложите эффективный алгоритм, который решает, возможно ли это, и если возможно — выбирает рекламу, выводимую для каждого пользователя.

17. Администраторы сети обратились к вам за помощью в диагностике. Сеть спроектирована для передачи трафика из источника s в приемник t, поэтому для ее моделирования можно использовать направленный граф $G = (V, E)$, в котором пропускная способность каждого ребра равна 1, а каждый узел лежит по крайней мере на одном пути от s до t.

Когда в сети все работает нормально, максимальный поток s–t в G имеет величину k. Однако хакерская атака уничтожила некоторые ребра, и среди оставшихся ребер не осталось пути из s в t. По причинам, в которые мы не будем углубляться, администраторы полагают, что уничтожено только k ребер — минимальное количество, необходимое для отделения s от i (то есть размер минимального разреза s–t); будем считать, что их оценка верна.

На узле s работает контрольная программа. Если передать ей команду *ping*(v) (для некоторого узла v), программа сообщает, доступен ли в настоящее время путь из s в v. (Таким образом, вызов *ping*(t) сообщит, что путь в настоящее время недоступен; с другой стороны, вызов *ping*(s) всегда сообщает о наличии пути от s до s.) Проверять все ребра в сети было бы непрактично, поэтому администраторы хотя определить размер сбоя при помощи этой программы и разумного применения команды *ping*.

А теперь задача: предоставьте алгоритм, который выдает серию команд *ping* к разным узлам сети, а затем выводит полное множество узлов, в настоящий момент недоступных из s. Конечно, можно опросить каждый узел в сети, но вам хотелось бы ограничиться меньшим количеством проверок (предполагается, что разрушены только k ребер). При выдаче серии команд ваш алгоритм может выбирать следующий проверяемый узел на основании результатов предшествующих операций *ping*.

Предложите алгоритм, который решает эту задачу с использованием всего $O(k \log n)$ проверок.

18. Рассмотрим задачу о двудольном паросочетании для двудольного графа $G = (V, E)$. Как обычно, мы говорим, что множество V разбито на множества X и Y, один конец каждого ребра принадлежит X, а другой принадлежит Y.

Если M — паросочетание в G, говорят, что M *покрывает* узел y, если y является концом одного из ребер в M.

(a) Рассмотрим следующую задачу: имеется граф G и паросочетание M в G. Для заданного числа k требуется решить, существует ли в G паросочетание M', для которого:

(i) M' содержит на k больше ребер, чем M, и

(ii) Каждый узел $y \in Y$, покрываемый M, также покрывается M'.

Эта задача называется *задачей расширения покрытия* с входными значениями G, M и k, а M' называется *решением* экземпляра задачи.

Предложите алгоритм с полиномиальным временем, который получает экземпляр задачи расширения покрытия, и либо возвращает решение M', либо сообщает (обоснованно) об отсутствии решения. (Приведите анализ времени выполнения и кратко обоснуйте его правильность.)

Примечание: возможно, часть (b) поможет вам в поиске решения.

Пример. Рассмотрим граф на рис. 7.29; допустим, M — паросочетание, состоящее из ребра (x_1, y_2). Приведенный выше вопрос задается для $k = 1$.

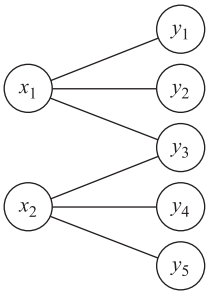


Рис. 7.29. Экземпляр расширения покрытия

В этом случае ответ на задачу расширения покрытия будет положительным. В качестве M' можно выбрать (например) паросочетание из двух ребер (x_1, y_2) и (x_2, y_4); M' содержит на одно ребро больше, чем M, и узел y_2 также покрывается M'.

(b) Приведите экземпляр задачи расширения покрытия, заданный параметрами G, M и k, для следующей ситуации.

У экземпляра есть решение, но в любом решении M' ребра M не образуют подмножество ребер M'.

(c) Пусть G — двудольный граф, а M — произвольное паросочетание в G. Рассмотрим следующие две характеристики.

- K_1 — размер наибольшего паросочетания M', для которого каждый узел y, покрываемый M, также покрывается M'.
- K_2 — размер наибольшего паросочетания M'' в G.

Очевидно, $K_1 \leq K_2$, так как K_2 определяется по *всем возможным* паросочетаниям в G.

Докажите, что $K_1 = K_2$; а именно, что максимальное паросочетание может быть получено даже в том случае, если ограничиться покрытием всех узлов, покрываемых исходным паросочетанием M.

19. Местная больница, которой вы уже помогали по разным вопросам планирования, обратилась с новой задачей. Для каждого из следующих n дней было определено количество необходимых врачей; таким образом, в день i должны присутствовать *ровно* p_i врачей.

Всего в больнице работает k врачей, каждому из которых предложено составить список дней, в которые он желает работать. Таким образом, врач j определяет множество L_j дней для своей работы.

Ваша система должна учесть все списки и попытаться вернуть для каждого врача j список L'_j, обладающий следующими свойствами:

(A) L'_j является подмножеством L_j, то есть врач j работает только в те дни, которые он считает приемлемыми.

(B) По всему множеству списков $L'_1, ..., L'_k$ в день i присутствуют ровно p_i врачей (для $i = 1, 2, ..., n$).

(a) Опишите алгоритм с полиномиальным временем выполнения, реализующий эту систему. А конкретно предложите алгоритм с полиномиальным временем, который получает числа $p_1, p_2, ..., p_n$, и списки $L_1, ..., L_k$, и делает одно из двух:

- Возвращает списки $L'_1, L'_2, ..., L'_k$, удовлетворяющие свойствам (A) и (B); или
- Сообщает (обоснованно) о том, что множество списков $L'_1, ..., L'_k$, удовлетворяющее свойствам (A) и (B), не существует.

(b) Руководство больницы обнаружило, что списки, полученные от врачей, слишком ограничены, и система слишком часто сообщает об отсутствии допустимого множества списков $L'_1, L'_2, ..., L'_k$ (сообщает правильно, но не вовремя).

По этой причине требования немного ослабляются. Добавляется новый параметр $c > 0$; система должна попытаться вернуть для каждого врача j список L'_j, обладающий следующими свойствами.

(A^*) L'_j содержит не более c дней, не входящих в список L_j.

(B) (То же) По всему множеству списков $L'_1, ..., L'_k$ в день i присутствуют ровно p_i врачей (для $i = 1, 2, ..., n$).

Опишите алгоритм с полиномиальным временем выполнения, реализующий измененную систему. Алгоритм получает числа $p_1, p_2, ..., p_n$, списки $L_1, ..., L_k$ и параметр $c > 0$, и делает одно из двух:

- Возвращает списки $L'_1, L'_2, ..., L'_k$, удовлетворяющие свойствам (A^*) и (B^*); или
- Сообщает (обоснованно) о том, что множество списков $L'_1, ..., L'_k$, удовлетворяющее свойствам (A^*) и (B^*), не существует.

20. Ваши знакомые участвуют в крупномасштабном эксперименте по изучению атмосферы. Они хотят получить измерения по множеству S из n разных атмосферных характеристик (например, концентрации озона в разных местах); они располагают m зондами, которые запускаются для проведения измерений. Каждый зонд может провести не более двух измерений.

    К сожалению, не все зонды способны измерять все условия, поэтому для каждого зонда $i = 1, ..., m$ имеется множество S_i характеристик, которые могут измеряться этим зондом. Наконец, чтобы результаты были более надежными, они планируют провести каждое измерение минимум с k разных зондов. (Обратите внимание: один зонд не может измерять одну характеристику дважды.) Теперь нужно рассчитать, какие характеристики должны измеряться тем или иным зондом.

    Пример. Допустим, $k = 2$, существуют $n = 4$ условий c_1, c_2, c_3, c_4, и имеются $m = 4$ зонда для измерения характеристик; при этом действуют ограничения $S_1 = S_2 = \{c_1, c_2, c_3\}$, и $S_3 = S_4 = \{c_1, c_3, c_4\}$. Один из возможных способов гарантировать, что каждая характеристика будет измерена не менее $k = 2$ раз, выглядит так:

- зонд 1 измеряет условия c_1, c_2,
- зонд 2 измеряет условия c_2, c_3,
- зонд 3 измеряет условия c_3, c_4,
- зонд 4 измеряет условия c_1, c_4.

(a) Предложите алгоритм с полиномиальным временем, который получает входные данные экземпляра задачи (n характеристик, множества S_i для каждого из m зондов, параметр k) и решает, возможно ли измерить каждую характеристику с k разных зондов при том, что каждый зонд может измерять не более двух характеристик.

(b) Вы показываете своим знакомым решение, вычисленное алгоритмом из пункта (a) — но к вашему изумлению, слышите: «Не годится — одна из характеристик измеряется только зондами одного поставщика». Но до этого о поставщиках вы ничего не слышали; кажется, в задаче появился дополнительный нюанс, о котором забыли упомянуть....

Каждый зонд производится одним из трех разных поставщиков, участвующих в эксперименте. Одно из требований к эксперименту заключается в том, что не должно быть ни одной характеристики, все k измерений которой получены с зондов, выпущенных одним поставщиком.

Допустим, зонд 1 выпущен первым поставщиком, зонды 2 и 3 — вторым, а зонд 4 — третьим. Тогда предыдущее решение уже не работает, так как оба измерения c_3 выполнены зондами второго поставщика. С другой стороны, мы можем использовать зонды 1 и 2 для измерения характеристик c_1, c_2, а зонды 3 и 4 — для измерения характеристик c_3, c_4.

Объясните, как преобразовать алгоритм с полиномиальным временем из части (a) в новый алгоритм, который решает, существует ли решение с выполнением всех требований из (a), а также нового требования о поставщиках.

21. Вы помогаете организовать занятия в колледже, который решил выдать всем студентам беспроводные планшетные компьютеры на семестр. Таким образом, имеется набор из n беспроводных планшетов; также имеется набор из n беспроводных точек доступа, к которым планшеты могут подключаться при нахождении в зоне действия.

В настоящее время планшеты хаотично распределены по всему городку; планшет l нахоится в зоне действия множества S' точек доступа. Будем считать, что каждый планшет находится в зоне действия по крайней мере одной точки доступа (так что множества S' не пусты); также предполагается, что в зоне действия каждой точки доступа p находится по крайней мере один планшет.

Для нормальной работы программ беспроводной связи необходимо постараться связать планшеты с точками доступа так, чтобы каждый планшет и каждая точка доступа были задействованы минимум в одном подключении. *Тестовым множеством* T будет называться совокупность упорядоченных пар вида (l, p) для планшета l и точки доступа p, обладающих следующими свойствами:

(i) Если $(l, p) \in T$, то l находится в зоне действия p (то есть $p \in S_l$).

(ii) Каждый планшет входит минимум в одну упорядоченную пару в T.

(iii) Каждая точка доступа входит минимум в одну упорядоченную пару в T.

Итак, проверяя все подключения, определяемые парами из T, мы можем убедиться в том, что программное обеспечение на каждом планшете и каждой точке доступа работает правильно.

Задача: для заданных множеств S_l для каждого планшета (какие планшеты находятся в зоне действия тех или иных точек доступа) и числа k решите, существует ли тестовое множество с размером, не превышающим k.

Пример. Допустим, $n = 3$; планшет 1 находится в зоне действия точек доступа 1 и 2; планшет 2 находится в зоне действия точки доступа 2; планшет 3 находится в зоне действия точек доступа 2 и 3. В этом случае множество пар

(планшет 1, точка доступа 1), (планшет 2, точка доступа 2),

(планшет 3, точка доступа 3)

образует тестовое множество с размером 3.

(a) Приведите пример экземпляра задачи, для которого не существует тестовое множество с размером n. (Вспомните: предполагается, что каждый планшет находится в зоне действия минимум одной точки доступа и в зоне действия каждой точки доступа p находится минимум один планшет.)

(b) Предложите алгоритм с полиномиальным временем, который получает экземпляр задачи (включая параметр k) и решает, существует ли тестовый размер с размером, не превышающим k.

22. Пусть M — матрица $n \times n$, каждый элемент которой равен 0 или 1; m_{ij} — элемент матрицы, находящийся на пересечении строки i и столбца j. *Диагональным элементом* называется элемент вида m_{ii} для некоторого i.

Перестановкой строк i и j матрицы M называется операция, при которой меняются местами значения m_{ik} и m_{jk} для $k = 1, 2, ..., n$. Перестановка двух столбцов определяется аналогично.

Матрица M называется *перестраиваемой*, если возможно выполнить перестановки некоторых пар строк и некоторых пар столбцов (в произвольной последовательности) так, чтобы после всех перестановок все диагональные элементы M были равны 1.

(a) Приведите пример патрицы M, которая не является перестраиваемой, но в каждой строке и каждом столбце которой по крайней мере один элемент равен 1.

(b) Предложите алгоритм с полиномиальным временем, определяющий, является ли матрица M с элементами 0-1 перестраиваемой.

23. Имеется потоковая сеть G с источником s и приемником t; вы хотите как-то выразить следующие интуитивные понятия: некоторые узлы явно находятся на «стороне источника» основных «узких мест»; другие узлы явно находятся на «стороне стока» основных «узких мест», а третьи располагаются посредине.

Однако в G может быть много минимальных разрезов, поэтому необходимо проявить осторожность в выборе точной формулировки.

Один из способов разбиения узлов G на три категории выглядит так:

- Узел v называется *верховым*, если для всех минимальных разрезов s–t (A, B) выполняется $v \in A$ — то есть v лежит на стороне источника каждого минимального разреза.
- Узел v называется *низовым*, если для всех минимальных разрезов s–t (A, B) выполняется $v \in B$ — то есть v лежит на стороне стока каждого минимального разреза.
- Узел v называется *срединным*, если он не является ни верховым, ни низовым; существует как минимум один минимальный разрез s–t (A, B), для которого $v \in A$, и как минимум один минимальный разрез s–t (A', B'), для которого $v \in B'$.

Предложите алгоритм, который получает потоковую сеть G и относит каждый из ее узлов к одной из трех категорий (верховой, низовой, срединный). Время выполнения алгоритма должно быть равно произведению времени, необходимого для вычисления одного максимального потока, на константу.

24. Пусть $G = (V, E)$ — направленный граф с источником $s \in V$, и стоком $t \in V$ и неотрицательными пропускными способностями ребер $\{c_e\}$. Предложите алгоритм с полиномиальным временем, который определяет, существует ли в G уникальный минимальный разрез s–t (то есть разрез s–t, пропускная способность строго меньше пропускной способности любого другого разреза s–t).

25. Предположим, вы живете в большой квартире с множеством друзей. За год нередко возникают ситуации, когда кто-то из вас оплачивает расходы, общие для некоторого подмножества жильцов, ожидая, что в конце года все будет более или менее сбалансировано. Например, один из вас может оплатить весь телефонный счет за месяц, другой время от времени закупает продукты на всех в ближайшем магазине, а третий иногда оплачивает своей кредиткой весь счет в местном ресторанчике.

Но год подошел к концу, и пришло время свести счеты. В квартире живут n людей; для каждой упорядоченной пары (i, j) существует накопленная за год сумма $a_{ij} \geq 0$, которую жилец i задолжал j. Требуется, чтобы для любых двух людей i и j по крайней мере одна из величин a_{ij} или a_{ji} была равна 0. Сделать это несложно: если выясняется, что i должен j положительную сумму x, а j должен i положительную сумму $y < x$, то мы вычитаем y из обоих сторон и объявляем $a_{ij} = x - y$ при $a_{ji} = 0$. В контексте этих величин *дисбаланс* жильца i определяется как сумма величин, которую i должны все остальные, за вычетом суммы величин, которые i должен всем остальным. (Дисбаланс может быть положительным, отрицательным или нулевым.)

Чтобы обнулить все дисбалансы и расстаться в хороших отношениях, некоторые жильцы выписывают чеки другим жильцам; другими словами, для некоторых упорядоченных пар (i, j) i выписывает чек j на сумму $b_{ij} > 0$. Множество чеков будет называться *расчетом*, если для каждого жильца i общая сумма

чеков, полученных i, за вычетом общей суммы чеков, выписанных i, равна дисбалансу i. Наконец, все жильцы чувствуют, что для i было бы неправильно выписывать чек j, если i на самом деле не задолжал j, поэтому расчет называется *корректным*, если при выписывании i чека j выполняется условие $a_{ij} > 0$. Покажите, что для любого множества a_{ij} всегда существует корректный расчет, при котором выписываются не более n-1 чека; для этого приведите алгоритм с полиномиальным временем для вычисления такого расчета.

26. При виде вышек сотовой связи, возвышающихся на кукурузных полях и пастбищах, становится понятно, что мобильные телефоны активно используются в сельской местности. Рассмотрим очень упрощенную модель сети сотовой связи в области с малой плотностью населения.

    Известны позиции нескольких базовых станций n, заданных точками $b_1, ..., b_n$ на плоскости. Также известны позиции n сотовых телефонов, заданных точками $p_1, ..., p_n$. Наконец, задан еще один параметр — *зона действия* $\Delta > 0$. Множество сотовых телефонов будет называться *полностью связным*, если каждый телефон можно связать с базовой станцией с выполнением следующих условий:

    - Все телефоны подключены к разным базовым станциям, и
    - Если телефон в точке p_i подключен к базовой станции в точке b_j, то расстояние по прямой между точками p_i и b_j не превышает Δ.

    Предположим, владелец сотового телефона в точке p_1 отправляется в поездку, перемещаясь на z единиц расстояния на восток. При перемещении сотового телефона придется обновлять его подключение к базовой станции (возможно, несколько раз), чтобы множество телефонов оставалось полностью связным.

    Предложите алгоритм с полиномиальным временем, который будет принимать решение о том, возможно ли сохранять полную связность множества телефонов во время перемещения одного сотового телефона (предполагается, что все остальные телефоны при этом остаются неподвижными). Если это возможно, выведите последовательность подключений телефонов к базовым станциям, достаточную для сохранения полной связности; если это невозможно, выведите точку на пути перемещающегося телефона, в которой сохранение полной связности становится невозможным. Алгоритм должен выполняться за время $O(n^3)$, если это возможно.

    Пример. Допустим, телефоны находятся в точках $p_1 = (0, 0)$ и $p_2 = (2, 1)$; базовые станции расположены в точках $b_1 = (1, 1)$ и $b_2 = (3,1)$; и $\Delta = 2$. Телефон из точки p_1 перемещается на восток на расстояние 4 единицы, завершая свое движение в точке (4, 0). В этом случае можно сохранить полную связность множества телефонов во время перемещения: сначала p_1 подключается к b_1, а p_2 к b_2, а затем во время перемещения p_1 переподключается к b_2, а p_2 к b_1 (например, при прохождении p_1 точки (2, 0)).

27. Ваши знакомые, работающие на Западном побережье, решают скооперироваться для совместных поездок на работу. К сожалению, сидеть за рулем никто не любит, поэтому схема кооперации должна быть справедливой, чтобы никому из участников не пришлось слишком часто вести машину. Простая круговая схема

не подходит, потому что никто не ездит на работу ежедневно, и подмножество людей, пользующихся машиной, изменяется от одного дня к другому.

Рассмотрим один из способов определения «справедливости». Допустим, люди обозначаются $S = \{p_1, ..., p_k\}$. Общим обязательством p_j по множеству дней является ожидаемое количество раз, в течение которых p_j пришлось бы вести машину, если бы водитель выбирался случайным образом среди тех, кто едет на работу каждый день. А конкретнее, предположим, что план совместных поездок рассчитан на d дней, и в i-й день на работу едет подмножество людей $S_i \subseteq S$. Тогда приведенное выше определение обязательства Δj для p_j может быть записано в виде $\Delta_j = \sum_{i:p_j \in S_i} \frac{1}{|S_i|}$. В идеале хотелось бы потребовать, что p_j ведет машину не более Δj раз; к сожалению, Δj может не быть целым числом.

Планом совместных поездок назовем выбор водителя на каждый день — то есть последовательность $p_{i_1}, p_{i_2}, ..., p_{i_d}$, где $p_{i_t} \in S_t$ — а *справедливым планом* назовем план, в котором каждый участник p_j выбирается водителем не более чем для $\lceil \Delta_j \rceil$ дней.

(a) Докажите, что для произвольной последовательности множеств $S_1, ..., S_d$ существует справедливое расписание совместных поездок.

(b) Предложите алгоритм для вычисления справедливого плана совместных поездок с временем выполнения, полиномиальным по k и d.

28. Группа студентов решила добавить новые функции в компьютерную систему подготовки учебных курсов для отработки аспектов планирования, в настоящее время не поддерживаемых программой. Работа начинается с модуля, который помогает планировать учебное время в начале семестра. Прототип работает по следующим правилам: прежде всего расписание остается постоянным, поэтому задачу планирования достаточно решить для одной недели. Администратор вводит набор неперекрывающихся одночасовых интервалов $I_1, I_2, ..., I_k$, в которые занятия могут проводиться ассистентами; итоговое расписание представляет собой подмножество (как правило, неполное) этих интервалов. Затем каждый ассистент вводит свое расписание на неделю с временем, в которое он будет свободен для проведения занятий.

Наконец, администратор указывает параметры a, b и c: каждому ассистенту было бы желательно выделить от a до b учебных часов в неделю, а количество учебных часов в неделю составляет ровно c.

Итак, задача в том, как назначить каждому ассистенту некоторые из учебных часов, чтобы каждый ассистент был доступен в каждый из его учебных часов, и при этом было обеспечено правильное количество учебных часов. (В каждый час должен присутствовать только один ассистент.)

Пример. Допустим, имеются пять временных интервалов:

I_1 = Понедельник 15–16; I_2 = Вторник 13–14; I_3 = Среда 10–11; I_4 = Среда 15–16; I_5 = Четверг 10–11.

Есть два ассистента; первый может работать в любое время в понедельник и среду во второй половине дня, а второй — в любое время во вторник, среду и четверг. (В общем случае правила доступности ассистентов могут быть более сложными, но мы не будем усложнять пример.) Наконец, каждому ассистенту должно быть выделено от $a = 1$ до $b = 2$ учебных часов, а общее количество учебных часов за неделю должно быть равно $c = 3$.

В одном из возможных решений первому ассистенту выделяются учебные часы из интервала I_1, а второму — часы из интервалов I_2 и I_5.

(a) Предложите алгоритм с полиномимальным временем, который получает входные данные экземпляра этой задачи (интервалы, свободное время ассистентов и параметры a, b и c) и делает одно из двух:

- Стоит действительное расписание с указанием того, какому ассистенту выделяются те или иные интервалы, или
- Сообщает (обоснованно) о том, что требуемый способ распределения учебных часов не существует.

(b) Функция планирования учебных часов становится очень популярной, и администрация начинает хотеть большего. В частности, она замечает, что было бы неплохо повысить количество учебных часов в дни сдачи домашних работ.

Администрация хочет иметь возможность задать новый параметр — «плотность» учебных часов для каждого дня недели: число d_i означает, что в заданный день недели i должно быть назначено не менее d_i учебных часов.

Допустим, в предыдущем примере добавляется ограничение, согласно которому в среду и четверг должно быть проведено по крайней мере по одному учебному часу. В этом случае приведенное выше решение не работает; но существует возможное решение, в котором первому ассистенту выделяются учебные часы в интервале I_1, а второму — часы в интервалах I_3 и I_5. (Другое решение — первому ассистенту выделяются часы в интервалах I_1 и I_4, а второму — в интервале I_5.)

Предложите алгоритм с полиномиальным временем, который строит расписание для более сложного набора ограничений. Алгоритм должен либо строить расписание, либо сообщать (обоснованно) о том, что оно не существует.

29. Ваши знакомые открыли небольшую компанию — в настоящее время они заняли под офис родительский гараж в Санта-Кларе. Сейчас компания занимается портированием своего программного обеспечения со старой на новую, более мощную платформу; сейчас она столкнулась со следующей проблемой.

Имеется набор из n приложений $\{1, 2, ..., n\}$, работающих в старой системе; некоторые из этих приложений нужно портировать в новую систему. Ожидается, что портирование приложения i в новую систему принесет чистую (денежную) прибыль $b_i \geq 0$. Приложения взаимодействуют друг с другом; если только одно из двух активно взаимодействующих приложений i и j было перенесено на новую платформу, компания несет потери $x_{ij} \geq 0$.

Если бы все было настолько просто, ваши знакомые портировали бы все n приложений с получением общей прибыли $\sum_i b_i$. К сожалению, возникает одна проблема...

Из-за мелких, но принципиальных несовместимостей между двумя платформами приложение 1 невозможно портировать; оно неизбежно остается в старой системе. С другой стороны, портирование части других приложений может оказаться экономически оправданным — с учетом всех прибылей и затрат на взаимодействие между приложениями в разных системах.

Итак, какие же приложения следует портировать на новую платформу (если они есть, конечно)? Предложите алгоритм с полиномиальным временем для нахождения множества $S \subseteq \{2, 3, ..., n\}$, для которого сумма прибылей от портирования приложений S на новую платформу с вычетом расходов будет максимальной.

30. Рассмотрим модификацию следующей задачи. В новой ситуации портироваться может любое приложение, но некоторые прибыли b_i при портировании на новую платформу оказываются отрицательными: если $b_i < 0$, то приложение i предпочтительно (на величину, определяемую i) оставить на старой платформе. Предоставьте алгоритм с полиномиальным временем для нахождения множества $S \subseteq \{1, 2, ..., n\}$, для которого сумма прибылей от портирования приложений S на новую платформу с вычетом расходов будет максимальной.

31. Ваши друзья проходят практику в маленькой, но перспективной компании Web-Exodus. Дежурная шутка среди работников этой компании: мощные серверы занимают в компьютерной меньше места, чем пустые коробки от компьютерного оборудования, хранящиеся на тот случай, если что-то придется возвращать поставщику для технического обслуживания.

Несколько дней назад поступила партия мониторов в больших коробках; мониторы разные, поэтому и коробки имеют разные размеры. Работники компании с утра пытаются придумать, как бы их разместить для хранения. Разумеется, для экономии места часть коробок можно хранить *внутри* других коробок.

Предположим, каждая коробка i представляет собой прямоугольный параллелепипед с длинами сторон (i_1, i_2, i_3); каждая длина стороны лежит строго в диапазоне от 0,5 метра до 1 метра. С геометрической точки зрения размещение одной коробки внутри другой означает, что меньшую коробку можно повернуть так, чтобы она помещалась внутри большей коробки по каждому измерению. Формально можно сказать, что коробка i с размерами (i_1, i_2, i_3) помещается внутри коробки j с размерами (j_1, j_2, j_3), если существует перестановка a, b, c измерений $\{1, 2, 3\}$, при которой $i_a < j_1$, $i_b < j_2$ и $i_c < j_3$. Конечно, вложение рекурсивно: если i находится внутри j, а j находится внутри k, то из всех трех коробок видна только коробка k. Мы будем называть *вложением* для множества из n коробок последовательность операций, при которой коробка i размещается внутри коробки j; а если внутри i уже лежат другие коробки, они также оказываются внутри j. (Также обратите внимание на следующее: так как каждая из сторон i имеет длину более 0,5 метра, а каждая из сторон j имеет длину меньше 1 метра, коробка i

занимает более половины размера по каждому измерению j, и после того, как коробка i окажется внутри j, ничего больше в j положить уже не удастся.) Коробка k во вложении называется *видимой*, если в результате последовательности операций она не вкладывается в другую коробку.

А теперь задача, с которой столкнулись в фирме Web-Exodus: так как место занимают только видимые коробки, как выбрать вложение, минимизирующее количество видимых коробок?

Предложите алгоритм с полиномиальным временем для решения этой задачи.

Пример. Имеются три коробки с размерами (0,6, 0,6, 0,6), (0,75, 0,75, 0,75) и (0,9, 0,7, 0,7). Первую коробку можно положить как во вторую, так и в третью; но в любом вложении как вторая, так и третья коротка будут видимы. Следовательно, минимально возможное количество видимых коробок равно 2, а придти к нему можно только одним способом — положив первую коробку во вторую.

32. Для заданного графа $G = (V, E)$ и натурального числа k можно определить отношение $\xrightarrow{G,k}$ для пар вершин G следующим образом: если $x,y \in V$, мы говорим, что $x \xrightarrow{G,k} y$, если в G существуют k путей из x в y, взаимно непересекающихся по ребрам. Можно ли утверждать, что для всех G и всех $k \geq 0$ отношение $\xrightarrow{G,k}$ транзитивно? Иначе говоря, если $x \xrightarrow{G,k} y$ и $y \xrightarrow{G,k} z$, то всегда ли $x \xrightarrow{G,k} z$? Приведите доказательство или контрпример.

33. Имеется направленный граф $G = (V, E)$; предположим, для каждого узла v количество ребер, входящих в v, равно количеству ребер, выходящих из v. Другими словами, для всех v

$$|\{(u, v) : (u, v) \in E\}| = |\{(v, w) : (v, w) \in E\}|$$

Пусть x, y — два узла G, и существуют k путей из x в y, взаимно непересекающихся по ребрам. В такой ситуации можно ли утверждать, что существуют k путей из y в x, взаимно непересекающихся по ребрам? Приведите доказательство или контрпример с объяснением.

34. *Динамические сети*, состоящие из маломощных беспроводных устройств, используются в самых разных ситуациях — например, при стихийных бедствиях, когда координаторам спасательных работ требуется контролировать обстановку в труднодостижимых областях. Предполагается, что партия беспроводных устройств сбрасывается в такую область с самолета, а потом объединяется в действующую сеть.

Обратите внимание: речь идет об (a) относительно недорогих устройствах, которые (b) сбрасываются с самолета в (c) аварийную зону; с учетом всей комбинации факторов необходимо предусмотреть возможность обработки отказов на разумном количестве узлов.

Если одно из устройств v обнаруживает опасность отказа, оно передает представление своего текущего состояния на другое устройство в сети. Каждое устройство обладает ограниченной дальностью передачи — допустим, оно

может взаимодействовать с другими устройствами, находящимися от него на расстоянии не более d метров. Кроме того, состояние не должно передаваться на устройство, на котором уже произошел сбой, поэтому следует предусмотреть некоторую избыточность: устройство v должно располагать множеством k других устройств, с которыми оно может связаться (каждое из которых находится на расстоянии не более d метров). Назовем его *резервным множеством* для устройства v.

(a) Имеется множество из n беспроводных устройств, позиции которых представлены парами координат (x, y). Разработайте алгоритм, который определяет, возможно ли выбрать резервное множество для каждого устройства (то есть k других устройств, каждое из которых удалено не более чем на d метров) с тем дополнительным свойством, что для некоторого параметра b никакое устройство не входит в резервное множества более чем b других устройств. Алгоритм должен выводить сами резервные множества, если их удастся найти.

(b) Представление о том, что для каждой пары устройств v и w существует четкая дихотомия между понятиями «в зоне действия» и «вне зоны действия» — всего лишь упрощенная абстракция. В более точном представлении существует функция снижениям мощности $f(\cdot)$, которая для пары устройств на расстоянии δ определяет мощность сигнала $f(\delta)$ на беспроводном подключении между этими устройствами. (Предполагается, что $f(\delta)$ убывает с возрастанием δ).

Это представление хотелось бы встроить в концепцию резервных множеств: среди k устройств резервного множества v должно быть как минимум одно, с которым возможна связь с очень высокой мощностью сигнала; по крайней мере одно устройство, доступное с умеренно высокой мощностью сигнала, и т. д. А конкретнее, имеются такие значения $p_1 \geq p_2 \geq \ldots \geq p_k$, что если резервное множество v состоит из устройств на расстояниях $d_1 \leq d_2 \leq \ldots \leq d_k$, должно выполняться условие $f(d_j) \geq p_j$ для всех j.

Предложите алгоритм, который определяет, возможно ли выбрать резервное множество для каждого устройства, подчиняющегося уточненному условию; при этом по-прежнему ни одно устройство не должно входить в резервные множества более чем b других устройств. И снова алгоритм должен выводить сами резервные множества, если их удастся найти.

35. Вы разрабатываете интерактивную программу сегментации изображений, которая работает следующим образом: в исходном состоянии используется конфигурация, описанная в разделе 7.10, с n пикселами, множеством соседних пар, параметрами $\{a_i\}$, $\{b_i\}$ и $\{p_{ij}\}$. Относительно этого экземпляра делаются два предположения. Во-первых, каждый из параметров $\{a_i\}$, $\{b_i\}$ и $\{p_{ij}\}$ является неотрицательным целым числом в диапазоне от 0 до d для некоторого числа d. Во-вторых, предполагается, что соседские отношения между пикселами обладают тем свойством, что каждый пиксел является соседом не более чем четырех других пикселов (в полученном графе из каждого узла выходит не более четырех ребер).

Сначала выполняется исходная сегментация (A_0, B_0), максимизирующая величину $q(A_0, B_0)$. В результате исходной сегментации некоторые пикселы могут быть отнесены к фону, хотя пользователь знает, что они должны принадлежать переднему плану. Увидев эту сегментацию на экране, пользователь может щелкнуть мышью на конкретном пикселе v_1, переводя его на передний план. Но программа не ограничивается простым переносом пиксела: вместо этого она должна вычислять сегментацию (A_1, B_1), максимизирующую величину $q(A_1, B_1)$ с учетом того, что v_1 принадлежит переднему плану. (Как это может пригодиться на практике? Допустим, при сегментации фотографии группы людей кто-то держит сумку, которая была ошибочно отнесена к фону. Если щелкнуть на одном пикселе, принадлежащем сумке, и пересчитать оптимальную сегментацию с учетом нового условия, вся сумка станет частью переднего плана).

Система должна дать возможность пользователю провести серию таких щелчков $v_1, v_2, ..., v_t$; после щелчка v_i система проводит сегментацию (A_i, B_i), максимизирующую величину $q(A_i, B_i)$ в соответствии с условием, что $v_1, v_2, ..., v_i$ находятся на переднем плане.

Предложите алгоритм, выполняющий все эти операции так, что исходная сегментаця вычисляется за время вычисления одного максимального потока, умноженное на константу, а взаимодействие с пользователем обрабатывается за время $O(dn)$ на каждый щелчок.

(Примечание: Упражнение с решением 1 из этой главы служит полезным примитивом для этого упражнения. Кроме того, симметричная операция перевода пиксела в фон выполняется аналогично, но здесь вам это делать не нужно.)

36. Рассмотрим еще одну разновидность задачи сегментации изображений из раздела 7.10. Мы разработаем решение задачи разметки изображения, в которой каждый пиксел помечается приблизительной оценкой расстояния от него до камеры (вместо простой разметки «фон/передний план», использованной в тексте). Допустимые значения меток для каждого пиксела равны 0, 1, 2, ..., M для некоторого целого M.

Пусть $G = (V, E)$ — граф, узлы которого соответствуют пикселам, а ребра обозначают соседние пары пикселов. *Разметкой* пикселов называется разбиение V на множества $A_0, A_1, ..., A_M$, где A_k — множество пикселов, помечаемых расстоянием k для $k = 0, ..., M$. Мы займемся поиском разметки с минимальной стоимостью; стоимость определяется двумя факторами. По аналогии с задачей сегментации «передний план/фон», имеется стоимость назначения: для каждого пиксела i и метки k стоимость $a_{i,k}$ определяет стоимость назначения метки k пикселу i. Затем, если двум соседним пикселам $(i, j) \in E$ назначаются разные метки, также действует штраф за разделение. В разделе 7.10 использовался штраф за разделение p_{ij}; в текущей задаче штраф за разделение также зависит от того, на какое расстояние удаляются пикселы: он пропорционален разности значений двух меток. Таким образом, общая стоимость q' разметки определяется следующим образом:

$$q'(A_0, ..., A_M) = \sum_{k=0}^{M} \sum_{i \in A_i} a_{i,k} + \sum_{k<\ell} \sum_{\substack{(i,j) \in E \\ i \in A_k, j \in A_\ell}} (\ell - k) p_{ij}.$$

Целью этой задачи является разработка алгоритма с полиномиальным временем, находящего оптимальную разметку для заданного графа G и параметров штрафов $a_{i,k}$ и p_{ij}. Алгоритм будет основан на построении потока в сети; чтобы упростить вашу задачу, мы приведем часть этой схемы

У потоковой сети имеется источник s и сток t. Кроме того, для каждого пиксела $i \in V$ в потоковую сеть включаются узлы $v_{i,k}$ для $k = 1, ..., M$, как показано на рис. 7.30 (для $M = 5$).

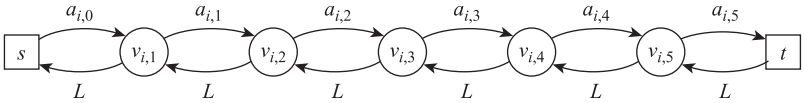


Рис. 7.30. Множество узлов, соответствующих одному пикселу i в упражнении 36 (с источником s и стоком t)

Для удобства записи узлы $v_{i,0}$ и $v_{i,M+1}$ соответствуют s и t соответственно для всех $i \in V$.

Теперь добавим ребра $(v_{i,k}, v_{i,k+1})$ с пропускной способностью $a_{i,k}$ для $k = 0, ..., M$; а также ребра $(v_{i,k+1}, v_{i,k})$ в противоположном направлении с очень большой пропускной способностью L. Этот набор узлов и ребер будет называться *цепью*, связанной с пикселом i.

Обратите внимание: если сделать эту «очень большую» пропускную способность L достаточно большой, то не будет такого минимального разреза (A, B), для которого ребро с пропускной способностью L выходит из множества A. (Насколько большой должна быть эта пропускная способность, чтобы это произошло?) Однако для любого минимального разреза (A, B) и каждого пиксела i в цепи, связанной с i, будет ровно одно ребро с малой пропускной способностью, выходящее из множества A. (Убедитесь в том, что если бы нашлось два таких ребра, то ребро с большой пропускной способностью также должно выходить из множества A.)

И наконец, вопрос: используйте узлы и ребра, определенные выше, для построения потоковой сети, позволяющей эффективно вычислить разметку с минимальной стоимостью по минимальному разрезу s–t. Также докажите, что ваше построение обладает желаемым свойством, и покажите, как восстановить разметку минимальной стоимости по разрезу.

37. В стандартной задаче о минимальном разрезе s–t предполагается, что все пропускные способности неотрицательны; с произвольными сочетаниями положительных и отрицательных пропускных способностей вычислительная сложность задачи заметно усложняется. Однако, как вы сейчас убедитесь, требование неотрицательности можно немного смягчить — и при этом задача по-прежнему будет решаться за полиномиальное время.

Имеется направленный граф $G = (V, E)$ с источником $s \in V$, стоком $t \in V$ и пропускными способностями ребер $\{c_e\}$. Предположим, для каждого ребра e, у которого ни s, ни t не являются конечной точкой, выполняется условие $c_e \geq 0$.

Таким образом, c_e может быть отрицательным для ребер e, у которых по крайней мере одним концом является s или t. Предложите алгоритм с полиномиальным временем выполнения для нахождения разреза s–t минимальной величины в таком графе. (Несмотря на новое требования неотрицательности, величина разреза s–t (A, B) по-прежнему определяется как сумма пропускных способностей всех ребер e, для которых начальный узел e принадлежит A, а конечный узел принадлежит B.)

38. Вы работаете с большой базой данных работников. В контексте задачи база данных представляется в виде двумерной таблицы T, состоящей из m строк (множество R) и n столбцов (множество C); строки соответствуют конкретным работникам, а столбцы — разным атрибутам.

    Возьмем простой пример: таблица содержит четыре столбца (имя, телефон, дата приема, имя руководителя), и в ней хранятся данные пяти работников.

name	phone number	start date	manager's name
Alanis	3-4563	6/13/95	Chelsea
Chelsea	3-2341	1/20/93	Lou
Elrond	3-2345	12/19/01	Chelsea
Hal	3-9000	1/12/97	Chelsea
Raj	3-3453	7/1/96	Chelsea

    Для заданного подмножества S столбцов можно построить новую, уменьшенную таблицу, которая содержит только столбцы из S. Назовем эту новую таблицу *проекцией* T на S, и обозначим ее T [S]. Например, если S = {`name`, `start date`}, то проекцией T [S] будет таблица, состоящая только из первого и третьего столбцов.

    Другая полезная операция, часто встречающаяся при работе с таблицами, — *перестановка столбцов*. Для заданной перестановки p новая таблица того же размера, что и T, строится простым переупорядочением столбцов в соответствии с p. Новая таблица называется перестановкой T относительно p и обозначается T_p.

    Имеются k разных подмножеств столбцов S_1, S_2, ..., S_k; вы собираетесь много работать с ними и поэтому хотите иметь их в удобном легкодоступном формате. Можно сохранить k проекций T [S_1], T [S_2], ..., T [S_k], но они займут слишком много места. Рассматривая возможные альтернативы, вы узнаете, что явное проецирование на каждое подмножество не обязательно, потому что используемая СУБД позволяет особенно эффективно работать со столбцами, если (в некотором порядке) элементы подмножества образуют префикс семейства столбцов в порядке «слева направо». Например, в нашем примере подмножества {`name`, `phone number`} и {`name`, `start date`, `phone number`,} образуют префиксы (первые два и первые три столбца слева соответственно); и это позволяет обрабатывать в этой таблице их намного эффективнее, чем подмножество {`name`, `start date`}, не образующее префикс. (Еще раз: для заданного подмножества S_i не определен конкретный порядок, и нас интересует, существует ли *некоторый* порядок, при котором оно образует префикс.)

А теперь вопрос: для заданного параметра $\ell < k$ возможно ли найти ℓ перестановок столбцов $p_1, p_2, ..., p_\ell$, чтобы для каждого из заданных подмножеств S_i (для $i = 1, 2, ..., k$) столбцы S_i образовали префикс по крайней мере для одной из перестановок столбцов таблицы $T_{p_1}, T_{p_2}, ..., T_{p_\ell}$? Такое множество перестановок будет называться *действительным решением* задачи; если действительное решение существует, это означает, что вам достаточно хранить всего ℓ перестановок таблиц вместо всех k проекций. Предложите алгоритм с полиномиальным временем для решения этой задачи; для экземпляров, для которых существует действительное решение, ваш алгоритм должен возвращать действительное множество из ℓ перестановок.

Пример. Для приведенной выше таблицы заданы подмножества

$$S_1 = \{\texttt{name}, \texttt{phone number}\},$$

$$S_2 = \{\texttt{name}, \texttt{start date}\},$$

$$S_3 = \{\texttt{name}, \texttt{manager's name}, \texttt{start date}\},$$

и $\ell = 2$. В этом случае действительное решение существует и достигается двумя перестановками

$$p_1 = \{\texttt{name}, \texttt{phone number}, \texttt{start date}, \texttt{manager's name}\},$$

$$p_2 = \{\texttt{name}, \texttt{start date}, \texttt{manager's name}, \texttt{phone number}\}.$$

В этом случае S_1 образует префикс для перестановочной таблицы T_{p_1}, а S_2 и S_3 образуют префиксы для перестановочной таблицы T_{p_2}.

39. Вы консультируете фирму, занимающуюся сбором статистики о параметрах окружающей среды. Фирма собирает статистику и публикует собранные данные в книге. Статистика относится к населению разных регионов и сохраняется в миллионах. Пример такой статистики приведен в следующей таблице.

Страна	**A**	**B**	**C**	**Итого**
Взрослые мужчины	11,998	9,083	2,919	24,000
Взрослые женщины	12,983	10,872	3,145	27,000
Дети	1,019	2,045	0,936	4,000
Итого	26,000	22,000	7,000	55,000

Для простоты будем считать, что суммы по строкам и столбцам являются целыми числами. *Задача округления результатов переписи* заключается в округлении всех данных до целых чисел без изменения сумм столбцов и строк. Каждое дробное число может округляться либо в большую, либо в меньшую сторону. Например, хорошее округление данных таблицы может выглядеть так:

Страна	A	B	C	Итого
Взрослые мужчины	11,000	10,000	3,000	24,000
Взрослые женщины	13,000	10,000	4,000	27,000
Дети	2,000	2,000	0,000	4,000
Итого	26,000	22,000	7,000	55,000

(a) Сначала рассмотрите частный случай, когда все данные лежат в диапазоне от 0 до 1. В этом случае имеется матрица дробных чисел от 0 до 1, а задача заключается в округлении каждой дроби до 0 или 1 без изменения сумм строк и столбцов. Для проверки возможности такого округления используйте метод вычисления потока.

(b) Рассмотрите задачу округления результатов переписи в том виде, в котором она определена выше: суммы строк и столбцов являются целыми числами, каждое дробное число α требуется округлить до $\lfloor \alpha \rfloor$ или $\lceil \alpha \rceil$. Для проверки возможности такого округления используйте метод вычисления потока.

(c) Докажите, что округление, которые мы ищем в (a) и (b), существует всегда.

40. В во многих вычислительных задачах можно поставить вопрос о «стабильности» или «надежности» ответа. Подобные вопросы можно задать и о комбинаторных задачах; это один из способов формулировки задачи о минимальном остовном дереве.

Имеется граф $G = (V, E)$ со стоимостью c_e каждого ребра e. Стоимости рассматриваются как величины, которые измерялись экспериментально, и могут содержать возможные ошибки. Следовательно, минимальное остовное дерево, вычисленное для G, может не совпадать с «настоящим» минимальным остовным деревом.

Для заданных параметров $\varepsilon > 0$ и $k > 0$, а также конкретного ребра $e' = (u, v)$, хотелось бы выдвинуть утверждение следующего вида.

(*) Даже если стоимость каждого ребра изменится не более чем на ε (с увеличением или уменьшением), а стоимости k ребер, отличных от e', дополнительно заменятся разными произвольными значениями, ребро e' все равно не будет принадлежать никакому минимальному остовному дереву G.

Такое свойство предоставляет своего рода гарантию того, что вхождение e' в минимальное остовное дерево маловероятно, даже если предположить значительные ошибки измерений.

Предложите алгоритм с полиномиальным временем, который получает G, e', ε и k, и решает, выполняется ли свойство (*) для e'.

41. Допустим, вы управляете группой процессоров и планируете выполнение серии заданий. Задания обладают следующими характеристиками: у каждого задания имеется время поступления a_j, когда оно становится доступным для обработки;

длина ℓ_j, которая показывает, какое время потребуется для его выполнения; и предельное время d_j, к которому оно должно быть завершено. Будем считать, что $0 < \ell_j \leq d_j - a_j$. Каждое задание может выполняться на любых процессорах, но только на одном в любой момент времени; задание может быть приостановлено, а потом продолжено с точки приостановки (возможно, по прошествии некоторого времени) на другом процессоре.

Кроме того, группа процессоров не является полностью статичной: общий пул состоит из k возможных процессоров, но для каждого процессора i существует интервал времени $[t_i, t'_i]$, во время которого он является доступным; во все остальное время процессор недоступен.

Требуется решить, удастся ли своевременно выполнить все задания или нет, с учетом всех требований к заданиям и данных о доступности процессоров. Предложите алгоритм с полиномиальным временем выполнения, который либо строит расписание, завершающее все задания к предельным срокам, либо сообщает (обоснованно) о том, что такого расписания не существует. Считайте, что все параметры, связанные с задачей, являются целыми числами.

Пример. Имеются два задания J_1 и J_2. Задание J_1 поступает во время 0, должно быть завершено ко времени 4, и имеет длину 3. Задание J_2 поступает во время 1, должно быть завершено к времени 3 и имеет длину 2. Также доступны два процессора P_1 и P_2. Процессор P_1 свободен в интервале времени от 0 до 4; процессор P_2 доступен в интервале от 2 до 3. В данном случае существует расписание, позволяющее вовремя выполнить оба задания.

- Во время 0 задание J_1 запускается на процессоре P_1.
- Во время 1 задание J_1 приостанавливается, и на P_1 запускается задание J_2.
- Во время 2 выполнение J_1 продолжается на процессоре P_2. (J_2 продолжает выполняться на P_1.)
- Во время 3 задание J_2 завершается к своему предельному времени. Процессор P_2 становится недоступным, поэтому задание J_1 перемещается обратно на P_1 для выполнения завершающего интервала обработки.
- Во время 4 задание J_1 завершается на процессоре P_1.

Обратите внимание: решения, в котором бы не использовались приостановка и перемещение заданий, не существует.

42. Предложите алгоритм с полиномиальным временем выполнения для следующей минимизирующей аналогии задачи максимального потока: Имеется направленный граф $G = (V, E)$ с источником $s \in V$, стоком $t \in V$ и числами (пропускными способностями) $\ell(v, w)$ для каждого ребра $(v, w) \in E$. Мы определяем поток f; как обычно, величина потока требует, чтобы все ребра, кроме s и t, удовлетворяли ограничению сохранения потока. Однако заданные числа определяют нижние границы потока через ребро — то есть они требуют, чтобы $f(v, w) \geq \ell(v, w)$ для каждого ребра $(v, w) \in E$, а верхняя граница для величины потока по ребрам отсутствует.

(a) Предложите алгоритм с полиномиальным временем выполнения, который находит действительный поток с минимальной возможной величиной.

(b) Докажите аналогию теоремы о максимальном потоке и минимальном разрезе для этой задачи (то есть равен ли минимальный поток максимальному разрезу?)

43. Вы пытаетесь решить задачу циркуляции, но она не является полностью корректной. В задаче установлены уровни потребления, но отсутствуют ограничения пропускной способности для ребер. Или, говоря более формально, существуют граф $G = (V, E)$ и уровни потребления d_v для каждого узла v (удовлетворяющие условию $\sum_{v \in V} d_v = 0$; требуется решить, существует ли поток f, для которого $f(e) \geq 0$ и $f^{in}(v) - f^{out}(v) = d_v$ для всех узлов $v \in V$. Обратите внимание: эта задача может быть решена при помощи алгоритма циркуляции из раздела 7.7, с присваиванием $c_e = +\infty$ для всех ребер $e \in E$ (или присваиванием c_e чрезвычайно большого числа для каждого ребра — допустим, больше суммы всех положительных уровней потребления d_v в графе).

    Требуется исправить граф так, чтобы задача стала корректной, поэтому было бы очень полезно узнать, почему задача некорректна в ее текущей формулировке. При ближайшем рассмотрении вы видите, что существует такое подмножество U узлов, что не существует ребра, входящего в U, но при этом $\sum_{v \in U} d_v > 0$. Вы быстро понимаете, что из существования такого множества немедленно следует невозможность существования потока: множество U имеет положительное общее потребление, поэтому ему необходим входящий поток, но в U не входят ребра. Пытаясь понять, насколько эта задача далека от разрешимости, вы задаетесь вопросом, насколько большим может быть потребление множества без входящих ребер.

    Предложите алгоритм с полиномиальным временем для нахождения подмножества $S \subset V$ узлов, для которых не существует ребра, входящего в S, и для которого сумма $\sum_{v \in S} d_v$ будет максимально возможной с учетом этого условия.

44. Имеется направленная сеть $G = (V, E)$ с корневым узлом r и множеством терминальных узлов $T \subseteq V$. Нам хотелось бы отсоединить от r как можно больше терминальных узлов, но с разрывом относительно небольшого количества ребер.

    Этот компромисс будет достигаться следующим образом. Для множества ребер $F \subseteq E$ $q(F)$ обозначает количество узлов $v \in T$, для которых не существует пути r-v в подграфе $(V, E–F)$. Предложите алгоритм с полиномиальным временем для нахождения множества F ребер, максимизирующего величину $q(F) - |F|$. (Обратите внимание: множество F может быть пустым).

45. Рассмотрим следующее определение. Имеется множество из n стран, участвующих во взаимной торговле. Для каждой страны имеется значение s_i ее бюджетного профицита; это число может быть как положительным, так и отрицательным (отрицательное число указывает на дефицит бюджета). Для каждой пары стран i,j известна общая величина e_{ij} всего экспорта из i в j; это число

всегда неотрицательно. Подмножество стран S называется *самостоятельным*, если сумма бюджетных профицитов стран S за вычетом общей величины всего экспорта из стран S в страны, в S не входящие, не отрицательна.

Предложите алгоритм с полиномиальным временем, который получает данные по множеству из n стран и решает, содержит ли оно непустое самостоятельное подмножество, не совпадающее с полным множеством.

46. В социологии часто встречаются графы G, узлы которых представляют людей, а ребра — дружеские отношения между ними. Будем считать, что дружеские отношения симметричны, поэтому граф можно считать ненаправленным.

    А теперь предположим, что мы ищем по графу G «тесно связанную» группу людей. Один из способов формализации этого понятия выглядит так: для подмножества S узлов $e(S)$ обозначает количество ребер в S — то есть количество ребер, оба конца которых принадлежат S. Определим величину *сцепления* S как $e(S)/|S|$. Естественной целью поиска становится множество S людей, для которых достигается максимальное сцепление.

    (a) Предложите алгоритм с полиномиальным временем, который получает рациональное число α и определяет, существуте ли множество S со сцеплением не ниже α.

    (b) Предложите алгоритм с полиномиальным временем, который находит множество S узлов с максимальным сцеплением.

47. Цель этой задачи — изучение разновидностей алгоритма проталкивания предпотока, ускоряющих практическое время выполнения без вреда для сложности в худшем случае. Вспомните, что алгоритм поддерживает свой инвариант: $h(v) \leq h(w) + 1$ для всех ребер (v, w) в остаточном графе текущего предпотока. Мы доказали, что если f является потоком (а не предпотоком) с этим инвариантом, то этот поток максимален. Высоты возрастали монотонно, а анализ времени выполнения был основан на ограничении возможного количества изменений высот у ребер. Практический опыт показывает,ч то алгоритм почти всегда работает намного быстрее, чем предполагает худший случай, а реальным «узким местом» алгоритма является изменение меток узлов (а не ненасыщающие проталкивания, которые ведут к худшему случаю в теоретическом анализе). Целью приведенной ниже задачи является снижение количества изменений меток посредством увеличения высоты на величину, большую 1. Допустим, имеется граф G с n узлами, m ребрами, пропускными способностями c, источником s и стоком t.

    (a) Алгоритм проталкивания предпотока, описанный в разделе 7.4, в начале своей работы назначает поток, равный пропускной способности c_e всех ребер e, выходящих из источника; назначает поток 0 по всем остальным ребрам; присваивает $h(s) = n$, и $h(v) = 0$ для всех остальных узлов $v \in V$. Предложите процедуру $O(m)$ для инициализации высот узлов, которая была бы лучше процедуры из раздела 7.4. Ваш метод должен назначать каждому узлу v высоту, максимально возможную для исходного потока.

(b) В этой части мы добавим в алгоритм проталкивания предпотока новый шаг — *интервальное изменение меток*, в ходе которого метки многих узлов будут увеличиваться более чем на 1. Рассмотрим предпоток f и высоты h, удовлетворяющие инварианту. *Пропуском* в системе высот называется такое целое число $0 < h < n$, что не существует узла с высотой, равной h. Обозначим A множество узлов v с высотами $n > h(v) > h$. Изменением меток пропусков называется процесс изменения высот всех узлов в A так, чтобы они были равны n. Докажите, что алгоритм проталкивания предпотока с интервальным изменением меток является действительным алгоритмом максимального потока. Заметьте, что единственный новый факт, который нужно доказать — что интервальное изменение меток сохраняет приведенный выше инвариант: $h(v) \leq h(w) + 1$ для всех ребер (v, w) в остаточном графе.

(c) В разделе 7.4 мы доказали, что $h(v) \leq 2n - 1$ на протяжении алгоритма. В этой модификации выполняется условие $h(v) \leq n$. Идея заключается в том, что мы «замораживаем» все узлы при достижении высоты n; другими словами, узлы на высоте n перестают считаться активными, и поэтому не используются для операций *push* и *relabel*. В этом случае в конце алгоритма мы имеем функцию предпотока и высоты, которая удовлетворяет приведенному выше инварианту, так что весь избыточный поток находится на высоте n. Пусть B — множество узлов v, для которых существует путь от v к t в остаточном графе текущего предпотока, а $A = V - B$. Докажите, что в конце алгоритма (A, B) является разрезом s–t с минимальной пропускной способностью.

(d) Алгоритм в части (c) вычисляет минимальный разрез s–t, но не находит максимальный поток (так как завершается на предпотоке, содержащем избыточные потоки). Предложите алгоритм, который получает предпоток f в конце алгоритма из части (c), и преобразует его в максимальный поток за время, не превышающее $O(mn)$. (Подсказка: рассмотрите узлы с избыточным потоком, и попробуйте отправить избыточный поток обратно к s, используя только те ребра, по которым поток поступил.)

48. В разделе 7.4 мы рассмотрели алгоритм проталкивания предпотока, а также обсудили одно конкретное правило выбора вершин. Сейчас будет рассмотрено другое правило выбора. Мы также рассмотрим варианты алгоритма с ранним завершением (и наайдем разрез, близкий к минимально возможному).

(a) Пусть f — любой предпоток. Так как f не обязательно является действительным потоком, может оказаться, что величина $f^{\text{out}}(s)$ намного выше величины максимального потока в G. Покажите, что $f^{\text{in}}(t)$ является нижней границей для величины максимального потока.

(b) Рассмотрим предпоток f и совместимую разметку h. Вспомните, что множества $A = \{v : \textit{существует путь } s\textit{–}v \textit{ в остаточном графе } G_f\}$ и $B = V - A$ определяют разрез s–t для любого предпотока f, имеющего совместимую разметку h. Покажите, что пропускная способность разреза (A, B) равна $c(A, B) = \sum_{v \in B} e_f(v)$.

Объединение (a) и (b) позволяет алгоритму завершиться на ранней стадии и вернуть (A, B) как разрез с «приближенно-минимальной» пропускной способностью в предположении, что значение $c(A, B) - f^{\text{in}}(t)$ достаточно мало. Далее мы рассмотрим реализацию, которая стремится сократить это значение, пытаясь протолкнуть поток из узлов с большим количеством избыточного потока.

(c) Версия алгоритма проталкивания предпотока с масштабированием поддерживает параметр масштабирования Δ. Изначально Δ присваивается большая степень 2. На каждом шаге алгоритм выбирает узел, у которого избыточный поток не менее Δ, при минимально возможной высоте. Если узлов (кроме t) с избыточным потоком не менее Δ не осталось, алгоритм делит Δ на 2 и продолжает. Обратите внимание: этот алгоритм является действительной реализацией обобщенного алгоритма проталкивания предпотока. Алгоритм выполняется по фазам. Одна фаза продолжается до тех пор, пока значение Δ остается неизменным. В начале алгоритма Δ имеет максимальную пропускную способность, а при его завершении $\Delta = 1$. Следовательно, в ходе выполнения количество фаз масштабирования не превышает $O(\log C)$. Покажите, как реализовать эту разновидность алгоритма так, чтобы время выполнения было ограничено $O(mn + n \log C + K)$, если алгоритм использует K ненасыщающих операций `push`.

(d) Покажите, что количество ненасыщающих операций `push` в предыдущем алгоритме не превышает $O(n^2 \log C)$. Вспомните, что количество фаз масштабирования имеет границу $O(\log C)$. Чтобы определить границу количества ненасыщающих операций `push` в одной фазе масштабирования, рассмотрите потенциальную функцию $\Phi = \sum_{v \in V} h(v)e_f(v)/\Delta$. Как ненасыщающая операция `push` влияет на Φ? Какие операции могут привести к увеличению Φ?

49. Рассмотрим задачу о распределении: имеется множество из n станций, способных выполнять некую работу, и множество из k заявок на обслуживание (например, станции — вышки сотовой связи, а запросы — сотовые телефоны). Каждый запрос может обслуживаться заданным множеством станций. Задача в таком виде может быть представлена двудольным графом G: одна сторона для станций, другая для клиентов, и клиент x соединяется со станцией y ребром (x, y), если клиент x может обслуживаться станцией y. Будем считать, что каждая станция способна обслуживать не более одного клиента. Используя метод вычисления максимального потока, можно решить, возможно или нет обслужить всех клиентов, или же получить распределение подмножества клиентов по станциям, максимизирующее количество обслуженных клиентов.

    В этой задаче рассматривается разновидность задачи с дополнительным усложнением: клиенты предлагают разные суммы денег за обслуживание. Пусть U — множество клиентов, а клиент $x \in U$ готов заплатить $v_x \geq 0$. Требуется найти подмножество $X \subset U$, максимизирующее $\sum_{x \in X} v_x$ — такое, что существует распределение клиентов из X между станциями.

Рассмотрим следующий жадный метод: клиенты обрабатываются в порядке убывания «ценности» (с произвольной разбивкой «ничьих»). При рассмотрении клиента x алгоритм либо «обещает» обслуживание x, либо отвергает x по следующему жадному правилу: x добавляется в множество X в том и только в том случае, если существует распределение $X \cup \{x\}$ по станциям; в противном случае x отвергается. Помните, что отвергнутые клиенты не будут рассматриваться позднее. (Если уж приходится отвергнуть высокооплачиваемого клиента, по крайней мере ему можно сказать об этом сразу.) При этом распределение принятых клиентов по обслуживающим станциям не осуществляется по жадному принципу: распределение фиксируется только после фиксации множества принятых клиентов. Приведет ли этот жадный подход к оптимальному множеству клиентов? Докажите, что это так, или приведите контрпример.

50. Рассмотрим следующую задачу планирования. Имеются m машин, каждая из которых может обрабатывать задания. Требуется распределить задания между машинами (каждое задание должно быть назначено ровно одной машине) и упорядочить задания на машинах так, чтобы минимизировать функцию стоимости. Машины работают с разной скоростью, но потребности в вычислительных ресурсах у всех заданий одинаковы. В более формальном выражении с каждой машиной i связывается параметр ℓ_i, и выполнение каждого задания потребует времени ℓ_i, если оно будет назначено на машину i.

    Количество заданий равно n; все задания требуют одинаковых затрат вычислительных ресурсов, но имеют разные уровни срочности. Для каждого задания j определена функция стоимости $c_j(t)$, которая определяет стоимость выполнения задания j за время t. Предполагается, что стоимости неотрицательны и монотонны по t.

    Расписание представляет собой распределение заданий по машинам, и для каждой машины оно определяет порядок выполнения заданий. Задание, назначенное на машину i первым, завершается за время ℓ_i, второе задание — за время ℓ_i и т. д. Пусть для расписания S время завершения задания j в этом расписании обозначается $t_S(j)$. Стоимость такого расписания равна $\cos t(S) = \sum_j c_j(t_S(j))$.

    Предложите алгоритм с полиномиальным временем для нахождения расписания с минимальной стоимостью.

51. Вашим друзьям надоела игра «Шесть шагов до Кевина Бэйкона»[1] (в конце концов, разве это не обычный поиск в ширину?), и они решили изобрести нечто более содержательное с алгоритмической точки зрения. Вот как работает их система.

    Все начинается с множества X из n актрис, множества Y из n актеров, и двух игроков P_0 и P_1. Игрок P_0 называет актрису $x_1 \in X$; игрок P_1 называет актера y_1, снимавшегося в одном фильме с x_1; игрок P_0 называет актрису x_2, снимавшуюся в одном фильме с y_1, и т. д. Таким образом, P_0 и P_1 совместно строят последовательность $x_1, y_1, x_2, y_2, \ldots$, в которой каждый актер/актриса снимается в одном

[1] См. https://ru.wikipedia.org/wiki/Шесть_шагов_до_Кевина_Бэйкона. — *Примеч. пер.*

фильме со своим непосредственным предшественником. Игрок P_i ($i = 0,1$) проигрывает, если в свой ход он не может назвать ни один элемент своего множества, не называвшийся ранее.

Допустим, имеется конкретная пара таких множеств X и Y, с полной информацией: кто, с кем, в каких фильмах снимался. *Стратегией* для игрока P_i в нашей задаче называется алгоритм, который получает текущую последовательность x_1, y_1, x_2, y_2, ... и генерирует действительный следующий ход для P_i (предполагается, что сейчас ход P_i). Предложите алгоритм с полиномиальным временем, который решает, какой из двух игроков может обеспечить себе победу в конкретном экземпляре игры.

Примечания и дополнительная литература

Концепция сетевых потоков впервые была сформулирована в целостном виде в работе Форда и Фалкерсона (Ford, Fulkerson, 1962). Сейчас сетевые потоки стали самостоятельной областью исследований, по которой можно легко прочитать отдельный учебный курс; например, дополнительную информацию можно найти в обзоре Голдберга, Тардоса и Тарьяна (Goldberg, Tardos, Tarjan, 1990) и книге Ахуджа, Маньянти и Орлина (Ahuja, Magnanti, Orlin, 1993).

Шрайвер (Schrijver, 2002) предоставляет интересную историческую информацию о начальной стадии работы Форда и Фалкерсона над задачей потока. Те из нас, кому всегда казалось, что в задаче минимального разреза присутствует некий деструктивный оттенок, найдут в этой работе подтверждение своей позиции: в ней цитируется недавно рассекреченный отчет ВВС США, из которого следует, что изначально задача нахождения минимального разреза формулировалась для железнодорожной сети Советского Союза, а целью была дезорганизация перевозок.

Как упоминается в тексте, задачи о двудольном паросочетании и непересекающихся путях были сформулированы за несколько десятилетий до задачи о максимальном потоке; только разработка теории сетевых потоков поставила все эти задачи на общую методологическую основу. У теории паросочетаний в двудольных графах было много независимых первооткрывателей; вероятно, чаще других упоминаются П. Холл (Hall, 1935) и Кёниг (König, 1916). Задача нахождения путей, непересекающихся по ребрам, от источника к стоку, эквивалентна задаче о максимальном потоке с пропускными способностями, равных 1; этот частный случай был разрешен (фактически в эквивалентной форме) Менгером (Menger, 1927).

Алгоритм проталкивания предпотока был разработан Голдбергом (Goldberg, 1986), а его эффективная реализация разработана Голдбергом и Тарьяном (Goldberg, Tarjan, 1986). Высокопроизводительный код для этого и других алгоритмов сетевого потока можно найти на сайте Эндрю Голдберга.

Алгоритм сегментации изображений с использованием минимальных разрезов был разработан Грейгом, Портеусом и Сехолтом (Greig, Porteous, Seheult, 1989), а применение минимальных разрезов стало играть активную роль в области обработки изображений (например, см. Векслер (Veksler, 1999), Колмогоров и Забих

(Kolmogorov, Zahir, 2004)); расширения этого подхода обсуждаются в главе 12. Уэйн (Wayne, 2001) приводит дополнительную информацию о выбывании бейсбольных команд, а также приписывает популяризацию этой задачи в 1960-х годах Алану Хоффману. Другие практические применения сетевых потоков и разрезов рассматриваются в книге Ахуджа, Маньянти и Орлина (Ahuja, Magnati, Orlin, 1993).

Задача нахождения идеального паросочетания с минимальной стоимостью является частным случаем задачи о потоке с минимальной стоимостью, выходящей за рамки материала. Существует несколько эквивалентных формулировок задачи о потоке с минимальной стоимостью; в одной из формулировок дается потоковая сеть с пропускными способностями c_e и стоимостями C_e ребер; стоимость потока f равна сумме стоимостей ребер, взвешенной по величине передаваемого по ним потока, $\sum_e C_e f(e)$, а целью является построение максимального потока с минимальной общей стоимостью. Задача нахождения потока с минимальной стоимостью решается за полиномиальное время, и тоже имеет множество практических применений; алгоритмы для этой задачи обсуждаются в книгах Кука и др. (Cook et al., 1998) и Ахуджа, Маньянти и Орлина (Ahuja, Magnati, Orlin, 1993).

Поток в сети моделирует задачи маршрутизации, которые могут быть сведены к задаче построения разных путей от одного источника к одном стоку, существует более общий (и более общий) класс задач маршрутизации, в котором пути могут одновременно строиться между разными парами отправителей и получателей. Отношения между этими классами задач весьма нетривиальны; эта проблема, а также алгоритмы для более сложных видов задач маршрутизации обсуждаются в главе 11.

Примечания к упражнениям

Упражнение 8 создано на основе задачи, о которой мы узнали от Боба Блэнда; упражнение 16 основано на обсуждении с Уди Манбером; упражнение 26 основано на обсуждении с Джорданом Эренрихом; упражнение 35 основано на обсуждении с Юрием Бойковвым, Ольгой Векслер и Рамином Забихом; упражнение 36 основано на результатах Хироси Исикавы и Дэви Гейгера, а также Бойкова, Векслер и Забиха; упражнение 38 основано на задаче, от которой нам рассказал Эл Демерс; упражнение 46 основано на результатах Пикара и Ратлиффа.

Глава 8
NP-полнота и вычислительная неразрешимость

Мы подошли к одной из важнейших переходных точек в книге. До сих пор мы разрабатывали эффективные алгоритмы для самых разнообразных задач и даже добились некоторых результатов в неформальной классификации задач, имеющих эффективные решения, — например, задач, которые могут быть выражены минимальными разрезами графа, или задачам, которые формулируются средствами динамического программирования. Но хотя при этом мы нередко отмечали другие задачи, для которых не могли предложить решения, никакие попытки реальной классификации или описания категорий задач, не имеющих эффективного решения, не предпринимались.

В самом начале книги, когда приводились первые основополагающие определения, мы договорились о том, что нашим рабочим критерием эффективности будет полиномиальное время выполнения. Как упоминалось ранее, у такого конкретного определения есть одно важное преимущество: оно позволяет привести математическое доказательство того, что некоторые задачи не могут быть решены при помощи алгоритмов с полиномиальным временем (то есть «эффективных»).

Когда ученые занялись серьезными исследованиями вычислительной сложности, им удалось сделать первые шаги к доказательству того, что эффективных алгоритмов для решения некоторых особенно сложных задач не существует. Но для многих фундаментальных дискретных вычислительных задач (из области оптимизации, искусственного интеллекта, комбинаторики, логики и т. д.) этот вопрос оказался слишком сложным и с тех пор так и остается открытым: алгоритмы с полиномиальным временем для таких задач неизвестны, но и доказательств того, что таких алгоритмов не существует, пока нет.

В свете этой формальной неоднозначности, которая только укреплялась с течением времени, в области изучения вычислительной сложности был достигнут заметный прогресс. Ученым удалось охарактеризовать широкий спектр задач в этой «серой зоне» и доказать их эквивалентность в следующем смысле: если для одной из таких задач будет найден алгоритм с полиномиальным временем выполнения, это будет означать существование алгоритма для всех таких задач. Такие задачи

называются *NP-полными,* — смысл этого термина вскоре станет более понятен. Существуют буквально тысячи *NP*-полных задач в самых разных областях; похоже, этот класс содержит значительную долю фундаментальных задач, сложность которых не поддается разрешению. Формулировка *NP*-полноты и доказательство эквивалентности всех таких задач в действительности являются весьма значительным результатом: это означает, что все эти открытые вопросы в действительности представляют собой один открытый вопрос, один тип сложности, который мы пока не понимаем в полной мере.

С практической точки зрения *NP*-полнота фактически означает «запредельную вычислительную сложность, хотя мы пока не можем этого доказать». Доказательство того, что задача является *NP*-полной, становится веской причиной для прекращения поисков эффективного алгоритма. С таким же успехом можно искать эффективный алгоритм для любой другой знаменитой вычислительной задачи, которая заведомо является *NP*-полной; многие ученые пытались найти эффективные алгоритмы для таких задач — но пока безуспешно.

8.1. Полиномиальное сведение

Мы намереваемся исследовать пространство вычислительно сложных задач, чтобы в конечном итоге прибыть к математической характеристике большого класса. Основным приемом в этом исследовании будет сравнение относительной сложности разных задач; нам хотелось бы найти формальное выражение для таких утверждений, как «задача X обладает как минимум не меньшей сложностью, чем задача Y». Для формализации будет применен метод *сведения*: чтобы показать, что некоторая задача X по меньшей мере не менее сложна, чем другая задача Y, мы докажем, что «черный ящик», способный решить задачу X, также сможет решить задачу Y. (Другими словами, решение X будет достаточно мощным для того, чтобы также решить Y.)

Чтобы точно выразить этот критерий, мы предположим, что задача X в нашей вычислительной модели может быть напрямую решена с полиномиальным временем. Допустим, имеется некий «черный ящик», способный решать экземпляры задачи X; если передать ему входные данные для экземпляра X, то «черный ящик» за один шаг вернет правильный ответ. А теперь зададимся следующим вопросом:

> (*) Возможно ли решить произвольный экземпляр задачи Y с полиномиальным количеством стандартных вычислительных шагов, дополняемых полиномиальным количеством обращений к «черному ящику», решающему задачу X?

Если ответ на этот вопрос положителен, то мы используем запись $Y \leq_P X$; это читается как «задача Y является полиномиальносводимой к X», или «сложность X по крайней мере не меньше сложности Y (в отношении полиномиального времени)». Обратите внимание: в этом определении учитываются затраты времени на передачу входных данных «черному ящику», решающему X, и на получение ответа от «черного ящика».

Такая формулировка сводимости чрезвычайно естественна. Когда мы задаемся вопросом о сводимости к задаче X, все выглядит так, словно вычислительная модель дополняется специализированным процессором, решающим экземпляры X за один шаг. А теперь разберемся, какую же вычислительную мощь предоставляет нам этот специализированный процессор?

У нашего определения $\leq_P$ имеется одно важное следствие: предположим, $Y \leq_P X$ и существует алгоритм с полиномиальным временем для решения X. Тогда польза от специализированного «черного ящика» для X не столь уж велика; его можно заменить алгоритмом с полиномиальным временем для X. Подумайте, что произойдет с нашим алгоритмом для задачи Y, использующим полиномиальное количество шагов плюс полиномиальное количество вызовов «черного ящика». Он превращается в алгоритм, использующий полиномиальное количество шагов плюс полиномиальное количество вызовов процедуры, выполняемой за полиномиальное время; другими словами, он превращается в алгоритм с полиномиальным временем. Таким образом, мы доказали следующий факт.

(8.1) Допустим, $Y \leq_P X$. Если задача X решается за полиномиальное время, то и задача Y может быть решена за полиномиальное время.

Собственно, этот факт, хотя и неявно, уже использовался ранее в книге. Вспомните, как мы решали задачу о двудольном паросочетании с использованием полиномиального объема предварительной обработки и времени решения одного экземпляра задачи о максимальном потоке. Так как задача о максимальном потоке может быть решена за полиномиальное время, мы заключили, что это возможно и для задачи о двудольном паросочетании. Кроме того, задача сегментации изображения решалась с использованием полиномиального объема предварительной обработки и времени решения одного экземпляра задачи о минимальном разрезе с теми же последствиями. Оба примера могут рассматриваться как прямые применения (8.1). В самом деле, (8.1) предоставляет отличный способ разработки алгоритмов с полиномиальным временем для новых задач: сведение к задаче, которую мы уже умеем решать за полиномиальное время.

Тем не менее в этой главе (8.1) будет использоваться для установления вычислительной неразрешимости некоторых задач. Мы будем заниматься довольно нетривиальным делом: речь пойдет об оценке разрешимости задач даже в том случае, когда мы не знаем, как решать их за полиномиальное время. Для этой цели будет использовано утверждение, противоположное (8.1); оно достаточно важно, чтобы представить его как отдельный факт.

(8.2) Допустим, $Y \leq_P X$. Если задача Y не решается за полиномиальное время, то и задача X не может быть решена за полиномиальное время.

Утверждение (8.2) очевидно эквивалентно (8.1), но оно подчеркивает наш общий план: если имеется заведомо сложная задача Y и мы показываем, что $Y \leq_P X$, то сложность «распространяется» на X; задача X должна быть сложной, или в противном случае она могла бы использоваться для решения Y.

Так как на практике мы не знаем, возможно ли решить изучаемые задачи за полиномиальное время или нет, обозначение $\leq_P$ будет использоваться для установления относительной сложности между задачами.

С учетом сказанного можно установить некоторые отношения сводимости между исходным набором фундаментально сложных задач.

Первое сведение: независимое множество и вершинное покрытие

Задача о независимом множестве, представленная в числе пяти типичных задач в главе 1, послужит первым прототипным примером сложной задачи. Алгоритм с полиномиальным временем для нее неизвестен, но мы также не знаем, как доказать, что такого алгоритма не существует.

Вернемся к формулировке задачи о независимом множестве, так как в нее будет добавлен один нюанс. Вспомните, что в графе $G = (V, E)$ множество узлов $S \subseteq V$ называется *независимым*, если никакие два узла в S не соединены ребром. Найти малое независимое множество в графе несложно (например, одиночный узел образует независимое множество); трудности возникают при поиске больших независимых множество так как требуется построить большой набор узлов, в котором нет соседних узлов. Например, множество узлов {3, 4, 5} является независимым множеством размера 3 в графе на рис. 8.1, тогда как множество узлов {1, 4, 5, 6} является независимым множеством большего размера.

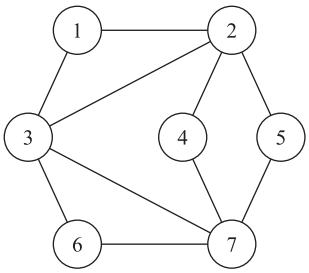


Рис. 8.1. Граф, для которого размер наибольшего независимого множества равен 4, а наименьшее вершинное покрытие имеет размер 3

В главе 1 была представлена задача нахождения наибольшего независимого множества в графе G. В контексте нашего текущего исследования, направленного на сводимость задач, намного удобнее работать с задачами, на которые можно ответить только «да/нет», поэтому задача о независимом множестве будет сформулирована следующим образом:

> Для заданного графа G и числа k содержит ли G независимое множество с размером не менее k?

С точки зрения разрешимости с полиномиальным временем оптимизационная версия задачи (найти максимальный размер независимого множества) не так уж сильно отличается от версии с проверкой условия (решить, существует ли в G независимое множество с размером не менее заданного k, — да или нет). Зная метод решения первой версии, мы автоматически решим вторую (для произвольного k).

Но также существует чуть менее очевидный обратный факт: если мы можем решить версию задачи о независимом множестве с проверкой условия для всех k, то мы также можем найти максимальное независимое множество. Для заданного графа G с n узлами версия с проверкой условия просто проверяется для каждого k; наибольшее значение k, для которого ответ будет положительным, является размером наибольшего независимого множества в G. (А при использовании бинарного поиска достаточно применить версию с проверкой условия для $O(\log n)$ разных значений k.) Простая эквивалентность между проверкой условия и оптимизацией также встречается в задачах, которые будут рассматриваться ниже.

Теперь для демонстрации нашей стратегии по установлению связи между сложными задачами будет рассмотрена другая фундаментальная задача из теории графов, для которой неизвестен эффективный алгоритм: задача о вершинном покрытии. Для заданного графа $G = (V, E)$ множество узлов $S \subseteq V$ образует *вершинное покрытие*, если у каждого ребра $e \in E$ по крайней мере один конец принадлежит S.

Обратите внимание на один нюанс: в задаче о вершинном покрытии в данном случае вершины «покрывают», а ребра являются «покрываемыми объектами». Найти большое вершинное покрытие для графа несложно (например, полное множество вершин); трудно найти покрытие меньшего размера. Задача о вершинном покрытии формулируется следующим образом.

Для заданного графа G и числа k содержит ли G вершинное покрытие с размером не более k?

Например, для графа на рис. 8.1 множество узлов {1, 2, 6, 7} является вершинным покрытием с размером 4, а множество {2, 3, 7} является вершинным покрытием с размером 3.

Мы не знаем, как решить задачу о независимом множестве или задачу о вершинном покрытии за полиномиальное время; но что можно сказать об их относительной сложности? Сейчас мы покажем, что эти задачи имеют эквивалентную сложность; для этого мы установим, что *Независимое множество* $\leq_P$ *Вершинное покрытие*, а также *Вершинное покрытие* $\leq_P$ *Независимое множество*. Это напрямую вытекает из следующего факта.

(8.3) Имеется граф $G = (V, E)$. S является независимым множеством в том и только в том случае, если его дополнение $V–S$ является вершинным покрытием.

Доказательство. Предположим, что S является независимым множеством. Рассмотрим произвольное ребро $e = (u, v)$. Из независимости S следует, что u и v не могут одновременно принадлежать S; следовательно, один из этих концов должен принадлежать $V–S$. Отсюда следует, что у каждого ребра как минимум один конец принадлежит $V–S$, а значит, $V–S$ является вершинным покрытием.

И наоборот, предположим, что $V–S$ является вершинным покрытием. Рассмотрим два любых узла u и v в S. Если бы они были соединены ребром e, то ни один из концов e не принадлежал бы $V–S$, что противоречило бы предположению о том, что $V–S$ является вершинным покрытием. Отсюда следует, что никакие два узла в S не соединены ребром, а значит, S является независимым множеством. ■

Сведения в обе стороны между двумя задачами следуют напрямую из (8.3).

(8.4) *Независимое множество* $\leq_P$ *Вершинное покрытие*.

Доказательство. Имея «черный ящик» для решения задачи о вершинном покрытии, мы могли бы решить, содержит ли G независимое множество с размером не менее k, обратившись к «черному ящику» с вопросом о том, содержит ли G вершинное покрытие с размером не более $n - k$. ■

(8.5) *Вершинное покрытие* $\leq_P$ *Независимое множество*.

Доказательство. Имея «черный ящик» для решения задачи о независимом множестве, мы могли бы решить, содержит ли G вершинное покрытие в размером не более k, обратившись к «черному ящику» с вопросом о том, содержит ли G независимое множество с размером не менее $n - k$. ■

Итак, приведенный анализ поясняет наш план в целом: хотя мы не знаем, как эффективно решать задачу о независимом множестве или задачу о вершинном покрытии, из (8.4) и (8.5) следует, что решение любой задачи предоставит эффективное решение другой, то есть эти два факта устанавливают относительные уровни сложности этих задач.

Применим эту стратегию для ряда других задач.

Сведение к более общему случаю: вершинное покрытие к покрытию множества

Задача о независимом множестве и задача о вершинном покрытии представляют собой две разные категории задач. Задача о независимом множестве может рассматриваться как «задача упаковки»: требуется «упаковать» как можно больше вершин с учетом конфликтов (ребер), препятствующих нам в этом. С другой стороны, задача о вершинном покрытии может рассматриваться как «задача покрытия»: требуется экономно «накрыть» все ребра в графе с использованием минимального количества вершин.

Задача о вершинном покрытии представляет собой задачу покрытия, сформулированную «на языке» графов; существует более общая задача покрытия — *задача покрытия множества*, в которой требуется покрыть произвольное множество объектов с использованием нескольких меньших множеств. Мы можем сформулировать задачу покрытия множества следующим образом:

> Задано множеств U из n элементов, набор S_1, ..., S_m подмножеств U и число k. Существует ли набор из не более чем k таких множеств, объединение которых равно всему множеству U?

Представьте, например, что имеются m программных компонентов и множество U из n функций, которыми должна обладать система. i-й компонент реализует множество $S_i \subseteq U$ функций. В задаче покрытия множества требуется включить небольшое количество компонентов, чтобы система поддерживала все n функций.

На рис. 8.2 изображен пример задачи покрытия множества: десять кругов представляют элементы основного множества U, а семь овалов и многоугольников — множества S_1, S_2, ..., S_7. В этом экземпляре задачи присутствует совокупность множеств, объединение которых равно U: например, можно выбрать высокий овал слева с двумя многоугольниками.

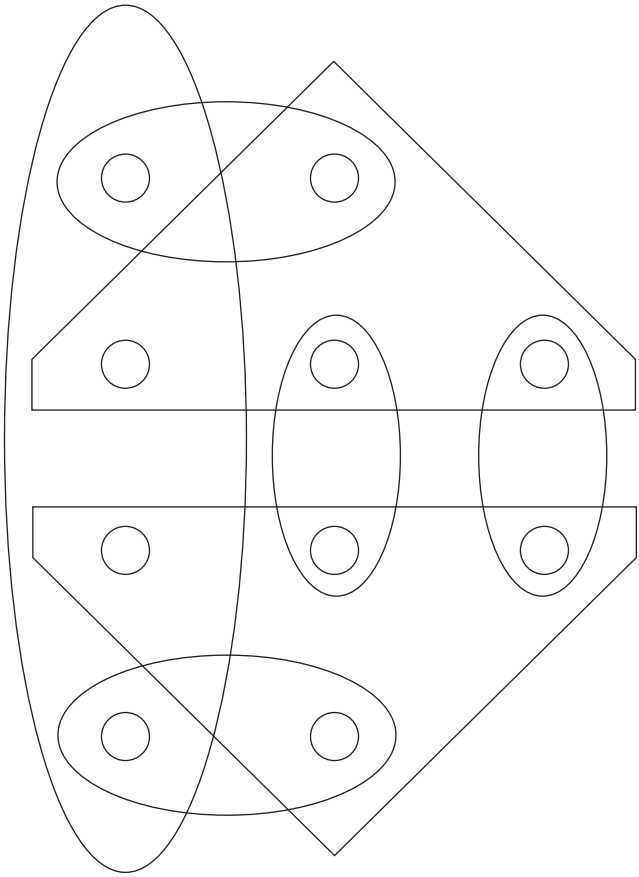


Рис. 8.2. Пример задачи покрытия множества

На интуитивном уровне понятно, что задача вершинного покрытия является особым случаем задачи покрытия множества: в последнем случае мы пытаемся покрыть произвольное множество произвольными подмножествами, тогда как в первом ставится частная задача покрытия ребер графа с использованием множеств ребер, инцидентных вершинам. Более того, можно доказать следующее сведение.

(8.6) *Вершинное покрытие* $\leq_p$ *Покрытие множества*.

Доказательство. Допустим, имеется «черный ящик», который умеет решать задачу о покрытии множества, и произвольный экземпляр задачи о вершинном покрытии, заданный графом $G = (V, E)$ и числом k. Как использовать «черный ящик» для решения этой задачи?

Наша цель — найти покрытие для ребер E, поэтому мы формулируем экземпляр задачи покрытия множества, в которой универсальное множество U равно E. Каждый раз, когда мы выбираем вершину в задаче вершинного покрытия, мы покрываем все ребра, инцидентные этой вершине; следовательно, для каждой

вершины $i \in V$ в экземпляр задачи покрытия множества добавляется множество $S_i \subseteq U$, состоящее из всех ребер G, инцидентных i.

Утверждается, что U может быть покрыто с использованием не более k из множеств S_1, ..., S_n в том, и только в том случае, если G имеет вершинное покрытие с размером не более k. Это утверждение доказывается очень просто. Если $\ell \le k$ множеств $S_{i_1},...,S_{i_\ell}$ покрывают U, то каждое ребро в G инцидентно одной из вершин $i_1,...,i_\ell$, а значит, множество $\{i_1,...,i_\ell\}$ является вершинным покрытием G с размером $\ell \le k$. И наоборот, если $\{i_1,...,i_\ell\}$ является вершинным покрытием G с размером $\ell \le k$, то множества $S_{i_1},...,S_{i_\ell}$ покрывают U.

Итак, для заданного экземпляра задачи о вершинном покрытии формулируется экземпляр задачи покрытия множества, описанный выше, который передается «черному ящику». Положительный ответ на исходный вопрос дается в том, и только в том случае, если «черный ящик» отвечает положительно.

(Вы можете убедиться в том, что экземпляр задачи покрытия множества на рис. 8.2 действительно соответствует полученному в ходе сведения, — начните с графа на рис. 8.1). ■

Обратите внимание на один важный момент, относящийся как к доказательству, так и к предшествующим сведениям в (8.4) и (8.5). Хотя определение $\le_P$ позволяет выдать много обращений к «черному ящику» для задачи покрытия множества, мы выдали только одно. В самом деле, наш алгоритм вершинного покрытия состоял просто из кодирования задачи в один экземпляр задачи покрытия множества и последующего использования полученного ответа как ответа на общую задачу. Это относится практически ко всем сведениям, которые мы будем рассматривать; в них отношение $Y \le_P X$ будет устанавливаться преобразованием экземпляра Y в один экземпляр X, передачей «черному ящику» этого экземпляра X и использованием полученного ответа для экземпляра Y.

Подобно тому как задача покрытия множества является естественным обобщением задачи о вершинном покрытии, существует естественное обобщение задачи о независимом множестве как задачи упаковки для произвольных множеств. А именно *задача упаковки множества* определяется следующим образом:

> Для заданного множества U из n элементов, набора S_1, ..., S_m подмножеств U и числа k существует ли набор из минимум k таких подмножеств, из которых никакие два не пересекаются?

Иначе говоря, мы хотим «упаковать» большое количество множеств с тем ограничением, что никакие два из них не пересекаются.

Рассмотрим пример ситуации, в которой может возникнуть такая задача. Допустим, имеется множество U ресурсов, используемых в монопольном режиме, и множество из m программных процессов. Для выполнения i-го процесса необходимо множество ресурсов $S_i \subseteq U$. Таким образом, задача упаковки множества ищет большую совокупность процессов, которые могут выполняться одновременно, без перекрытия требований к ресурсам (то есть без конфликтов).

Это естественная аналогия (8.6), а ее доказательство почти не отличается от приведенного; подробности остаются читателю для самостоятельной работы.

(8.7) *Независимое множество* $\le_P$ *Упаковка множества.*

8.2. Сведение с применением «регуляторов»: задача выполнимости

Сейчас мы займемся более абстрактными задачами, которые формулируются в булевой записи. Эти задачи применяются для моделирования широкого спектра задач, в которых требуется присваивать значения условных переменных для выполнения заданного набора ограничений: например, подобные формальные конструкции часто встречаются в области искусственного интеллекта. После знакомства с этими задачами мы свяжем их посредством сведения с задачами графов и множеств, рассматривавшимися ранее.

Задачи SAT и 3-SAT

Допустим, имеется множество X из *n булевых* переменных $x_1, ..., x_n$; каждая переменная может принимать значение 0 или 1 (эквиваленты `false` и `true`). *Литералом* по X называется одна из переменных x_i или ее отрицание $\overline{x_i}$. Наконец, *условием* называется обычная дизъюнкция литералов

$$t_1 \vee t_2 \vee \cdots \vee t_\ell .$$

(Еще раз: все $t_i \in \left\{x_1, x_2, ..., x_n, \overline{x_1}, ..., \overline{x_n}\right\}$). Мы говорим, что условие имеет длину ℓ, если оно содержит ℓ литералов.

А теперь дадим формальное определение присваиванию значений, удовлетворяющих набору условий. *Логическим присваиванием* для X называется присваивание значения 0 или 1 каждому x_i; другими словами, это функция $\nu : X \to \{0,1\}$. Присваивание ν неявно задает $\overline{x_i}$ значение истинности, противоположное x_i. Присваивание *выполняет* условие C, если после него C дает результат 1 по условиям булевой логики; это эквивалентно требованию о том, чтобы по крайней мере один из литералов в C имел значение 1. Присваивание выполняет совокупность условий $C_1, ..., C_k$, если в результате его все C_i дают результат 1; иначе говоря, если в результате его конъюнкция

$$C_1 \wedge C_2 \wedge ... \wedge C_k$$

дает результат 1. В этом случае ν называется *выполняющим присваиванием* в отношении $C_1, ..., C_k$, а набор условий $C_1, ..., C_k$ называется *выполнимым*.

Рассмотрим простой пример. Допустим, имеются три условия:

$$(x_1 \vee \overline{x_2}), (\overline{x_1} \vee \overline{x_3}), (x_2 \vee \overline{x_3}).$$

Логическое присваивание ν, которое задает всем трем переменным значение 1, не является выполняющим, потому что с ним не выполняется второе из перечисленных условий; с другой стороны, присваивание ν', которое задает всем переменным значение 0, является выполняющим.

Теперь можно привести формулировку *задачи выполнимости*, также обозначаемой *SAT*:

Для заданного множества условий C_1, ..., C_k по множеству переменных $X = \{x_1, ..., x_n\}$ существует ли выполняющее логическое присваивание?

Существует частный случай SAT, обладающий эквивалентной сложностью, но при этом более понятный; в нем все условия содержат ровно три литерала (соответствующих разным переменным). Назовем эту задачу *задачей* 3-*выполнимости*, или 3-*SAT*:

Для заданного множества условий C_1, ..., C_k, каждое из которых имеет длину 3, по множеству переменных $X = \{x_1, ..., x_n\}$, существует ли выполняющее логическое присваивание?

Задачи выполнимости и 3-выполнимости являются фундаментальными задачами комбинаторного поиска; они содержат основные составляющие сложной вычислительной задаче и в предельно упрощенном виде. Требуется принять n независимых решений (присваивания всем x_i) так, чтобы выполнить набор ограничений. Существуют разные способы выполнения каждого ограничения по отдельности, но решения придется скомбинировать так, чтобы все ограничения выполнялись одновременно.

Сведение задачи 3-SAT к задаче о независимом множестве

А теперь свяжем вычислительную сложность, воплощенную в задачах SAT и 3-SAT, с другой (на первый взгляд) сложностью, представленной поиском независимых множеств и вершинных покрытий в графах, а именно: мы покажем, что 3-*SAT* $\leq_P$ *Независимое множество*. Основная трудность в подобных доказательствах очевидна: в задаче 3-SAT речь идет о присваивании значений булевым переменным с учетом ограничений, тогда как задача о независимом множестве направлена на выбор вершин в графе. Чтобы решить экземпляр задачи 3-SAT с использованием «черного ящика» для задачи о независимом множестве, необходимо как-то закодировать все эти булевы ограничения в узлах и ребрах графа, чтобы критерий выполнимости соответствовал существованию большого независимого множества.

Этот прием демонстрирует общий принцип проектирования сложных сведений $Y \leq_P X$: построение «регуляторов» из компонентов в задаче X для представления того, что происходит в задаче Y.

(8.8) 3-*SAT* $\leq_P$ *Независимое множество*.

Доказательство. Имеется «черный ящик» для задачи о независимом множестве; мы хотим решить экземпляр задачи 3-SAT, состоящий из переменных $X = \{x_1, ..., x_n\}$ и условий C_1, ..., C_k.

Чтобы правильно взглянуть на проблему сведения, следует понять, что существуют две концептуально различающиеся точки зрения на экземпляр 3-SAT.

- Первый способ представления экземпляра 3-SAT был предложен ранее: для каждой из n переменных принимается независимое решение 0/1, а успех достигается при достижении одного из трех способов выполнения каждого условия.
- Тот же экземпляр 3-SAT можно представить иначе: нужно выбрать один литерал из каждого условия, а затем найти логическое присваивание, в результате которого все эти литералы дают результат 1 с выполнением всех условий. Итак, успех достигается в том случае, если вы сможете выбрать литерал из каждого условия так, чтобы никакие два выбранных литерала не «конфликтовали»; мы говорим, что *конфликт* двух литералов возникает тогда, когда один равен переменной x_i, а другой ее отрицанию $\overline{x_i}$. Если удастся избежать конфликтов, мы сможем найти логическое присваивание, в результате которого выбранные литералы из каждого условия дают результат 1.

Следующее сведение базируется на втором представлении экземпляра 3-SAT; мы закодируем его с использованием независимых множеств в графе. Сначала построим граф $G = (V, E)$, состоящий из $3k$ узлов, сгруппированных в k треугольников, как показано на рис. 8.3. Таким образом, для $i = 1, 2, ..., k$ строятся три вершины v_{i1}, v_{i2}, v_{i3}, соединенные друг с другом ребрами. Каждой вершине присваивается метка; v_{ij} помечается j-м литералом из условия C_i экземпляра 3-SAT.

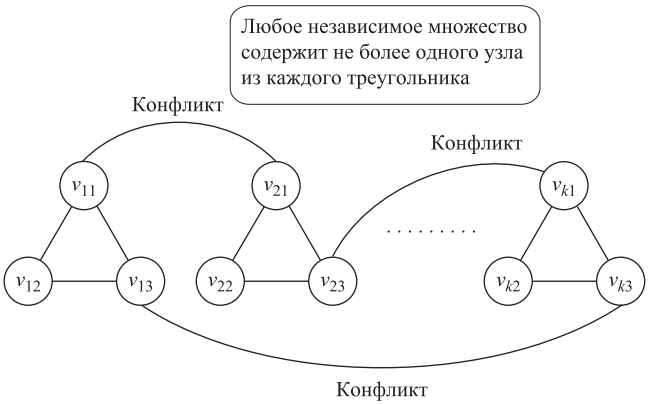


Рис. 8.3. Сведение задачи 3-SAT к задаче о независимом множестве

Прежде чем продолжать, рассмотрим, как выглядят независимые множества с размером k на этом графе: так как две вершины не могут быть выбраны из одного треугольника, они состоят из всех способов выбора одной вершины из каждого треугольника. Так реализуется наша цель по выбору литерала в каждом условии, который дает результат 1; но пока мы не сделали ничего, чтобы предотвратить конфликт между двумя литералами.

Конфликты будут кодироваться добавлением новых ребер в граф: каждую пару вершин, метки которых соответствуют конфликтующим литералам, мы соединяем ребром. Приведет ли это к уничтожению всех независимых множеств

размера k или такое множество существует? Это зависит от того, можно ли выбрать один узел из каждого треугольника, чтобы при этом не были выбраны конфликтующие пары узлов. Но ведь именно то, что требуется в экземпляре 3-SAT!

Утверждается, что исходный экземпляр задачи 3-SAT выполним в том и только в том случае, если построенный граф G имеет независимое множество с размером не менее k. Во-первых, если экземпляр 3-SAT выполним, то каждый треугольник в графе содержит как минимум один узел, метка которого дает результат 1. Пусть S — множество, состоящее из одного такого узла в каждом треугольнике. Множество S является независимым; если бы между двумя узлами $u, v \in S$ существовало ребро, то метки u и v должны были бы конфликтовать; но это невозможно, так как обе они дают результат 1.

И наоборот, предположим, что граф G содержит независимое множество S с размером не менее k. Тогда прежде всего размер S равен точно k, и оно должно содержать один узел из каждого треугольника. Далее утверждается, что существует логическое присваивание ν для переменных в экземпляре 3-SAT, обладающее тем свойством, что метки всех узлов S дают результат 1. Как же построить такое присваивание ν? Для каждой переменной x_i, если ни x_i, ни $\overline{x_i}$ не используется как метка узла в S, мы присваиваем $\nu(x_i) = 1$. В противном случае либо x_i, либо $\overline{x_i}$ является меткой узла в S; потому что если бы один узел в S был помечен x_i, а другой $\overline{x_i}$, то эти два узла были бы соединены ребром, что противоречило бы нашему предположению о том, что S является независимым множеством. Если x_i является меткой узла в S, мы присваиваем $\nu(x_i) = 1$; в противном случае выполняется присваивание $\nu(x_i) = 0$. При таком построении ν все метки узлов в S дают результат 1.

Так как G содержит независимое множество с размером не менее k в том, и только в том случае, если исходный экземпляр 3-SAT был выполнимым, сведение завершено. ■

Транзитивность сведения

Мы рассмотрели несколько разных сложных задач, относящихся к разным классам, и выяснили, что они тесно связаны друг с другом. Мы можем вычислить несколько дополнительных отношений, используя факт транзитивности $\leq_P$.

(8.9) Если $Z \leq_P Y$ и $Y \leq_P X$, то $Z \leq_P X$.

Доказательство. Располагая «черным ящиком» для решения X, мы покажем, как решить экземпляр задачи Z; по сути, мы просто выполняем композицию двух алгоритмов, подразумеваемых $Z \leq_P Y$ и $Y \leq_P X$. Алгоритм для Z выполняется с использованием «черного ящика» для Y; но каждое обращение к «черному ящику» для Y *моделируется* полиномиальным количеством шагов, использующих алгоритм, решающий экземпляры Y с использованием «черного ящика» для X. ■

Свойство транзитивности весьма полезно. Например, так как мы доказали, что 3-*SAT* $\leq_P$ *Независимое множество* $\leq_P$ *Вершинное покрытие* $\leq_P$ *Покрытие множества*, можно сделать вывод, что 3-*SAT* $\leq_P$ *Покрытие множества*.

8.3. Эффективная сертификация и определение *NP*

Сведение задач было первым главным компонентом в нашем изучении вычислительной неразрешимости. Вторым компонентом является получение характеристик класса задач, с которыми мы имеем дело. Объединение этих двух компонентов с мощной теоремой Кука и Левина приводит к некоторым неожиданным последствиям.

Вспомните, что в главе 1, при первом знакомстве с задачей о независимом множестве, мы спрашивали: можно ли сказать хоть что-нибудь положительное о сложности задачи (с вычислительной точки зрения)? И действительно, кое-что положительное нашлось: если граф содержит независимое множество с размером не менее k, то этот факт можно легко доказать, представив такое независимое множество. Аналогичным образом, если экземпляр 3-SAT является выполнимым, это можно доказать, представив выполняющее присваивание. Поиск такого присваивания может быть невероятно сложной задачей, но после того, как сложная работа по его поиску будет завершена, остается подставить значения в условия и убедиться в том, что все они выполняются.

Здесь важен контраст между *поиском* решения и *проверкой* предложенного решения. Для задачи о независимом множестве или 3-SAT алгоритм поиска решений с полиномиальным временем нам неизвестен; однако проверка предлагаемого решения таких задач легко выполняется за полиномиальное время. Чтобы понять, что эта проблема не так уж тривиальна, представьте, какая проблема возникла бы, если бы потребовалось доказать невыполнимость экземпляра задачи 3-SAT. Какие «доказательства» можно было бы предъявить, чтобы убедить вас за полиномиальное время в невыполнимости задачи?

Задачи и алгоритмы

В этом заключена суть характеристики; теперь мы перейдем к ее формализации. Входные данные вычислительной задачи могут быть закодированы в виде конечной двоичной строки s. Длина строки s обозначается $|s|$. Задача принятия решения X будет описываться множеством строк, для которых возвращается ответ «да». Алгоритм A для задачи принятия решения получает входную строку s и возвращает «да» или «нет» — это возвращаемое значение будет обозначаться $A(s)$. Алгоритм A решает задачу X, если для всех строк s условие $A(s) = да$ выполняется в том, и только в том случае, если $s \in X$.

Как обычно, мы говорим, что A имеет полиномиальное время выполнения, если существует такая полиномиальная функция $p(\cdot)$, что для каждой входной строки s алгоритм A завершается для s не более чем за $O(p(|s|))$ шагов. До настоящего момента мы занимались задачами, которые решались за полиномиальное время. В приведенной выше записи этот факт можно выразить как множество P всех задач X, для которых существует алгоритм A с полиномиальным временем выполнения, решающий X.

Эффективная сертификация

Как же формализовать идею о том, что решение задачи может быть *проверено* эффективно независимо от того, насколько эффективно может решаться сама задача? Структура «алгоритма проверки» для задачи X отличается от структуры алгоритма, ищущего решение; для «проверки» решения необходима входная строка s и отдельная строка t («сертификат»), доказывающая, что s является «положительным» экземпляром X.

Итак, алгоритм B называется эффективным *сертифицирующим алгоритмом* для задачи X, если он обладает следующими свойствами:

- B — алгоритм с полиномиальным временем, получающий два входных аргумента s и t;
- существует такая полиномиальная функция p, что для каждой строки s условие $s \in X$ выполняется в том, и только в том случае, если существует строка t, для которой $|t| \leq P(|s|)$ и $B(s, t) = да$.

Чтобы понять, о чем в действительности говорит это определение, потребуется некоторое время. Эффективный сертифицирующий алгоритм следует рассматривать как подход к задаче X с «управленческой» точки зрения. Он не пытается самостоятельно решить, принадлежит ли входная строка s множеству X. Вместо этого он эффективно оценивает предлагаемые «доказательства» t, что s принадлежит X (при условии, что они не слишком длинные), и он является правильным алгоритмом в том слабом смысле, что s принадлежит X тогда, и только тогда, когда существует доказательство, которое его в этом убедит.

Эффективный сертифицирующий алгоритм B может использоваться как центральный компонент алгоритма «грубой силы» для задачи X: для входной строки s проверить все строки t длины $\leq P(|s|)$ и проверить, выполняется ли условие $B(s, t) = да$ для каких-либо из этих строк. Однако существование B не дает никакого очевидного способа построения эффективного алгоритма, который решает X; в конце концов, мы еще должны найти строку t, для которой $B(s, t)$ вернет ответ «да», а количество возможностей для t экспоненциально велико.

NP: класс задач

Мы определим NP как множество задач, для которых существует эффективный сертифицирующий алгоритм[1]. Сразу можно заметить одно из его свойств.

(8.10) $P \subseteq NP$.

Доказательство. Рассмотрим задачу $X \in P$; это означает, что существует алгоритм с полиномиальным временем, который решает X. Чтобы показать, что $X \in NP$, необходимо показать, что существует эффективный сертифицирующий алгоритм B для X.

[1] Операция поиска строки t, с которой эффективный сертифицирующий алгоритм примет ввод s, часто рассматривается как *недетерминированный поиск* по пространству возможных доказательств t; по этой причине термин NP был выбран как сокращение для «Nondeterministic Polynomial time» («недетерминированное полиномиальное время»).

Сделать это очень просто; B строится по следующей схеме. При получении входной пары (s, t) сертифицирующий алгоритм B просто возвращает значение $A(s)$. (Считайте B своего рода «прагматиком», который игнорирует предложенное доказательство t и просто решает задачу своими силами.) Почему B является эффективным сертификатором для X? Очевидно, он выполняется с полиномиальным временем, как и A. Если строка $s \in X$, то для всех t длины не более $p(|s|)$ имеем $B(s, t) = да$. С другой стороны, если $s \notin X$, то для всех t длины не более $p(|s|)$ имеем $B(s, t) = нет$. ■

Нетрудно убедиться в том, что задачи, представленные в первых двух разделах, относятся к категории *NP*: достаточно определить, как эффективный сертифицирующий алгоритм для каждой из них будет использовать «сертификат», то есть строку t. Например:

- для задачи 3-SAT сертификат t является результатом присваивания булевых значений переменным; сертифицирующий алгоритм B проверяет заданное множество условий в соответствии с этим присваиванием;
- для задачи о независимом множестве сертификат t является описанием множества, содержащим не менее k вершин; сертифицирующий алгоритм B проверяет, что среди этих вершин нет двух, соединенных ребром;
- для задачи покрытия множества сертификат t представляет собой список k множеств из заданного набора; сертифицирующий алгоритм проверяет, что объединение этих множеств равно базовому множеству U.

Тем не менее мы не можем доказать, что решение каких-либо из этих задач требует более чем полиномиального времени. Более того, мы не можем доказать, что в *NP* входит хотя бы одна задача, не принадлежащая *P*. Итак, вместо конкретной теоремы мы можем только задать вопрос:

(8.11) Присутствует ли в *NP* задача, не принадлежащая *P*? Возможно, $P = NP$?

Вопрос о том, истинно равенство $P = NP$ или нет, является фундаментальной проблемой в теории алгоритмов и одной из самых известных задач в современной теории вычислений. Общественное мнение склоняется к тому, что $P \neq NP$, и это считается рабочей гипотезой в этой области, однако никаких убедительных технических свидетельств для этого нет. Скорее считается, что результат $P = NP$ выглядит слишком невероятно, чтобы быть правдой. Неужели может существовать общее преобразование задачи проверки решения в куда более сложную задачу фактического поиска решения? Неужели могут существовать общие средства проектирования эффективных алгоритмов, достаточно мощных для решения всех этих сложных задач, которые почему-то не были обнаружены? На неудачные попытки разработки алгоритмов с полиномиальным временем для сложных задач в *NP* были потрачены колоссальные усилия; пожалуй, самое естественное объяснение всех этих повторяющихся неудач состоит в том, что эти задачи просто не могут быть решены за полиномиальное время.

8.4. *NP*-полные задачи

Не достигув прогресса в разрешении вопроса $P = NP$, ученые обратились к сопутствующему, но более доступному вопросу: какие задачи в *NP* обладают наибольшей сложностью? Сведение к полиномиальному времени позволяет нам рассмотреть этот вопрос и получить представление о структуре *NP*.

Возможно, самый естественный способ определения «наиболее сложной» задачи X основан на следующих двух свойствах: (i) $X \in NP$ и (ii) для всех $Y \in NP$ $Y \leq_P X$. Другими словами, мы требуем, чтобы каждая задача в *NP* сводилась к X. Такая задача X будет называться *NP-полной задачей*.

Следующий факт подкрепляет нашу трактовку термина «наиболее сложная».

(8.12) Допустим, X является *NP*-полной задачей. Тогда X решается за полиномиальное время в том, и только в том случае, если $P = NP$.

Доказательство. Очевидно, если $P = NP$, то задача X решается за полиномиальное время, так как она принадлежит *NP*. И наоборот, предположим, что X решается за полиномиальное время. Если Y — любая другая задача в *NP*, то $Y \leq_P X$, а следовательно, из (8.1) задача Y может быть решена за полиномиальное время. Отсюда $NP \subseteq P$; в сочетании с (8.10) приходим к искомому заключению. ■

Из (8.12) вытекает одно важное следствие: если в *NP* существует *любая* задача, которая не решается за полиномиальное время, то никакая *NP*-полная задача не может быть решена за полиномиальное время.

Выполнимость булевой схемы: первая *NP*-полная задача

Наше определение *NP*-полноты имеет очень полезные свойства. Но прежде чем погружаться в увлекательные размышления по этому поводу, стоит остановиться и обратить внимание на один факт: сам факт *существования NP*-полных задач не так уж очевиден. Почему не могут существовать две несовместимые задачи X' и X'', для которых не существует $X \in NP$ с тем свойством, что $X' \leq_P X$ и $X'' \leq_P X$? Почему в *NP* не может существовать бесконечная последовательность задач $X_1, X_2, X_3, ...$, каждая из которых строго сложнее предыдущей? Чтобы доказать *NP*-полноту задачи, необходимо показать, как она может использоваться для кодирования любой задачи из *NP*. Эта проблема намного сложнее тех, с которыми мы сталкивались в разделах 8.1 и 8.2, когда искали способы кодирования конкретных задач в контексте других.

В 1971 году Кук и Левин независимо показали, как это делается для очень естественных задач в *NP*. Возможно, самым естественным выбором для первой *NP*-полной задачи была *задача выполнимости булевой схемы* (circuit satisfiability), описанная ниже.

Чтобы дать определение задачи, необходимо четко сформулировать, что имеется в виду под «булевой схемой». Рассмотрим стандартные булевы операторы, которые используются при определении задачи выполнимости: $\wedge$ (И), $\vee$ (ИЛИ)

и ¬ (НЕ). Наше определение строится по образцу физической схемы, собранной из логических элементов, реализующих эти операторы. Затем булева схема K определяется как помеченный, направленный ациклический граф наподобие изображенного на рис. 8.4.

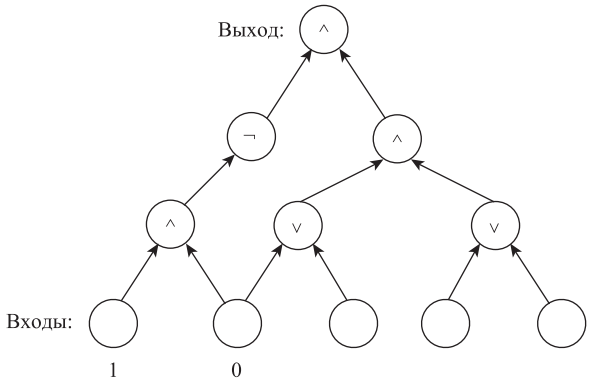


Рис. 8.4. Схема с тремя входами, двумя дополнительным источниками, которым присвоены логические значения, и одним выходом

- *Источники* в K (узлы, не имеющие входящих ребер) помечаются одной из констант — 0 или 1 либо именем конкретной переменной. Узлы последнего типа в дальнейшем называются *входами* схемы.
- Все остальные узлы помечаются одним из булевых операторов $\wedge$, $\vee$ или ¬; узлы с меткой $\wedge$ или $\vee$ имеют два входящих ребра, а узлы с меткой ¬ — одно входящее ребро.
- Также существует один узел, который не имеет выходящих ребер и представляет *выход* — результат, вычисляемый схемой.

Схема вычисляет функцию своих входов следующим естественным способом. Ребра представляются как «провода», передающие сигнал 0/1 от узла, из которого они выходят. Любой узел v, кроме источников, получает значения из входящих ребер (или ребра) и применяет к нему оператор, которым он помечен. Результат операции $\wedge$, $\vee$ или ¬ передается по ребрам (или ребру), выходящим из v. Итоговое значение, производимое схемой, вычисляется в выходном узле.

Для примера рассмотрим схему на рис. 8.4. Двум левым источникам присвоены значения 1 и 0, а следующие три источника соответствуют входам. Если подать на входы значения 1, 0, 1 (слева направо), то в логических элементах второй строки будут получены значения 0, 1, 1, в логических элементах третьей строки — значения 1, 1, и на выходе — значение 1.

Задача выполнимости булевой схемы формулируется следующим образом. Для заданной схемы нужно решить, существует ли распределение значений по входам, для которого на выходе будет получено значение 1. (Если оно существует, схема называется *выполнимой*, а *выполнимым распределением* называется распределение, приводящее к значению 1 на выходе). Как только что было показано, наш пример

на рис. 8.4 является выполнимым (со значениями 1, 0, 1 на входах). Теорему Кука и Левина можно рассматривать как эквивалентную следующему утверждению.

(8.13) Задача выполнимости булевой схемы является *NP*-полной.

Как упоминалось выше, чтобы доказать (8.13), следует рассмотреть произвольную задачу X в NP и показать, что $X \leq_P$ *Выполнимость булевой схемы*. Мы не будем описывать доказательство (8.13) во всех подробностях, но понять основную идею, заложенную в его основу, не так уж трудно. Мы используем тот факт, что любой алгоритм, который получает фиксированное количество n битов на входе и выдает ответ «да/нет», может быть представлен схемой только что определенного типа: такая схема эквивалентна алгоритму в том смысле, что ее результат равен 1 точно для тех входных значений, для которых алгоритм выдает ответ «да». Более того, если алгоритм выполняется за серию шагов, полиномиальных по n, то схема имеет полиномиальный размер. Этот переход от алгоритма к схеме является частью доказательства (8.13), в которое мы углубляться не будем, хотя оно вполне естественно с учетом того факта, что реализация алгоритмов на физических компьютерах может быть сведена к операциям с используемым множеством логических элементов $\wedge$, $\vee$ и $\neg$. (Обратите внимание на важность изменения количества входных битов, в котором отражается основное различие между алгоритмами и схемами: алгоритм обычно без труда справляется с разными вариантами ввода с разной длиной, а в структуре схемы размер входных данных жестко фиксируется.)

Как использовать это отношение между алгоритмами и схемами? Мы пытаемся показать, что $X \leq_P$ *Выполнимость булевой схемы*, — то есть для заданных входных данных s решить, выполняется ли $s \in X$, с использованием «черного ящика» для решения экземпляров задачи выполнимости булевой схемы. Сейчас мы знаем о задаче X лишь то, что у нее имеется эффективный сертифицирующий алгоритм $B(\cdot,\cdot)$. Итак, чтобы проверить $s \in X$ для конкретных входных данных s длины n, необходимо ответить на следующий вопрос: существует ли t длины $p(n)$, для которого $B(s, t) =$ *да*?

Мы ответим на этот вопрос при помощи «черного ящика» для задачи выполнимости булевой схемы. Так как нас интересует только ответ для конкретных входных данных s, $B(\cdot,\cdot)$ будет рассматриваться как алгоритм для $n + p(n)$ битов (входные данные s и сертификат t), который мы преобразуем в схему K полиномиального размера с $n + p(n)$ источниками. Первые n источников жестко кодируются значениями битов s, а остальные $p(n)$ источников помечаются переменными, представляющими биты t; последние источники станут входами для K.

Остается заметить, что $s \in X$ в том, и только в том случае, если существует такой способ назначить входные биты K, чтобы схема выдавала на выходе 1, — иначе говоря, в том, и только в том случае, если схема K выполнима. Тем самым устанавливается $X \leq_P$ *Выполнимость булевой схемы* и завершается доказательство (8.13). ■

Пример

Чтобы лучше понять суть того, что происходит в доказательстве (8.13), мы рассмотрим простой и конкретный пример. Допустим, имеется следующая задача.

Содержит ли заданный граф G независимое множество из двух узлов?

Обратите внимание: эта задача принадлежит к категории *NP*. Посмотрим, как решить экземпляр этой задачи построением эквивалентного экземпляра задачи выполнимости булевой схемы.

Следуя структуре приведенного выше доказательства, мы сначала рассмотрим эффективный сертифицирующий алгоритм для этой задачи. Входные данные s представляют граф из n узлов, который будет задаваться $\binom{n}{2}$ битами: для каждой пары узлов будет присутствовать бит, указывающий, соединены ли эти два узла ребром. Сертификат t может задаваться n битами: для каждого узла включается бит, указывающий, принадлежит ли этот узел предлагаемому независимому множеству. Эффективный сертифицирующий алгоритм должен проверить два факта: что по крайней мере два бита в t равны 1 и что никакие два бита в t не равны 1, если они представляют два конца ребра (что определяется соответствующим битом в s).

Теперь для конкретной длины n, соответствующей интересующим нас входным данным s, строится эквивалентная схема K. Предположим, к примеру, что мы хотим получить ответ на задачу для графа G с тремя узлами u, v, w, в котором узел v соединен с u и w. Это означает, что мы имеем дело с вводом длины $n = 3$. На рис. 8.5 изображена схема, эквивалентная эффективному сертифицирующему алгоритму для нашей задачи с произвольным графом из трех узлов. (Фактически правая часть схемы проверяет, что были выбраны как минимум два узла, а левая — что не были выбраны оба конца любого ребра). Ребра G кодируются константами в первых трех источниках, а остальные три источника (представляющие узлы, включаемые в независимое множество) остаются переменными. Заметим, что выполнимость этого экземпляра задачи выполнимости схемы проверяется передачей 1, 0, 1 на вход. Это соответствует выбору узлов u и w, которые действительно образуют независимое множество из двух узлов в трехузловом графе G.

Доказательство *NP*-полноты других задач

Утверждение (8.13) открывает путь к более полному пониманию сложных задач в *NP*: получив первую *NP*-полную задачу, мы можем обнаружить много больше таких задач при помощи простого наблюдения.

(8.14) Если Y — *NP*-полная задача, а X — задача в *NP*, обладающая свойством $Y \leq_P X$, то задача X является *NP*-полной.

Доказательство. Так как $X \in NP$, необходимо проверить только свойство (ii) определения. Пусть Z — любая задача в *NP*. Имеем $Z \leq_P Y$ вследствие *NP*-полноты Y и $Y \leq_P X$ по определению. Из (8.9) следует, что $Z \leq_P X$. ■

Итак, хотя доказательство (8.13) потребовало тяжелой работы по рассмотрению всех возможных задач в *NP*, доказательство *NP*-полноты других задач потребовало только сведения от одной, заведомо *NP*-полной задачи, — благодаря (8.14).

Ранее мы рассмотрели примеры сведения для некоторых базовых вычислительно сложных задач. Чтобы установить их *NP*-полноту, необходимо связать задачу

выполнимости булевой схемы с этим набором задач. Проще всего это было сделать для задачи, с которой у нее больше всего общего: задачей 3-SAT.

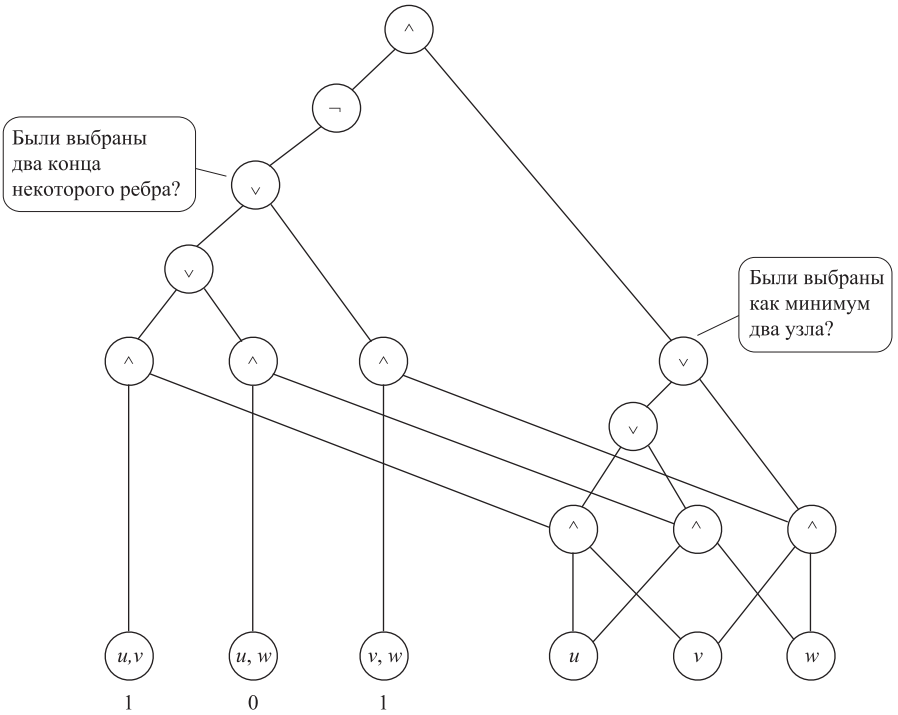


Рис. 8.5. Схема для проверки наличия 2-узлового независимого множества в 3-узловом графе

(8.15) Задача 3-SAT является *NP*-полной.

Доказательство. Очевидно, задача 3-SAT принадлежит *NP*, так как мы можем проверить за полиномиальное время, что предложенный вариант присваивания удовлетворяет заданному набору условий. Для доказательства ее *NP*-полноты будет использоваться сведение $SAT \leq_P 3\text{-}SAT$.

Для заданного экземпляра задачи выполнимости булевой схемы мы сначала построим эквивалентный экземпляр SAT, в котором каждое условие содержит не более трех переменных. Затем этот экземпляр SAT будет преобразован в эквивалентный, в котором каждое условие содержит *ровно* три переменные. Таким образом, последний набор условий будет экземпляром 3-SAT, а следовательно, завершает сведение.

Итак, рассмотрим произвольную булеву схему K. Переменная x_v связывается с каждым узлом v схемы для кодирования логического значения, содержащегося в этом узле схемы. Далее мы определим условия задачи SAT. Сначала нужно закодировать требование о том, что схема правильно вычисляет значение в каждом логическом элементе по входным значениям. Возможны три случая в зависимости от трех типов элементов.

- Если узел *v* помечен операцией $\neg$ и его единственное входящее ребро выходят из узла *u*, то должно выполняться $x_v = \overline{x_u}$. Чтобы гарантировать это, мы добавляем два условия: $(x_v \vee x_u)$ и $(\overline{x_v} \vee \overline{x_u})$.
- Если узел *v* помечен операцией $\vee$ и два его входящих ребра выходят из узлов *u* и *w*, то должно выполняться $x_v = x_u \vee x_w$. Чтобы гарантировать это, мы добавляем следующие условия: $(x_v \vee \overline{x_u})$, $(x_v \vee \overline{x_w})$ и $(\overline{x_v} \vee x_u \vee x_w)$.
- Если узел *v* помечен операцией $\wedge$ и два его входящих ребра выходят из узлов *u* и *w*, то должно выполняться $x_v = x_u \wedge x_w$. Чтобы гарантировать это, мы добавляем следующие условия: $(\overline{x_v} \vee x_u)$, $(\overline{x_v} \vee x_w)$ и $(x_v \vee \overline{x_u} \vee \overline{x_w})$.

Наконец, необходимо гарантировать, что константы в источниках имеют указанные значения, а на выходе выдается результат 1. Соответственно для источника *v*, помеченного константным значением, добавляется условие с одной переменной x_v или $\overline{x_v}$, под воздействием которого x_v принимает заданное значение. Для выходного узла *o* добавляется условие с одной переменной x_o, которое требует, чтобы значение *o* было равно 1. На этом построение завершается.

Не так сложно показать, что только что построенный экземпляр SAT эквивалентен заданному экземпляру задачи выполнимости булевой схемы. Чтобы продемонстрировать эквивалентность, необходимо привести два обоснования. Во-первых, предположим, что заданная схема *K* выполнима. Выполняющее присваивание входам схемы расширяется для создания значений во всех узлах *K* (как было сделано в примере на рис. 8.4). Это множество значений очевидным образом обеспечивает выполнимость построенного экземпляра SAT.

Чтобы провести рассуждения в другом направлении, мы предположим, что построенный экземпляр SAT является выполнимым. Рассмотрим выполняющее присваивание в этом экземпляре и значения переменных, соответствующие входам схемы *K*. Утверждается, что эти значения образуют выполняющее присваивание для схемы *K*. Чтобы убедиться в этом, достаточно заметить, что условия SAT гарантируют, что значения, присвоенные всем узлам *K*, совпадают с теми, которые вычисляются схемой для этих узлов. В частности, выходу будет присвоено значение 1, поэтому присваивание значений входам обеспечивает выполнимость *K*.

Итак, мы показали, как создать экземпляр SAT, эквивалентный задаче выполнимости булевой схемы. Но работа еще не завершена, так как нашей целью является создание экземпляра 3-SAT, который требует, чтобы длина всех условий была равна в точности 3, — а в созданном нами экземпляре некоторые условия имеют длину 1 или 2. Итак, для завершения доказательства необходимо преобразовать этот экземпляр SAT в эквивалентный экземпляр, в котором каждое условие состоит ровно из трех переменных.

Для этого мы введем четыре новые переменные: z_1, z_2, z_3, z_4. Идея заключается в том, чтобы гарантировать, что для каждого выполняющего присваивания $z_1 = z_2 = 0$; для этого добавляются условия $(\overline{z_i} \vee z_3 \vee z_4)$, $(\overline{z_i} \vee \overline{z_3} \vee z_4)$, $(\overline{z_i} \vee z_3 \vee \overline{z_4})$

и $\left(\overline{z_i} \vee \overline{z_3} \vee \overline{z_4}\right)$ для $i = 1$ и $i = 2$. Обратите внимание: выполнение всех этих условий возможно только в том случае, если $z_1 = z_2 = 0$.

Теперь рассмотрим условие в построенном экземпляре SAT, которое содержит один литерал t (которым может быть переменная или отрицание переменной). Каждый такой литерал заменяется условием $(t \vee z_1 \vee z_2)$. Аналогичным образом каждое условие, содержащее два литерала (допустим, $t \vee t'$), заменяется условием $(t \vee t' \vee z_1)$. Полученная формула 3-SAT очевидно эквивалентна формуле SAT, содержащей не более трех переменных в каждом условии, что завершает доказательство. ■

Используя этот результат и последовательность сведений

$$3\text{-}SAT \leq_P \textit{Независимое множество} \leq_P \textit{Вершинное покрытие} \leq_P \textit{Покрытие множества},$$

мы через (8.14) приходим к следующему выводу:

(8.16) Все следующие задачи являются *NP*-полными: задача о независимом множестве, задача упаковки множества, задача о вершинном покрытии и задача покрытия множества.

Доказательство. Каждая из этих задач обладает тем свойством, что она принадлежит *NP*, и задача 3-SAT (а следовательно, и задача SAT) может быть сведена к ней. ■

Общая стратегия доказательства *NP*-полноты новых задач

В оставшейся части этой главы мы в основном будем заниматься изучением других *NP*-полных задач. В частности, мы обсудим другие разновидности сложных вычислительных задач и докажем, что некоторые представители этих категорий являются *NP*-полными. Как упоминалось в начале главы, для этого есть чисто практическая причина: так как широко распространено мнение о том, что $P \neq NP$, идентификация *NP*-полноты задачи может стать веским признаком того, что задача не может быть решена за полиномиальное время.

Основная стратегия доказательства *NP*-полноты новой задачи X выглядит примерно так:

1. Доказать, что $X \in NP$.
2. Выбрать задачу Y, которая заведомо является *NP*-полной.
3. Доказать, что $Y \leq_P X$.

Ранее было замечено, что многие сведения $Y \leq_P X$ состоят из преобразования заданного экземпляра Y в экземпляр X с тем же ответом. Для решения X используется одно обращение к «черному ящику». При использовании подобных сведений приведенная выше стратегия превращается в следующий план доказательства *NP*-полноты.

1. Доказать, что $X \in NP$.
2. Выбрать задачу Y, которая заведомо является NP-полной.
3. Рассмотреть произвольный экземпляр s_Y задачи Y и показать, как построить за полиномиальное время экземпляр s_X задачи X, обладающий следующими свойствами.

   (a) Если s_Y является экземпляром Y, на который дается ответ «да», то и s_X является экземпляром X, на который дается ответ «да».

   (b) Если s_X является экземпляром X, на который дается ответ «да», то и s_Y является экземпляром Y, на который дается ответ «да».

Другими словами, тем самым устанавливается, что s_Y и s_X имеют одинаковый ответ.

Проводились исследования, направленные на изучение различий между сведениями с полиномиальным временем для специальной структуры (обращение к «черному ящику» с одним вопросом и буквальное использование полученного ответа) и более общей концепцией сведения с полиномиальным временем, при которых обращения к «черному ящику» могут производиться многократно. (Более ограниченный вариант сведения называется *сведением по Карпу*, тогда как более общий вариант называется *сведением по Куку* или *сведением по Тьюрингу* с полиномиальным временем.) Здесь эти различия рассматриваться не будут.

8.5. Задачи упорядочения

До настоящего момента рассматривались задачи, которые (как, например, задачи о независимом множестве и вершинном покрытии) подразумевали перебор подмножеств набора объектов; также были представлены задачи (как, например, 3-SAT), основанные на поиске комбинаций 0/1 в наборе переменных. Еще один тип вычислительно сложных задач связан с поиском по множеству всех *перестановок* коллекции объектов.

Задача коммивояжера

Вероятно, самой известной из задач упорядочения является *задача коммивояжера*. Представьте коммивояжера, который должен посетить n городов $v_1, v_2, ..., v_n$. Коммивояжер начинает с города v_1, в котором он живет, и хочет найти *маршрут* — порядок, в котором он посетит все остальные города и вернется домой. Требуется найти маршрут с минимальным суммарным расстоянием всех поездок.

Чтобы формализовать эту задачу, мы воспользуемся предельно обобщенной концепцией расстояния: для каждой упорядоченной пары городов (v_i,v_j) задается неотрицательное число $d(v_i, v_j)$, которое считается расстоянием от v_i до v_j. От расстояний не требуется ни симметричность (может оказаться, что $d(v_i, v_j) \neq d(v_j,v_i)$), ни выполнение неравенства треугольника (сумма $d(v_i,v_j)$ и $d(v_j,v_k)$ может быть

меньше «прямого» расстояния $d(v_i,v_k)$). Это делается для того, чтобы формулировка задачи была как можно более общей. Дело в том, что задача коммивояжера естественным образом встречается во многих приложениях, которые не имеют отношения к городам и коммивояжерам. Например, формулировка задачи коммивояжера применяется в таких областях, как планирование оптимальных перемещений манипулятора, который должен просверлить отверстия в n точках на поверхности микросхемы; или для обслуживания запросов ввода/вывода к диску; или для упорядочения выполнения n программных модулей для минимизации времени переключения контекста.

Итак, для заданного множества расстояний требуется упорядочить города в *маршрут* $v_{i_1}, v_{i_2}, ..., v_{i_n}$, с $i_1 = 1$, чтобы свести к минимуму общее расстояние $\sum_j d(v_{i_j}, v_{i_{j+1}}) + d(v_{i_n}, v_{i_1})$. Требование $i_1 = 1$ просто «ориентирует» маршрут так, чтобы он начинался в исходном городе, а слагаемые в сумме просто задают расстояние от текущего города в маршруте до следующего. (Последнее слагаемое в сумме определяет расстояние, необходимое для того, чтобы коммивояжер мог вернуться домой в конце маршрута.)

А вот как выглядит задача коммивояжера в версии с принятием решения:

Для заданного набора расстояний между n городами и границы D существует ли маршрут, длина которого не превышает D?

Задача о гамильтоновом цикле

У задачи коммивояжера имеется естественная аналогия, которая образует одну из фундаментальных задач в теории графов. Для заданного направленного графа $G = (V, E)$ цикл C в графе G называется *гамильтоновым циклом*, если он посещает каждую вершину ровно один раз, — иначе говоря, он «обходит» все вершины без повторений. Например, направленный граф на рис. 8.6 содержит несколько гамильтоновых циклов; в одном из них узлы посещаются в порядке 1, 6, 4, 3, 2, 5, 1, а в другом — в порядке 1, 2, 4, 5, 6, 3, 1.

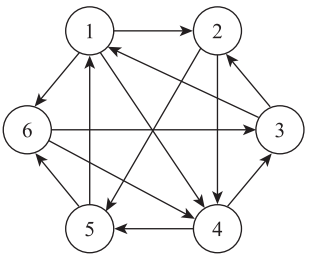


Рис. 8.6. Направленный граф, содержащий гамильтонов цикл

Задача о гамильтоновом цикле формулируется просто:

Содержит ли заданный направленный граф G гамильтонов цикл?

Доказательство *NP*-полноты задачи о гамильтоновом цикле

Сейчас мы покажем, что обе задачи являются *NP*-полными. Для этого сначала будет установлена *NP*-полнота задачи о гамильтоновом цикле, а затем задача о гамильтоновом цикле будет сведена к задаче о коммивояжере.

(8.17) Задача о гамильтоновом цикле является *NP*-полной.

Доказательство. Сначала мы покажем, что задача о гамильтоновом цикле принадлежит *NP*. Для направленного графа $G = (V, E)$ сертификатом, подтверждающим наличие решения, является упорядоченный список вершин гамильтонового цикла. По этому списку можно проверить за полиномиальное время, что каждая вершина входит в список ровно один раз, а каждая последовательная пара соединена ребром; эти проверки установят, что упорядочение определяет гамильтонов цикл.

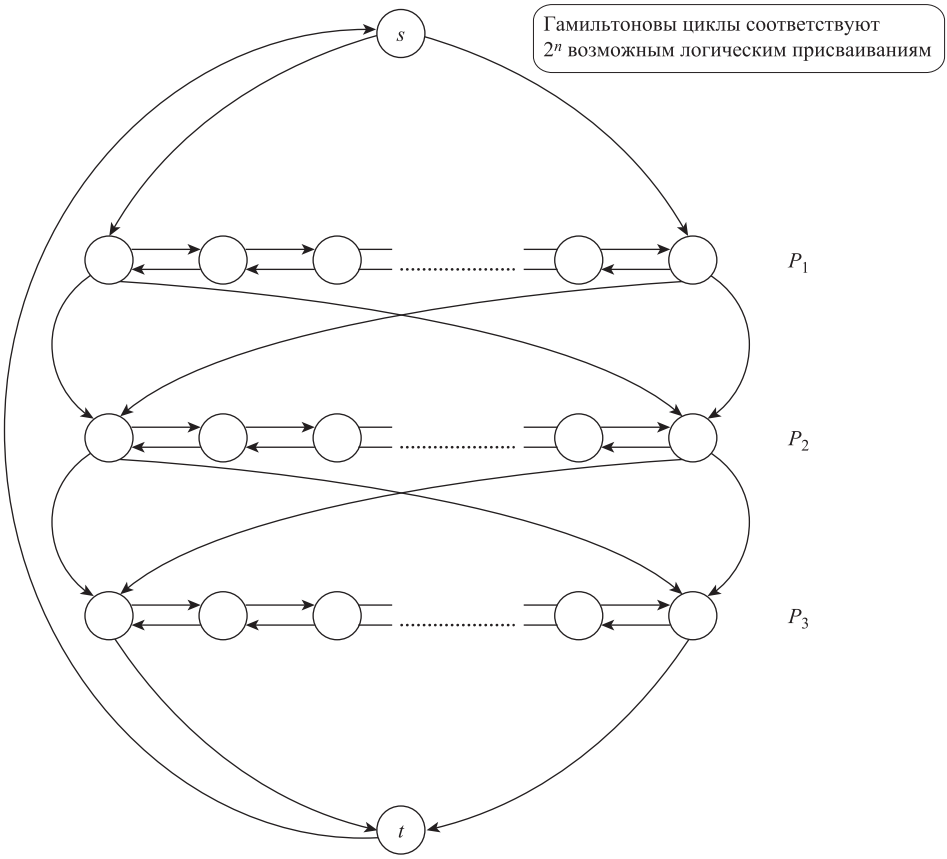


Рис. 8.7. Сведение задачи 3-SAT к задаче о гамильтоновом цикле; часть 1

Покажем, что 3-*SAT* $\leq_P$ *Гамильтонов цикл*. Почему мы выполняем сведение к 3-SAT? Столкнувшись с задачей о гамильтоновом цикле, мы на самом деле понятия не имеем, *что* выбрать для сведения; задача достаточно сильно отличается от всех задач, которые мы видели до сих пор, так что реальной основы для выбора нет. В такой ситуации одна из возможных стратегий заключается в возврате к 3-SAT, потому что комбинаторная структура этой задачи очень проста. Конечно, эта стратегия гарантирует по крайней мере некоторый уровень сложности при сведении, потому что переменные и условия необходимо будет выразить на языке графов.

Итак, рассмотрим произвольный экземпляр 3-SAT с переменными $x_1, ..., x_n$ и условиями $C_1, ..., C_k$. Необходимо показать, как решить эту задачу, если имеется возможность обнаружения гамильтоновых циклов в направленных графах. Как обычно, стоит сосредоточиться на важнейших ингредиентах 3-SAT: значения переменных можно задать по своему усмотрению, и для выполнения каждого условия есть три возможности.

Начнем с описания графа, содержащего 2^n разных гамильтоновых циклов, естественно соответствующих возможным 2^n логическим присваиваниям значений переменных. После этого мы добавим узлы для моделирования ограничений, накладываемых условиями.

Мы строим n путей $P_1, ..., P_n$, где P_i состоит из узлов $v_{i1}, v_{i2}, ..., v_{ib}$ для величины b, которая, как предполагается, немного больше количества условий k; допустим, $b = 3k + 3$. Существуют ребра из v_{ij} к $v_{i,j+1}$ и в обратном направлении, от $v_{i,j+1}$ к v_{ij}. Таким образом, обход P_i может осуществляться «слева направо» от v_{i1} к v_{ib}, или «справа налево», от v_{ib} к v_{i1}.

Эти пути связываются следующим образом: для всех $i = 1, 2, ..., n - 1$ мы определяем ребра из v_{i1} в $v_{i+1,1}$ и в $v_{i+1,b}$. Также определяются ребра из v_{ib} в $v_{i+1,1}$ и $v_{i+1,b}$. В граф добавляются два дополнительных узла s и t; мы определяем ребра из s в v_{11} и v_{1b}; из v_{n1} и v_{nb} в t; и из t в s.

Построение до настоящего момента изображено на рис. 8.7. Здесь важно задержаться и подумать, как выглядят гамильтоновы циклы на нашем графе. Так как из t выходит только одно ребро, мы знаем, что любой гамильтонов цикл C должен использовать ребро (t, s). После входа в s цикл C может идти по пути P_1 либо слева направо, либо справа налево; какое бы направление ни было выбрано, затем он идет по пути P_2 либо слева направо, либо справа налево; и т. д., вплоть до завершения P_n и входа в t. Другими словами, существуют ровно 2^n разных гамильтоновых циклов, и они соответствуют n независимым выборам направления обхода каждого P_i.

Эта структура естественно моделирует n независимых вариантов присваивания значений переменных $x_1, ..., x_n$ в экземпляре 3-SAT. Следовательно, каждый гамильтонов цикл будет однозначно идентифицироваться следующим логическим присваиванием: если C проходит P_i слева направо, то x_i присваивается 1; в противном случае x_i присваивается 0.

Теперь добавим узлы для моделирования условий; экземпляр 3-SAT является выполнимым в том, и только в том случае, если останется хотя бы один гамильтонов цикл. Рассмотрим конкретный пример — условие

$$C_1 = x_1 \vee x_2 \vee x_3.$$

На языке гамильтоновых циклов это условие означает: «Цикл должен проходить P_1 слева направо; или он должен проходить P_2 справа налево; или он должен проходить P_3 слева направо». Итак, мы добавляем узел c_1 (рис. 8.8), который делает именно это. (Некоторые ребра не показаны на рисунке для ясности.) Для некоторого значения ℓ узел c_1 имеет ребра *из* $v_{1\ell}$, $v_{2,\ell+1}$ и $v_{3\ell}$, а также ребра *в* $v_{1,\ell+1}$, $v_{2,\ell}$, $v_{3,\ell+1}$. Следовательно, он легко вставляется в любой гамильтонов цикл, который проходит P_1 слева направо, для чего узел c_1 посещается между $v_{1\ell}$ и $v_{1,\ell+1}$; аналогичным образом c_1 может быть вставлен в любой гамильтонов цикл, проходящий P_2 справа налево или P_3 слева направо. Он не может быть вставлен в гамильтонов цикл, который не делает ничего из перечисленного.

На более общем уровне узел c_j определяется для каждого условия C_j. В каждом пути P_i позиции узлов $3j$ и $3j + 1$ резервируются для переменных, участвующих в условии C_j. Предположим, условие C_j содержит литерал t. Если $t = x_i$, то добавляются ребра $(v_{i,3j}, c_j)$ и $(c_j, v_{i,3j+1})$; если $t = \overline{x_i}$, то добавляются ребра $(v_{i,3j+1}, c_j)$ и $(c_j, v_{i,3j})$.

На этом построение графа G завершается. Далее, в соответствии с приведенной выше общей схемой доказательства *NP*-полноты, мы утверждаем, что экземпляр 3-SAT выполним в том, и только в том случае, если G содержит гамильтонов цикл.

Для начала предположим, что для экземпляра 3-SAT существует выполняющее присваивание; тогда мы определяем гамильтонов цикл в соответствии с приведенным выше неформальным планом. Если x_i в выполняющем присваивании задается значение 1, то путь P_i обходится слева направо; в противном случае обход производится справа налево. Для каждого условия C_j, так как оно выполняется вследствие присваивания, будет по крайней мере один путь P_i, в котором обход будет осуществляться в «правильном» направлении относительно узла c_j, и его можно вставить в маршрут по ребрам, инцидентным $v_{i,3j}$ и $v_{i,3j+1}$.

И наоборот, предположим, что в G существует гамильтонов цикл C. Здесь важно заметить следующее: если C входит в узел c_j по ребру из $v_{i,3j}$, то он должен выходить по ребру в $v_{i,3j+1}$. В противном случае у $v_{i,3j+1}$ останется только один непосещенный сосед, а именно $v_{i,3j+2}$, поэтому маршрут не сможет посетить этот узел и сохранить гамильтоново свойство. И симметрично, если маршрут входит из $v_{i,3j+1}$, он должен немедленно выйти в $v_{i,3j}$. Следовательно, для каждого узла c_j узлы, находящиеся непосредственно до и после c_j в цикле C, соединяются ребром e в G; следовательно, если удалить c_j из цикла и вставить это ребро e для каждого j, мы получим гамильтонов цикл C' для подграфа $G - \{c_1, ..., c_k\}$. Это наш исходный подграф, до добавления узлов условий; как упоминалось ранее, любой гамильтонов цикл в этом подграфе должен полностью пройти каждый путь P_i в том или ином направлении. Соответственно мы используем C' для определения следующего логического присваивания для экземпляра 3-SAT. Если C' проходит P_i слева направо, то мы задаем $x_i = 1$; в противном случае задается $x_i = 0$. Так как больший цикл C смог посетить узел каждого условия c_j, по крайней мере один из путей проходился в «правильном» направлении относительно узла c_j, поэтому с определенным нами присваиванием выполняются все условия.

Установлено, что экземпляр 3-SAT выполним в том, и только в том случае, если G содержит гамильтонов цикл; на этом наше доказательство завершается.

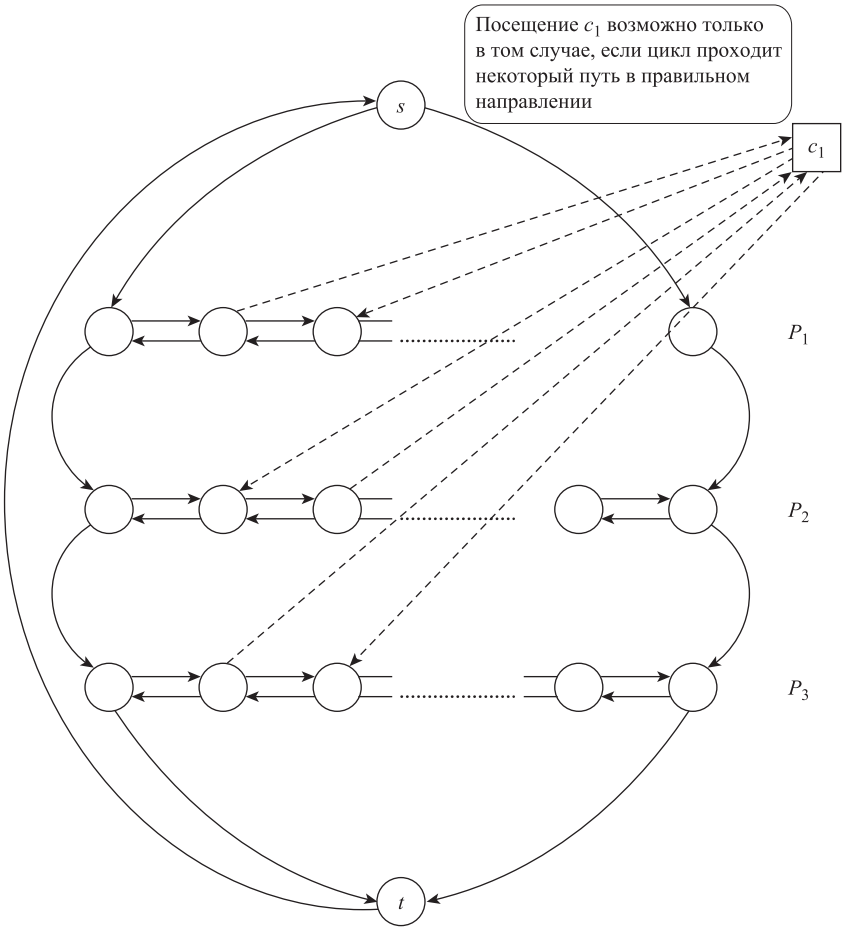


Рис. 8.8. Сведение задачи 3-SAT к задаче о гамильтоновом цикле; часть 2

Доказательство *NP*-полноты задачи коммивояжера

Вооружившись базовым результатом сложности задачи о гамильтоновом цикле, мы можем перейти к демонстрации сложности задачи коммивояжера.

(8.18) Задача коммивояжера является *NP*-полной.

Доказательство. Легко увидеть, что задача коммивояжера принадлежит *NP*: сертификат представляет собой перестановку городов, а сертифицирующий алгоритм проверяет, что длина соответствующего маршрута не превышает заданной границы.

Теперь покажем, что *Гамильтонов цикл* $\leq_P$ *Коммивояжер*. Для заданного направленного графа $G = (V,E)$ определяется следующий экземпляр задачи коммивояжера. Каждому узлу v_i графа G соответствует город v'_i. Мы определяем $d(v'_i,v'_j)$ равным 1, если в G существует ребро (v_i,v_j), и 2 — в противном случае.

Утверждается, что G содержит гамильтонов цикл в том, и только в том случае, если в экземпляре задачи коммивояжера имеется маршрут длины не более n. Если G содержит гамильтонов цикл, то упорядочение соответствующих городов определяет маршрут длины n. И наоборот, предположим, что существует маршрут длины не более n. Выражение для длины маршрута представляет собой сумму n слагаемых, каждое из которых не менее 1; следовательно, все слагаемые должны быть равны 1. Следовательно, каждая пара узлов G, соответствующих соседним городам маршрута, должна быть соединена ребром; из этого следует, что упорядочение соответствующих узлов должно образовать гамильтонов цикл. ■

Обратите внимание: возможная *асимметричность* расстояний в задаче коммивояжера ($d(v'_i,v'_j) \neq d(v'_j,v'_i)$) сыграла важнейшую роль; так как граф из экземпляра задачи о гамильтоновом цикле является направленным, наше сведение дало экземпляр задачи коммивояжера с асимметричными расстояниями.

Собственно, аналогия задачи о гамильтоновом цикле для ненаправленных графов также является *NP*-полной; хотя здесь доказательство не приводится, оно строится на относительно несложном сведении от задачи для направленного графа. При использовании задачи о гамильтоновом цикле в ненаправленном графе точная аналогия (8.18) может использоваться для доказательства *NP*-полноты задачи коммивояжера с симметричными расстояниями.

Конечно, в самом известном частным случае задачи коммивояжера расстояния определяются множеством из n точек на плоскости. Задачу о гамильтоновом цикле можно свести и к этому частному случаю, хотя сделать это намного сложнее.

Расширения: задача о гамильтоновом пути

Иногда бывает полезно рассмотреть разновидность задачи о гамильтоновом цикле, в которой не обязательно возвращаться к начальной точке. Для заданного направленного графа $G = (V, E)$ путь P в G называется *гамильтоновым путем*, если каждая вершине входит в него ровно один раз. (Путь может начинаться с любого узла и заканчиваться на любом узле при условии соблюдения указанного ограничения.) Такой путь состоит из разных узлов $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ в определенном порядке, образующих все множество вершин V; в отличие от гамильтонова цикла, наличие ребра из v_{i_n} обратно в v_{i_1} не обязательно. Задача о гамильтоновом пути формулируется так:

Содержит ли заданный направленный граф G гамильтонов путь?

Используя сложность задача о гамильтоновом цикле, мы покажем следующее.

(8.19) Задача о гамильтоновом пути является *NP*-полной.

Доказательство. Прежде всего задача о гамильтоновом пути принадлежит *NP*: сертификатом может быть путь в G, а сертифицирующий алгоритм проверяет, что он действительно является путем и каждый узел входит в него ровно один раз.

Один из способов демонстрации *NP*-полноты задачи о гамильтоновом пути заключается в сведении от 3-SAT, почти идентичного тому, которое мы использовали для задачи о гамильтоновом цикле: мы строим граф, изображенный на рис. 8.7, с тем отличием, что в него не включается ребро из t в s. Если в измененном графе нет ни одного гамильтонова пути, он должен начинаться в s (так как s не имеет входящих ребер) и завершаться в t (так как t не имеет выходящих ребер). Одно изменение позволяет более или менее дословно применить аргумент, использованный в сведении гамильтонова цикла, для обоснования того, что выполняющее присваивание для экземпляра 3-SAT существует в том, и только в том случае, если существует гамильтонов путь.

Альтернативный способ демонстрации *NP*-полноты гамильтонова пути основан на доказательстве *Гамильтонов цикл* $\leq_P$ *Гамильтонов путь*. Для заданного экземпляра задачи о гамильтоновом цикле, заданном направленным графом G, строится граф G' по следующим правилам: мы выбираем произвольный узел v в G и заменяем его двумя новыми узлами v' и v''. Все ребра, выходившие из v в G, теперь выходят из v'; а все ребра, входившие в v в G, теперь входят в v''. А точнее, каждое ребро (v, w) в G заменяется ребром (v', w); а каждое ребро (u, v) в G заменяется ребром (u, v''). На этом построение G' завершается.

Утверждается, что G' содержит гамильтонов путь в том, и только в том случае, если G содержит гамильтонов цикл. Предположим, что C — гамильтонов цикл в G; рассмотрим обход этого цикла, начиная и заканчивая в узле v. Легко увидеть, что то же упорядочение узлов образует в G' гамильтонов путь, который начинается с v' и завершается в v''. И наоборот, предположим, что P — гамильтонов путь в G'; очевидно, он должен начинаться в v' (так как v' не содержит входящих ребер) и заканчиваться в v'' (так как v'' не содержит выходящих ребер). Если заменить v' и v'' на v, то это упорядочение узлов образует гамильтонов цикл в G.

8.6. Задачи о разбиении

В следующих двух разделах рассматриваются две фундаментальные задачи о *разбиении*, в которых ищутся способы разбиения коллекции объектов на подмножества. Сейчас мы продемонстрируем *NP*-полноту задачи, которая будет называться *задачей о трехмерном сочетании*. В следующем разделе будет рассмотрена *задача раскраски графа*, связанная с разбиением узлов графа.

Задача о трехмерном сочетании

Начнем с обсуждения задачи о трехмерном сочетании, которая может считаться усложненной версией задачи о двудольном паросочетании, которая разбиралась ранее. Задача о двудольном паросочетании может рассматриваться следующим образом: даны два множества X и Y, каждое из которых имеет размер n, и множество P пар из $X \times Y$. Вопрос: существует ли в P такое множество из n пар, в котором

каждый элемент в $X \cup Y$ содержится ровно в одной паре? Связь с задачей двудольного паросочетания очевидна: множество P таких пар просто соответствует ребрам двудольного графа.

Мы уже знаем, как решать задачу двудольного паросочетания за полиномиальное время. Однако ситуация заметно усложняется при переходе от упорядоченных пар к упорядоченным триплетам. Рассмотрим следующую задачу о трехмерном сочетании:

> Для заданных непересекающихся множеств X, Y и Z, каждое из которых имеет размер n, и заданного множества $T \subseteq X \times Y \times Z$ упорядоченных триплетов существует ли в T такое множество из n триплетов, что каждый элемент $X \cup Y \cup Z$ содержится ровно в одном из этих триплетов?

Такой набор триплетов называется *идеальным трехмерным сочетанием*.

Интересный аспект задачи о трехмерном сочетании, кроме ее связи с двудольным паросочетанием, заключается в том, что она одновременно образует частные случаи как задачи покрытия множества, так и задачи упаковки множества: фактически мы ищем покрытие универсального множества $X \cup Y \cup Z$ набором непересекающихся множеств. Конкретнее, задача о трехмерном сочетании является частным случаем задачи покрытия множества, поскольку мы ищем покрытие универсального множества $U = X \cup Y \cup Z$ с использованием не более n множеств из заданного набора (триплетов). Аналогичным образом задача о трехмерном сочетании является частным случаем задачи упаковки множества, так как мы ищем n непересекающихся подмножеств универсального множества $U = X \cup Y \cup Z$.

Доказательство *NP*-полноты трехмерного сочетания

Приведенные выше рассуждения легко преобразуются в доказательства того, что *Трехмерное сочетание* $\leq_P$ *Покрытие множества* и *Трехмерное сочетание* $\leq_P$ *Упаковка множества*. Но это не поможет установить *NP*-полноту задачи о трехмерном сочетании, потому что эти сведения просто показывают, что задача о трехмерном сочетании может быть сведена к некоторым очень сложным задачам. Нам нужно провести доказательство в обратном направлении: что известная *NP*-полная задача может быть сведена к задаче о трехмерном сочетании.

(8.20) Задача о трехмерном сочетании является *NP*-полной.

Доказательство. Как и следовало ожидать, принадлежность задачи о трехмерном сочетании *NP* доказывается легко. Для заданного набора триплетов $T \subset X \times Y \times Z$ сертификатом, подтверждающим решение, может быть набор триплетов $T' \subseteq T$. За полиномиальное время можно убедиться в том, что каждый элемент в $X \cup Y \cup Z$ принадлежит ровно одному из триплетов в T'.

Для сведения мы снова возвращаемся к задаче 3-SAT. Пожалуй, это выглядит более странно, чем в случае с гамильтоновым циклом, из-за тесной связи задачи о трехмерном сочетании с задачами упаковки множества и покрытия множества; но в действительности закодировать требования к разбиению в любой из этих

задач очень трудно. Итак, рассмотрим произвольный экземпляр 3-SAT с n переменными $x_1, ..., x_n$ и k условиями $C_1, ..., C_k$. Мы покажем, как решить его, если существует возможность обнаружения идеальных трехмерных сочетаний.

Общая стратегия такого сведения очень похожа (на очень высоком уровне) на метод, который использовался в сведении 3-SAT к гамильтоновому циклу. Сначала мы спроектируем регуляторы для кодирования независимых вариантов выбора, задействованных в логическом присваивании каждой переменной; затем будут добавлены регуляторы для кодирования ограничений, наложенных условиями. В этом построении все элементы экземпляра трехмерного сочетания будут изначально называться просто «элементами» без указания того, берутся ли они из X, Y или Z. В конце они естественным образом раскладываются на эти три множества.

Рассмотрим базовый регулятор, связанный с переменной x_i. Определим элементы $A_i = \{a_{i1}, a_{i2}, ..., a_{i,2k}\}$, образующие *ядро* регулятора; определим элементы $B_i = \{b_{i1}, ..., b_{i,2k}\}$ на *концах* регулятора. Для всех $j = 1, 2, ..., 2k$ определяется триплет $t_{ij} = (a_{ij}, a_{i\,j+1}, b_{ij})$ с интерпретацией сложения по модулю $2k$. Три таких регулятора изображены на рис. 8.9. В регуляторе i триплет t_{ij} называется *четным*, если j четно, или *нечетным*, если j нечетно. Аналогичным образом конец b_{ij} будет называться *четным* или *нечетным*.

Это будут единственные триплеты, содержащие элементы A_i, поэтому мы уже можем что-то сказать о том, как они должны покрываться в любом идеальном сочетании: необходимо использовать либо все четные триплеты в регуляторе i, либо все нечетные триплеты в гаджете i. На этой идее основан наш метод кодирования идеи о том, что x_i задается значение 0 или 1; выбор всех четных триплетов представляет назначение $x_i = 0$, а выбор всех нечетных триплетов представляет назначение $x_i = 1$.

Решение о четности/нечетности также может рассматриваться с другой точки зрения — в контексте концов регуляторов. Решив использовать четные триплеты, мы покрываем четные концы регуляторов и оставляем нечетные концы свободными; для нечетных триплетов покрываются четные концы регуляторов, а нечетные концы остаются свободными. Таким образом, решение о присваивании x_i можно рассматривать следующим образом: свободные нечетные концы соответствуют 0, тогда как свободные четные концы соответствуют 1. Пожалуй, такое представление упростит понимание оставшейся части построения.

Пока что выбор четности/нечетности может приниматься независимо для каждого из n регуляторов переменных. Теперь добавим элементы для моделирования условий и ограничения выбираемых вариантов присваивания. Как и в доказательстве (8.17), рассмотрим пример условия

$$C_1 = x_1 \vee \overline{x_2} \vee x_3.$$

На языке трехмерных сочетаний это означает: «Сочетание по центрам регуляторов должно оставить свободными четные концы первого регулятора; или оно должно оставить свободными нечетные концы второго регулятора; или оно должно оставить свободными нечетные концы третьего регулятора». Итак, мы добавляем *регулятор условия*, который делает именно это. Он состоит из множества двух элементов ядра $P_1 = \{p_1, p'_1\}$ и трех триплетов, их содержащих. Один триплет имеет форму (p_1, p'_1, b_{1j}) для четного конца b_{1j}; другой включает p_1, p'_1 и нечетный конец $b_{2j'}$;

и третий включает p_1, p'_1 и четный конец $b_{3,j}''$. Только эти три триплета покрывают P_1, поэтому мы знаем, что один из них должен использоваться; тем самым точно обеспечивается соблюдение условия.

В общем случае для условия C_j создается регулятор с двумя элементами ядра $P_j = \{p_j, p'_j\}$, а также определяются три триплета, содержащие P_j. Предположим, условие C_j содержит литерал t. Если $t = x_i$, мы определяем триплет $(p_j, p'_j, b_{i,2j})$; если $t = \overline{x_i}$, то определяется триплет $(p_j, p'_j, b_{i,2j-1})$. Только регулятор условия j использует концы b_{im} с $m = 2j$ или $m = 2j - 1$; таким образом, регуляторы условий никогда не «конкурируют» друг с другом за свободные концы.

Построение почти закончено, но осталась еще одна проблема. Допустим, множество условий имеет выполняющее присваивание. В этом случае для каждого регулятора переменной принимается соответствующий выбор четности/нечетности; для каждого регулятора условия остается как минимум один свободный конец, так что все элементы ядер регуляторов условий также получают покрытие. Проблема в том, что *покрытие получили еще не все концы*. Мы начали с $n \cdot 2k = 2nk$ концов; триплеты $\{t_{ij}\}$ обеспечили покрытие nk из них, а регуляторы условий — еще k. Остается обеспечить покрытие еще $(n - 1)k$ концов.

Проблема решается очень простым приемом: в конструкцию добавляются $(n - 1)k$ «регуляторов завершения». Регулятор завершения i состоит из двух элементов ядра $Q_i = \{q_i, q'_i\}$ и триплета (q_i, q_i, b) для *каждого* конца b в каждом регуляторе переменной. Это последняя часть построения.

Итак, если множество условий имеет выполняющее присваивание, то в каждом регуляторе переменной делается соответствующий выбор четности/нечетности; как и прежде, при этом остается как минимум один свободный конец для каждого регулятора условия. Использование регуляторов завершения для покрытия всех оставшихся концов гарантирует, что покрытие обеспечено для всех элементов ядер в регуляторах переменных, условий и завершения, а также для всех концов.

И наоборот, предположим, что в построенном экземпляре существует идеальное трехмерное сочетание. Тогда, как упоминалось выше, в каждом регуляторе переменной сочетание выбирает либо все четные $\{t_{ij}\}$, либо все нечетные $\{t_{ij}\}$. В первом случае в экземпляре 3-SAT задаются $x_i = 0$, а во втором — $x_i = 1$. Теперь рассмотрим условие C_j; было ли оно выполнено? Так как два элемента ядра в P_j получили покрытие, по крайней мере один из трех регуляторов переменных, соответствующих литералам из C_j, сделал «правильный» выбор решения четности/нечетности, что привело к присваиванию, удовлетворяющему C_j.

Доказательство практически завершено, осталось ответить на последний вопрос: действительно ли мы сконструировали экземпляр трехмерного сочетания? Имеется набор элементов и триплеты, содержащие некоторые из них, но действительно ли эти элементы можно разбить на соответствующие множества X, Y и Z равного размера?

К счастью, ответ на этот вопрос положителен. Мы можем определить X как множество всех a_{ij} с четными j, множество всех p_j и множество всех q_i. Y определяется как множество всех a_{ij} с нечетными j, множество всех p'_j и множество всех q'_i. Наконец, Z определяется как множество всех концов b_{ij}. Легко убедиться в том, что каждый триплет содержит по одному элементу из множеств X, Y и Z. ■

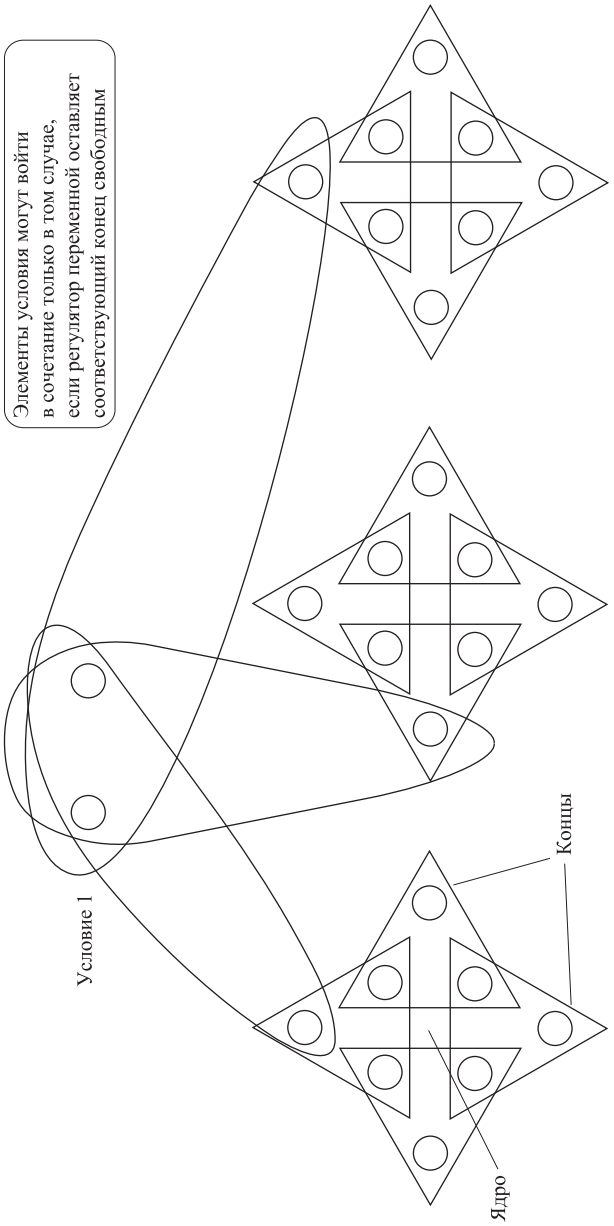


Рис. 8.9. Сведение задачи 3-SAT к задаче о трехмерном сочетании

8.7. Задача о раскраске графа

При раскрашивании карты (например, стран на карте мира) желательно раскрасить соседние области в разные цвета, чтобы их границы были четко видны, — но при этом, чтобы избежать лишней пестроты, использовать минимальное количество цветов. В середине XIX века Фрэнсис Гатри заметил, что карту графств Англии можно раскрасить таким образом всего четырьмя цветами, и его заинтересовало, обладает ли этим свойством любая карта. Он спросил своего брата, который передал вопрос одному из своих профессоров; так родилась знаменитая математическая задача: *гипотеза четырех цветов*.

Задача о раскраске графа

Под *раскраской графа* понимается аналогичный процесс с ненаправленным графом G, в котором узлы играют роль раскрашиваемых областей, а ребра представляют соседние пары. Требуется назначить цвет каждому узлу G так, чтобы при наличии ребра (u, v) узлам u и v были назначены разные цвета; целью является выполнение этих условий с минимальным набором цветов. В более формальном варианте k-раскраской G называется такая функция $f: V \to \{1, 2, ..., k\}$, что для каждого ребра (u, v) выполняется $f(u) = f(v)$. (Таким образом, цвета обозначаются 1, 2, ..., k, а функция f представляет выбор цвета для каждого узла.) Если для G существует k-раскраска, он называется *k-раскрашиваемым*.

В отличие от карт на плоскости, вполне очевидно, что не существует фиксированной константы k, с которой любой граф имеет k-раскраску: например, если взять множество из n узлов и соединить каждую пару ребром, то для раскраски полученного графа потребуется n цветов. Тем не менее алгоритмическая версия задачи очень интересна:

Для заданного графа G и границы k имеет ли граф G k-раскраску?

Будем называть эту задачу *задачей раскраски графа* — или *k-раскраски*, когда мы хотим подчеркнуть конкретный выбор k.

Задача раскраски графа находит очень широкий диапазон применения. Хотя реальная потребность в ней в картографии пока неочевидна, эта задача естественным образом возникает в ситуациях с распределением ресурсов при наличии конфликтов.

- Допустим, имеется набор из n процессов в системе, которая позволяет выполнять несколько заданий в параллельном режиме. При этом некоторые пары заданий не могут выполняться одновременно, потому что им нужен доступ к некоторому ресурсу. Для следующих k временных квантов требуется спланировать выполнение процессов, чтобы каждый процесс выполнялся минимум в одном из них. Возможно ли это? Если построить граф G на базе множества процессов, соединяя два процесса ребром при наличии конфликта, то k-раскраска G будет представлять план выполнения, свободный от конфликтов: все узлы, окрашенные в цвет j, будут выполняться на шаге j, а вся конкуренция за ресурсы будет исключена.

- Другое известное применение задачи встречается при разработке компиляторов. Предположим, при компиляции программы мы пытаемся связать каждую переменную с одним из k регистров. Если две переменные используются одновременно, они не могут быть связаны с одним регистром (в противном случае присваивание одной переменной приведет к потере значения другой). Для множества переменных строится граф G; две переменные соединяются ребром в том случае, если они используются одновременно. K-раскраска G соответствует безопасному способу распределения переменных между регистрами: все узлы с раскраской j могут быть связаны с регистром j, так как никакие два из них не используются одновременно.
- Третий пример встречается при распределении частот для беспроводной связи: требуется назначить одну из k частот каждому из n устройств; если два устройства находятся достаточно близко друг к другу, им должны быть присвоены разные длины волн для предотвращения помех. Для решения задачи строится граф G для множества устройств; два узла соединяются в том случае, если они расположены достаточно близко для создания помех. В этом случае k-раскраска графа представляет такое назначение частот, при котором все узлы, работающие на одной частоте, находятся достаточно далеко друг от друга, чтобы избежать помех. (Интересно, что в этом применении задачи раскраски графа узлам назначаются позиции электромагнитного спектра — иначе говоря, в широком смысле это действительно цвета.)

Вычислительная сложность задачи о раскраске графа

Какую сложность имеет задача k-раскраски? Прежде всего, в случае $k = 2$ возникает задача, которую мы уже видели в главе 3. Вспомните, что мы рассматривали задачу определения того, является ли граф G двудольным, и показали, что она эквивалентна следующему вопросу: можно ли раскрасить узлы G в красный и синий цвета так, чтобы у каждого ребра один конец был красным, а другой синим?

Но последний вопрос в точности соответствует задаче о раскраске графа для $k = 2$ цветам (красный и синий). Таким образом, мы показали, что

(8.21) Граф G является 2-раскрашиваемым в том, и только в том случае, если он является двудольным.

Это означает, что алгоритм из раздела 3.4 может использоваться для принятия решения о том, является ли входной граф 2-раскрашиваемым, за время $O(m + n)$, где n — количество узлов G, а m — количество ребер.

Стоит перейти к $k = 3$ цветам, как ситуация значительно усложняется. Никакого простого эффективного алгоритма для задачи 3-раскраски не видно, и рассуждать об этой задаче вообще сложно. Например, изначально может возникнуть впечатление, что в любом графе, не являющиеся 3-раскрашиваемым, присутствует «доказательство» в форме четырех узлов, являющихся взаимно смежными (для которых потребуются четыре разных цвета) — но это не так. Например, граф на рис. 8.10

не имеет 3-раскраски по более тонкой (хотя и объяснимой) причине; более того, можно привести намного более сложные графы, не являющиеся 3-раскрашиваемыми по причинам, для которых трудно найти компактное объяснение.

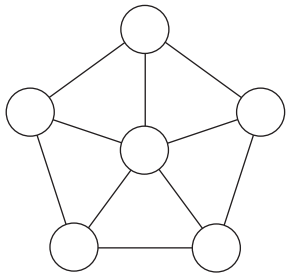


Рис. 8.10. Граф, не имеющий 3-раскраски

Более того, как вы сейчас убедитесь, в случае трех цветов задача уже становится очень сложной.

Доказательство *NP*-полноты задачи о 3-раскраске

(8.22) Задача о 3-раскраске является *NP*-полной.

Доказательство. Легко увидеть, почему задача принадлежит *NP*. Для заданных G и k одним из сертификатов, подтверждающих истинность ответа, является k-раскраска: за полиномиальное время можно убедиться в том, что в раскраске используется не более k цветов и никакая пара узлов, соединенных ребром, не окрашена в один цвет.

Как и другие задачи в этом разделе, задачу о 3-раскраске графа трудно связать с другой *NP*-полной задачей, виденной ранее, поэтому мы снова вернемся к 3-SAT. Заданный экземпляр 3-SAT с переменными $x_1, ..., x_n$ и условиями $C_1, ..., C_k$ будет решен с использованием «черного ящика» для решения задачи 3-раскраски.

Начало сведения выглядит вполне ожидаемо. Возможно, основное достоинство 3-раскраски при кодировании булевых выражений заключается в том факте, что мы можем связать узлы графа с конкретными литералами, а соединяя узлы ребрами, можно гарантировать, что им будут назначены разные цвета; это обстоятельство позволяет связать с одним узлом истинное, а с другим — ложное значение. С учетом сказанного мы определяем узлы и $\overline{v_i}$, соответствующие каждой переменной и ее отрицанию $\overline{x_i}$. Также определяются три «специальных узла» T, F и B (сокращения от *True*, *False* и *Base*).

Для начала мы соединим каждую пару узлов v_i, $\overline{v_i}$ ребром и соединим оба этих узла с *Base* (в результате чего образуется треугольник из v_i, $\overline{v_i}$ и *Base* для каждого i). Также *True*, *False* и *Base* соединяются в треугольник. Простой граф G, определенный к настоящему моменту, изображен на рис. 8.11, и он уже обладает рядом полезных свойств.

- В любой 3-раскраске графа G узлам v_i и $\overline{v_i}$ должны быть назначены разные цвета, и оба они должны отличаться от цвета *Base*.
- В любой 3-раскраске графа G узлам *True*, *False* и *Base* должны быть назначены все три цвета в некоторой перестановке. В дальнейшем эти три цвета будут называться цветами *True*, *False* и *Base* в зависимости от того, какому из узлов соответствует тот или иной цвет. В частности, это означает, что для всех i одно из значений v_i или $\overline{v_i}$ получает цвет *True*, а другому достается цвет *False*. В оставшейся части этого построения будем считать, что переменной x_i значение 1 в заданном экземпляре 3-SAT присваивается в том, и только в том случае, если узлу v_i назначается цвет *True*.

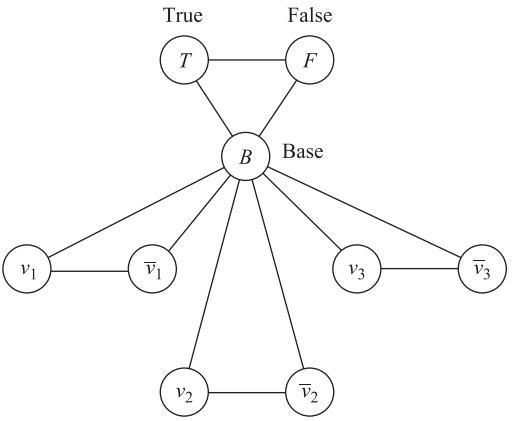


Рис. 8.11. Начало сведения для задачи о 3-раскраске

Короче говоря, мы получили граф G, в котором любая 3-раскраска неявно определяет логическое присваивание для переменных в экземпляре 3-SAT. Теперь необходимо нарастить G так, чтобы только выполняющие присваивания могли быть расширены до 3-раскрасок полного графа. Как это сделать?

Как и в других сведениях 3-SAT, рассмотрим условие вида $x_1 \vee \overline{x_2} \vee x_3$. На языке 3-раскраски графа G это означает: «По крайней мере одному из узлов v_1, $\overline{v_2}$ или v_3 должен быть назначен цвет *True*». Итак, нам нужен маленький подграф, который можно присоединить к G, чтобы любая 3-раскраска, расширяющаяся на этот подграф, обладала свойством назначения цвета *True* по крайней мере одному из узлов v_1, $\overline{v_2}$ или v_3. Найти такой подграф удается не сразу, но один из работающих вариантов изображен на рис. 8.12.

Этот подграф из шести узлов «присоединяется» к основному графу G в пяти существующих узлах: *True*, *False* и узлах, соответствующих трем литералам в условии, которое мы пытаемся представить (в данном случае v_1, $\overline{v_2}$ и v_3). Теперь предположим, что в некоторой 3-раскраске G всем трем узлам v_1, $\overline{v_2}$ или v_3 назначен цвет *False*. Тогда двум нижним затемненным узлам подграфа должен достаться цвет *Base*, а трем затемненным узлам над ними — соответственно цвета *False*, *Base*

и *True*, а следовательно, для верхних затемненных узлов цветов уже не останется. Другими словами, 3-раскраска, в которой ни одному из узлов v_1, $\overline{v_2}$ или v_3 не назначается цвет *True*, не может быть расширена до 3-раскраски этого подграфа[1].

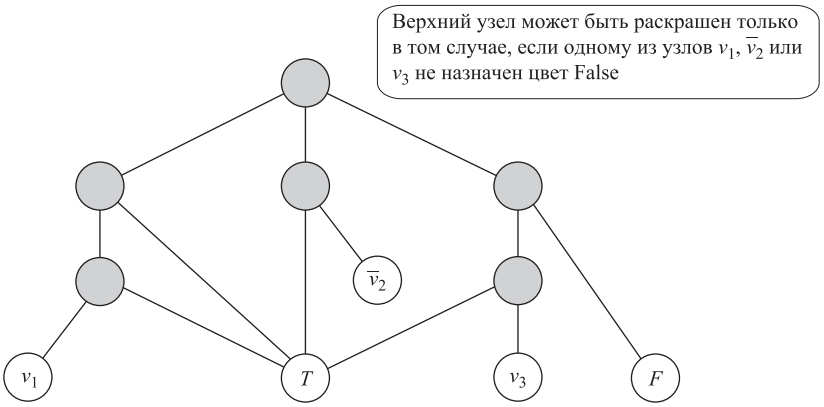


Рис. 8.12. Присоединение подграфа для представления условия $x_1 \vee \overline{x_2} \vee x_3$

Наконец, ручная проверка показывает, что если одному из узлов v_1, $\overline{v_2}$ или v_3 назначен цвет *True*, весь подграф может иметь 3-раскраску.

Далее остается лишь завершить построение: мы начинаем с графа G, определенного выше, и для каждого условия в экземпляре 3-SAT присоединяем подграф из шестви узлов, изображенный на рис. 8.12. Назовем полученный граф G'.

Утверждается, что заданный экземпляр 3-SAT выполним в том, и только в том случае, если граф G' имеет 3-раскраску. Сначала предположим, что для экземпляра 3-SAT существует выполняющее присваиванием. Определим раскраску G', окрасив *Base*, *True* и *False* в три разных цвета, а затем для каждого i назначим v_i цвет *True*, если $x_i = 1$, или цвет *False*, если $x_i = 0$. После этого $\overline{v_i}$ назначается единственный свободный цвет. Наконец, как объяснялось выше, теперь эта 3-раскраска может быть расширена на каждый подграф условия, состоящий из шести узлов, что приведет к 3-раскраске всех G'.

И наоборот, предположим, что у G' существует 3-раскраска. В этой раскраске каждому узлу v_i назначается либо цвет *True*, либо цвет *False*; переменная x_i задается соответствующим образом. Теперь утверждается, что в каждом условии экземпляра

[1] Этот аргумент дает представление о том, как был получен этот подграф. Цель заключается в том, чтобы верхний узел не мог получить никакой цвет. Мы начинаем с «присоединения» трех узлов, соответствующих литералам (все из которых окрашены в цвет False), в нижнем ряду. Затем для каждого узла мы продвигаемся вверх и ставим его в пару с узлом известного цвета, чтобы узлу сверху достался третий цвет. Продолжая таким образом, мы придем к узлу, которому гарантированно будет назначен нужный цвет. Соответственно мы форсируем назначение трех разных цветов, начиная с трех разных литералов, а затем подводим все три разноцветных узла к верхнему узлу, создавая ситуацию с невозможностью назначения цвета.

3-SAT по крайней мере один из литералов в условии имеет значение истинности 1. В противном случае всем трем соответствующим узлам в 3-раскраске G' был бы назначен цвет *False*, но, как было показано выше, в такой ситуации не существует 3-раскраски соответствующего подграфа условия, — возникает противоречие. ■

При $k > 3$ задача о 3-раскраске очень легко сводится к k-раскраске. Фактически все, что требуется, — взять экземпляр задачи 3-раскраски, представленный графом G, добавить $k - 3$ новых узлов и соединить эти новые узлы друг с другом и с каждым узлом в G. Полученный граф является k-раскрашиваемым в том, и только в том случае, если исходный граф G является 3-раскрашиваемым. Следовательно, k-раскраска для всех $k > 3$ также является *NP*-полной.

Заключение: о проверке гипотезы четырех цветов

В завершение этого раздела стоит рассказать и о том, как закончилась история гипотезы четырех цветов для карт на плоскости. Более чем через 100 лет гипотеза была доказана Аппелем и Хакеном в 1976 году. Структура доказательства была простой индукцией по количеству областей, но шаг индукции включал около двух тысяч относительно сложных случаев, проверку которых пришлось поручить компьютеру. Многие математики были недовольны таким результатом: они надеялись получить доказательство, которое бы давало представление о том, *почему* гипотеза была истинной, — а вместо этого получили перебор запредельной сложности, проверку которого нельзя было осуществить вручную. Задача поиска относительно короткого, доступного доказательства все еще остается открытой.

8.8. Численные задачи

А теперь рассмотрим некоторые вычислительно сложные задачи, основанные на арифметических операциях с числами. Как вы увидите, вычислительная неразрешимость в данном случае обусловлена способом кодирования некоторых из представленных ранее задач в представлении очень больших целых чисел.

Задача о суммировании подмножеств

Базовой задачей в этой категории будет задача о *суммировании подмножеств* — частный случай задачи о рюкзаке, рассмотренной в разделе 6.4 при рассмотрении динамического программирования. Версия этой задачи с принятием решения может быть сформулирована следующим образом:

> Задано множество натуральных чисел $w_1, ..., w_n$ и целевое число W. Существует ли подмножество $\{w_1, ..., w_n\}$, сумма которого равна ровно W?

Мы уже видели алгоритм для решения этой задачи; почему же сейчас она включается в список вычислительно сложных задач? Все возвращается к проблеме, которая была впервые упомянута при знакомстве с задачей о суммировании

подмножеств в разделе 6.4. Разработанный тогда алгоритм выполнялся за время $O(nW)$, которое может быть разумным при малых W, но становится безнадежно неприемлемым при больших W (и числах w_i). Для примера возьмем экземпляр со 100 числами, длина каждого из которых составляет 100 бит. В этом случае входные данные состоят всего из $100 \times 100 = 10\,000$ цифр, но значение W равно приблизительно 2^{100}.

В более общем виде эту проблему можно сформулировать, так как целые числа обычно будут задаваться в двоичном представлении, величина W в действительности экспоненциально зависит от размера входных данных; наш алгоритм не был алгоритмом с полиномиальным временем. (Тогда он был назван *псевдополиномиальным* для обозначения того факта, что он выполняется за время, полиномиальное по значению входных чисел, но не полиномиальное по размеру их представления.)

Эта проблема встречается во многих ситуациях; например, мы встречали ее в контексте алгоритмов сетевого потока, в которых пропускные способности были целочисленными. Возможно, другие примеры тоже покажутся вам знакомыми — например, безопасность такой криптографической системы, как RSA, обусловлена сложностью факторизации числа из 1000 битов. Но если бы время выполнения из 2^{1000} шагов считалось приемлемым, факторизация такого числа не создавала бы никаких трудностей.

Здесь стоит ненадолго задержаться и спросить себя: действительно ли это представление полиномиального времени для числовых операций является таким уж жестким ограничением? Например, для двух натуральных чисел w_1 и w_2, представленных в записи по основанию d (для некоторого $d > 1$), сколько времени займут операции сложения, вычитания или умножения? Эта тема была затронута в разделе 5.5, когда мы заметили, что стандартный способ выполнения этих операций, изучаемый на школьных уроках математики, имеет полиномиальное время выполнения (с малым показателем степени). Сложение и вычитание (с переносом) выполняется за время $O(\log w_1 + \log w_2)$, тогда как стандартный алгоритм умножения выполняется за время $O(\log w_1 \cdot \log w_2)$. (Вспомните, что в разделе 5.5 обсуждался асимптотически быстрый алгоритм умножения, который дети вряд ли самостоятельно откроют на уроках математики.)

Итак, основной вопрос: возможно ли решение задачи о суммировании подмножеств алгоритмом с (действительно) полиномиальным временем? Другими словами, существует ли алгоритм с временем выполнения, полиномиальным по n и $\log W$? Или полиномиальным только по n?

Доказательство *NP*-полноты задачи о суммировании подмножеств

Следующий результат наводит на мысль, что существование такого алгоритма маловероятно.

(8.23) Задача о суммировании подмножеств является *NP*-полной.

Доказательство. Сначала покажем, что задача о суммировании подмножеств принадлежит *NP*. Для заданных натуральных чисел $w_1, ..., w_n$ и целевого значения

W сертификатом, подтверждающим существование решения, будет подмножество $w_{i_1}, ..., w_{i_k}$, которое в сумме дает W. За полиномиальное время мы можем вычислить сумму этих чисел и убедиться в том, что она равна W.

Теперь попробуем свести заведомо *NP*-полную задачу к задаче о суммировании подмножеств. Так как мы ищем множество, которое в сумме дает заданную величину (а не ограничивается ею сверху или снизу), речь идет о комбинаторной задаче, основанной на получении точного значения. Задача о трехмерном сочетании является естественным кандидатом; мы покажем, что *Трехмерное сочетание* $\leq_P$ *Сумма подмножеств*. Основная хитрость будет связана со способом кодирования операций с множествами посредством суммирования целых чисел.

Рассмотрим экземпляр задачи о трехмерном сочетании, определяемый множествами X, Y, Z, каждое из которых имеет размер n, и множеством m триплетов $T \subseteq X \times Y \times Z$. Стандартный способ представления множеств основан на использовании *битовых векторов*: каждый элемент векторов соответствует разным элементам множества, и равен 1 в том, и только в том случае, если множество содержит этот элемент. Мы воспользуемся этим методом для представления триплетов $t = (x_i, y_j, z_k) \in T$: мы строим число w_t с $3n$ цифрами, которое содержит 1 в позициях i, $n + j$ и $2n + k$ и 0 во всех остальных позициях. Другими словами, для некоторого основания $d > 1$ $w_t = d^{i-1} + d^{n+j-1} + d^{2n+k-1}$.

Обратите внимание на то, что вычисление объединения триплетов *почти* соответствует целочисленному сложению: единицы заполняют те позиции, в которых любое из множеств содержит элементы. Мы говорим «почти», потому что при сложении используются переносы: лишние единицы в одном столбце «переносятся» и образуют ненулевой элемент в следующем столбце. В контексте операции объединения у переноса аналогов нет.

В текущей ситуации эта проблема решается простым приемом. Всего используется m чисел, каждое состоит из цифр 0 или 1; если предположить, что наши числа записаны по основанию $d = m + 1$, то переносов не будет вообще.

Построим следующий экземпляр задачи о суммировании подмножеств. Для каждого триплета $t = (x_i, y_j, z_k) \in T$ строится число w_t в записи по основанию $m + 1$, как определяется выше. Затем W определяется как число в записи по основанию $m + 1$ с $3n$ цифрами, каждая из которых равна 1, то есть $W = \sum_{j=0}^{3n-1} (m+1)^j$.

Утверждается, что множество T триплетов содержит идеальное трехмерное сочетание в том, и только в том случае, если существует подмножество чисел $\{w_t\}$, которое дает в сумме W. Предположим, существует идеальное трехмерное сочетание триплетов $t_1, ..., t_n$. В этом случае в сумме $w_{t_1} + \cdots + w_{t_n}$ в каждой из $3n$ позиций находится одна единица, поэтому результат равен W. И наоборот, предположим, существует множество чисел $w_{i_1}, ..., w_{i_k}$, которое в сумме дает W. Тогда, поскольку в представлении каждого w_{i_j} содержатся три единицы и переносы отсутствуют, мы знаем, что $k = n$. Следовательно, для каждой из $3n$ позиций цифр ровно одно из w_{i_j} содержит 1 в этой позиции. Следовательно, $t_1, ..., t_k$ образуют идеальное трехмерное сочетание. ■

Расширения: сложность некоторых задач планирования

Сложность задачи о суммировании подмножеств может использоваться для установления сложности целого класса задач планирования, включая некоторые задачи, не связанные с суммированием чисел. Ниже приведен хороший пример: естественное (но намного более сложное) обобщение задачи планирования, решенной нами в разделе 4.2 с использованием жадного алгоритма.

Допустим, дано множество из n задач, которые должны выполняться на одном компьютере. Для каждого задания i установлено *время доступности* r_i, когда оно впервые становится доступным для обработки; *предельное время* d_i, к которому оно должно быть завершено; и *продолжительность обработки* t_i. Будем считать, что все эти параметры являются целыми числами. Чтобы задание i было завершено, ему должен быть выделен смежный набор из t_i единиц времени где-то в интервале $[r_i, d_i]$. В любой момент на машине может выполняться только одно задание. Вопрос заключается в следующем: можно ли спланировать все задания для выполнения так, чтобы каждое задание было завершено к предельному времени? Назовем этот экземпляр *задачей планирования с временем доступности и предельным временем*.

(8.24) Задача планирования с временем доступности и предельным временем является *NP*-полной.

Доказательство. Для заданного экземпляра задачи сертификатом, доказывающим наличие решения, будет список начального времени выполнения для каждого задания. При наличии такого списка можно убедиться в том, что каждое задание выполняется в четко определенный интервал времени между временем доступности и предельным временем. Следовательно, задача принадлежит *NP*.

Теперь покажем, что задача о суммировании подмножеств сводится к этой задаче планирования. Рассмотрим экземпляр задачи о суммировании подмножеств с числами w_1, ..., w_n и целевым значением W. При построении эквивалентного экземпляра задачи планирования сначала бросается в глаза обилие параметров, которыми необходимо управлять: время доступности, предельное время и продолжительность. Здесь важно пожертвовать большей частью этой гибкости и получить «облегченный» экземпляр задачи, который, тем не менее, кодирует задачу о суммировании подмножеств.

Пусть $S = \sum_{i=1}^{n} w_i$. Мы определяем задания 1, 2, ..., n; задание i имеет время доступности 0, предельное время S+1 и продолжительность w_i. Задания из этого множества можно расположить в любом порядке, и все они завершатся вовремя.

Теперь мы установим дополнительные ограничения для экземпляра, чтобы решить его можно было только группировкой подмножества заданий, продолжительности которых в сумме дают ровно W. Для этого мы определяем $(n + 1)$-е задание с временем доступности W, предельным временем $W + 1$ и продолжительностью 1.

Рассмотрим любое действительное решение этого экземпляра задачи планирования. $(N + 1)$-е задание должно выполняться в интервале $[W, W + 1]$. Между общим временем доступности и общим предельным временем свободны S единиц времени; и суммарная продолжительность заданий также равна S. Таким образом, на машине

не должно быть простоев, когда не выполняется ни одно задание. В частности, если до времени W выполняются задания i_1, ..., i_k, то соответствующие числа $w_{i_1}, \ldots, w_{i_k}$ в экземпляре задачи о суммировании подмножеств в сумме дают ровно W.

И наоборот, если существуют числа $w_{i_1}, \ldots, w_{i_k}$, которые в сумме дают ровно W, их можно распланировать до задания $n + 1$, а остальные — после задания $n + 1$; и это расписание будет действительным решением для экземпляра задачи планирования. ■

Внимание: суммирование подмножеств с полиномиально ограничиваемыми числами

С задачей о суммировании подмножеств связан очень распространенный источник ошибок. И хотя он тесно связан с проблемами, о которых уже говорилось ранее, мы считаем, что эта ловушка заслуживает явного упоминания.

Рассмотрим особый случай задачи о суммировании подмножеств с n входными числами, в котором W ограничивается полиномиальной функцией n. В предположении, что $P \neq NP$, этот частный случай не является NP-полным.

Он не является NP-полным по той простой причине, что не может быть решен за время $O(nW)$ нашим алгоритмом динамического программирования из раздела 6.4; когда W ограничивается полиномиальной функцией n, это алгоритм с полиномиальным временем.

Все это вполне понятно; так зачем останавливаться на этом? Дело в том, что существует целая категория задач, которой часто приписывают NP-полноту (даже в опубликованных статьях) посредством сведения от частного случая суммы подмножеств. Ниже приведен типичный пример такой задачи, который мы назовем *задачей группировки компонентов*.

Для заданного графа G, не являющегося связным, и числа k существует ли подмножество связных компонентов, размер объединения которых равен в точности k?

Неправильное утверждение. Задача о группировке компонентов является NP-полной.

Неправильное доказательство. Задача о группировке компонентов принадлежит NP, доказательство не приводится. Теперь мы попытаемся показать, что *Суммирование подмножеств* $\leq_P$ *Группировка компонентов*. Для заданного экземпляра задачи о суммировании подмножеств с числами w_1, ..., w_n и целевого значения W экземпляр задачи группировки компонентов строится следующим образом: для каждого i строится путь P_i длины w_i. Граф G представляет собой объединение путей P_1, ..., P_n, каждый из которых является отдельным связным компонентом. Мы присваиваем $k = W$. Очевидно, что G содержит множество связных компонентов, объединение которых имеет размер k в том, и только в том случае, если некоторое подмножество чисел w_1, ..., w_n в сумме дает W. ■

Ошибка в этом «доказательстве» весьма коварна; в частности, утверждение в последнем предложении истинно. Проблема в том, что построение, описанное выше, не устанавливает, что *Суммирование подмножеств* $\leq_P$ *Группировка компонентов*, потому что эта задача требует более чем полиномиального времени.

При построении входных данных для «черного ящика», решающего задачу группировки компонентов, нам пришлось построить систему кодирования графа с размером $w_1 + ... + w_n$, а это требует времени, экспоненциального по размеру входных данных экземпляра задачи о суммировании подмножеств. По сути, задача о суммировании подмножеств работает с числами $w_1, ..., w_n$ в чрезвычайно компактном представлении, но задача группировки компонентов не получает «компактную» кодировку графа.

Проблема более фундаментальна, чем неправильность этого доказательства; на самом деле задача группировки компонентов может быть решена за полиномиальное время. Если $n_1, n_2, ..., n_c$ — размеры связных компонентов G, мы просто используем наш алгоритм динамического программирования для задачи о суммировании подмножеств, чтобы принять решение о том, дает ли некоторое подмножество этих чисел $\{n_i\}$ в сумме k. Необходимое для этого время выполнения равно $O(ck)$; и поскольку и c, и k ограничиваются n, получается время $O(n^2)$.

Таким образом, мы обнаружили новый алгоритм с полиномиальным временем посредством сведения в обратном направлении — к особому случаю задачи о суммировании подмножеств, решаемому за полиномиальное время.

8.9. Co-*NP* и асимметрия *NP*

Чтобы расширить свое представление об этом общем классе задач, вернемся к тем определениям, на которых базировался класс *NP*. Мы уже видели, что понятие эффективного сертифицирующего алгоритма не предполагает конкретного алгоритма для фактического решения задачи, который бы превосходил метод «грубой силы».

А теперь еще одно наблюдение: определение эффективного сертифицирования (а следовательно, и *NP*) по своей сути асимметрично. Входная строка s представляет экземпляр с положительной сертификацией в том, и только в том случае, если существует короткое значение t, для которого $B(s, t) = да$. Из отрицания этого утверждения мы видим, что входная строка представляет экземпляр с отрицательной сертификацией в том, и только в том случае, если для всех коротких t выполняется $B(s, t) = нет$.

Все это близко к нашим интуитивным представлениям о *NP*: если экземпляр сертифицируется положительно, мы можем привести короткое доказательство этого факта. Но для экземпляров с отрицательной сертификацией определение не гарантирует соответствующего короткого доказательства; ответ отрицателен просто потому, что не удается найти строку, которая бы служила доказательством. Вспомните вопрос из раздела 8.3: если набор условий невыполним, какие «доказательства» могли бы быстро убедить вас в невыполнимости задачи?

Для каждой задачи X существует естественная *дополняющая* задача $\overline{X}$: для всех входных строк s мы говорим, что $s \in \overline{X}$ в том, и только в том случае, если $s \notin \overline{X}$. Следует заметить, что если $X \in P$, то $\overline{X} \in P$, так как из алгоритма A, решающего X,

мы можем просто построить алгоритм $\overline{A}$, который выполняет A, а затем «инвертирует» его ответ.

Но при этом совершенно не очевидно, что из $X \in NP$ должно следовать, что $\overline{X} \in NP$. Вместо этого задача $\overline{X}$ обладает другим свойством: для всех s выполняется $s \in \overline{X}$ в том, и только в том случае, если для всех t длины не более $p(|s|)$, $B(s, t) = нет$. Это принципиально иное определение, которое невозможно обойти простым «инвертированием» вывода эффективного сертифицирующего алгоритма B для получения $\overline{B}$. Проблема в том, что «существует t» в определении NP превращается в «для всех t», и это серьезное изменение.

Существует класс задач, параллельных NP, спроектированных для моделирования этой проблемы; ему присвоено достаточно естественное название *co-NP*. Задача X принадлежит *co-NP* в том, и только в том случае, если дополняющая задача $\overline{X}$ принадлежит NP. Но мы не знаем наверняка, что NP и *co-NP* различны; можем только спросить:

(8.25) Выполняется ли $NP = co\text{-}NP$?

И снова принято считать, что $NP \neq co\text{-}NP$: из того, что экземпляры задач с положительной сертификацией имеют короткие доказательства, совершенно не следует, что экземпляры с отрицательной сертификацией также должны иметь короткие доказательства.

Доказательство $NP \neq co\text{-}NP$ стало бы еще более важным шагом, чем доказательство $P \neq NP$, по следующей причине:

(8.26) Если $NP \neq co\text{-}NP$, то $P \neq NP$.

Доказательство. На самом деле мы докажем обратное утверждение: из $P = NP$ следует $NP = co\text{-}NP$. Здесь важно то, что множество P замкнуто относительно дополнения; следовательно, если $P = NP$, то множество NP также будет замкнуто относительно дополнения. Или в более формальном изложении, начиная с предположения $P = NP$, имеем

$$X \in NP \Rightarrow X \in P \Rightarrow \overline{X} \in P \Rightarrow \overline{X} \in NP \Rightarrow X \in \text{co-}NP$$

и

$$X \in \text{co-}NP \Rightarrow \overline{X} \in NP \Rightarrow \overline{X} \in P \Rightarrow X \in P \Rightarrow X \in NP.$$

Из этого следует, что $NP \subseteq \text{co-}NP$ и $\text{co-}NP \subseteq NP$, а значит, $NP = co\text{-}NP$. ■

Хорошая характеризация: класс $NP \cap co\text{-}NP$

Если задача X принадлежит как NP, так и *co-NP*, то она обладает следующим полезным свойством: для ответа «да» существует короткое доказательство; для ответа «нет» также существует короткое доказательство. Задачи, принадлежащие этому пересечению $NP \cap co\text{-}NP$, называются имеющими *хорошую характеризацию*, так как для решения всегда существует хороший сертификат.

Это понятие напрямую соответствует некоторым результатам, которые мы видели ранее. Например, возьмем задачу определения того, содержит ли потоковая сеть поток с величиной как минимум v, для некоторого значения v. Чтобы доказать, что ответ на этот вопрос положительный, достаточно продемонстрировать поток, достигающий этой величины; это свойство согласуется с принадлежностью задачи *NP*. Но также можно доказать и отрицательный ответ: можно продемонстрировать разрез, пропускная способность которого строго меньше v. Этот дуализм между экземплярами с положительной и отрицательной сертификацией является сутью теоремы о максимальном потоке и минимальном разрезе.

Аналогичным образом теорема Холла для сочетаний из раздела 7.5 доказывала, что задача о двудольном идеальном паросочетании принадлежит $NP \cap co\text{-}NP$: мы могли предъявить либо идеальное паросочетание, либо множество вершин $A \subseteq X$, для которого общее количество соседей A строго меньше $|A|$.

Если задача X находится в P, то она принадлежит как *NP*, так и *co-NP*; следовательно, $P \subseteq NP \cap co\text{-}NP$. Интересно, что как наше доказательство теоремы о максимальном потоке и минимальном разрезе, так и наше доказательство теоремы Холла идут рука об руку с доказательствами более сильных результатов о том, что задачи о максимальном потоке и двудольном паросочетании являются задачами из P. Тем не менее хорошие характеризации сами по себе настолько элегантны, что их отдельная формулировка дает значительную концептуальную основу для рассуждения об этих задачах.

Естественно, нам хотелось бы знать, существует ли задача с хорошей характеризацией, например без алгоритма с полиномиальным временем. Однако этот вопрос также остается открытым:

(8.27) Выполняется ли $P = NP \cap co\text{-}NP$?

В отличие от вопросов (8.11) и (8.25), общественное мнение по этому поводу неоднородно. Отчасти это объясняется существованием многих случаев, в которых у задач отыскивались нетривиальные хорошие характеризации; а затем (порой через много лет) выяснялось, что задача имеет алгоритм с полиномиальным временем.

8.10. Частичная классификация сложных задач

Мы подошли к концу главы, в которой была предоставлена довольно обширная подборка *NP*-полных задач. В некотором отношении полезно знать побольше разных *NP*-полных задач: когда вы обнаруживаете новую задачу X и хотите доказать ее *NP*-полноту, нужно продемонстрировать $Y \leq_P X$ для некоторой известной *NP*-полной задачи Y — и чем шире выбор для Y, тем лучше.

В то же время чем больше вариантов найдется для Y, тем труднее может быть выбрать правильный вариант для конкретного сведения. Конечно, вся суть *NP*-полноты заключается в том, что одна из этих задач работает в сведении в том,

и только в том случае, если работают все (так как они эквивалентны в отношении сведения с полиномиальным временем), но сведение к конкретной задаче X от некоторых задач может быть намного, намного проще, чем от других.

С учетом этого обстоятельства мы приведем в завершающем разделе краткую сводку *NP*-полных задач, упоминавшихся в этой главе, разделив их на шесть основных категорий. Вместе с классификацией будут предложены рекомендации относительно выбора исходной задачи для использования в сведении.

Задачи упаковки

Задачи упаковки обычно имеют следующую структуру: дан набор объектов, требуется выбрать не менее k из них. Задача усложняется потенциальными конфликтами между объектами, которые не позволяют выбрать некоторые группы одновременно.

В этой главе были представлены две основные задачи упаковки.

- Задача о независимом множестве: для заданного графа G и числа k содержит ли граф G независимое множество с размером не менее k?
- Задача упаковки множества: для заданного множества U из n элементов, набора $S_1, ..., S_m$ подмножеств U и числа k существует ли набор из минимум k таких подмножеств, из которых никакие два не пересекаются?

Задачи покрытия

Задачи покрытия образуют естественный контраст с задачами упаковки. Как правило, для них характерна следующая структура: дан набор объектов, из которого выбирается подмножество, в совокупности достигающее некоторой цели. Требуется достичь этой цели с выбором только k объектов.

В этой главе были представлены две основные задачи покрытия.

- Задача о вершинном покрытии: для заданного графа G и числа k содержит ли G вершинное покрытие с размером не более k?
- Задача покрытия множества: для заданного множества U из n элементов, набора $S_1, ..., S_m$ подмножеств U и числа k существует ли набор из не более чем k таких множеств, объединение которых равно всему множеству U?

Задачи разбиения

Задачи разбиения подразумевают нахождение всех способов деления совокупности объектов на подмножества, при которых каждый объект входит ровно в одно подмножество.

Одна из основных задач разбиения — задача о трехмерном сочетании — естественным образом возникает тогда, когда имеется набор множеств и вы хотите решить задачу покрытия одновременно с задачей упаковки: выбрать часть множеств

так, чтобы они не пересекались, но при этом полностью покрывали универсальное множество.

- Задача о трехмерном сочетании: для заданных непересекающихся множеств X, Y и Z, каждое из которых имеет размер n, и заданного множества $T \subseteq X \times Y \times Z$ упорядоченных триплетов, существует ли в T такое множество из n триплетов, что каждый элемент $X \cup Y \cup Z$ содержится ровно в одном из этих триплетов?

Другая базовая задача разбиения — задача раскраски графа — встречается тогда, когда вы пытаетесь организовать разбиение объектов при наличии конфликтов (конфликтующие объекты не могут входить в одно множество).

- Задача о раскраске графа: для заданного графа G и границы k имеет ли граф G k-раскраску?

Задачи упорядочения

Первые три типа задач связаны с поиском по подмножествам набора объектов. В другой категории вычислительно сложных задач задействован поиск по множеству всех перестановок совокупности объектов.

Сложность двух базовых задач упорядочения определяется тем фактом, что задача требует упорядочения n объектов, но существуют ограничения, из-за которых одни объекты не могут размещаться после некоторых других.

- Задача о гамильтоновом цикле: содержит ли заданный направленный граф G гамильтонов цикл?
- Задача о гамильтоновом пути: содержит ли заданный направленный граф G гамильтонов путь?

Третья базовая задача упорядочения очень похожа на эти; в ней эти ограничения ослабляются введением стоимостей за размещение одного объекта после других.

- Задача о коммивояжере: для заданного набора расстояний между n городами и границы D существует ли маршрут, длина которого не превышает D?

Численные задачи

Сложность численных задач, упомянутых в этой главе, в основном происходит от задачи о суммировании подмножеств — особого случая задачи о рюкзаке, рассмотренной в разделе 8.8.

- Задача о суммировании подмножеств: задано множество натуральных чисел $w_1, ..., w_n$ и целевое число W. Существует ли подмножество $\{w_1, ..., w_n\}$, сумма которого равна ровно W?

Естественно попытаться применить сведение из задачи о суммировании подмножеств, когда имеется задача со взвешенными объектами, а целью является выбор объектов с учетом некоторого ограничения на общий вес выбранных объектов. Например, именно это происходит в доказательстве (8.24) при демонстрации *NP*-полноты планирования с временем доступности и предельным временем.

В то же время следует учитывать, что задача о суммировании подмножеств становится сложной только с действительно большими целыми числами; когда величины входных чисел ограничиваются полиномиальной функцией n, задача решается за полиномиальное время средствами динамического программирования.

Задачи соблюдения ограничений

Наконец, в этой главе рассматривались основные задачи соблюдения ограничений, включая задачу выполнимости булевой схемы, SAT и 3-SAT. Среди них для планирования сведений более полезной была задача 3-SAT.

> 3-SAT: для заданного множества условий $C_1, ..., C_k$, каждое из которых имеет длину 3, по множеству переменных $X = \{x_1, ..., x_n\}$, существует ли выполняющее логическое присваивание?

Из-за своей выразительности и гибкости задача 3-SAT часто становится полезной отправной точкой для сведений, в которых ни одна из предшествующих пяти категорий не находит естественного соответствия с рассматриваемой задачей. При проектировании сверток 3-SAT будет полезно вспомнить совет, приведенный в доказательстве (8.8) о том, что экземпляр 3-SAT может рассматриваться с двух разных точек зрения: (a) как поиск по вариантам присваивания переменным с учетом ограничения, что все условия должны быть выполнены, и (b) как поиск по вариантам выбора одного литерала из каждого условия с учетом ограничения о невозможности выбора конфликтующих литералов из разных условий. Каждое из этих представлений 3-SAT полезно, и каждое формирует ключевую идею для целого класса сведений.

Упражнения с решениями

Упражнение с решением 1

Вы консультируете небольшую компанию, которая управляет компьютерной системой с повышенным уровнем безопасности для выполняемой засекреченной работы. Чтобы система не использовалась в незаконных целях, на ней работает служебная программа для сохранения IP-адресов, по которым обращаются пользователи. Предполагается, что каждый пользователь в любую минуту обращается не более чем по одному IP-адресу; программа записывает в файл для каждого пользователя u и каждой минуты m значение $I(u, m)$, равное IP-адресу, к которому пользователь u обращался в минуту m. Если пользователь u в минуту m не обращался ни по какому IP-адресу, то $I(u, m)$ присваивается нуль-символ $\perp$.

Руководство компании только что узнало, что вчера система была использована для проведения сложной атаки на удаленные сайты. Атака проводилась с обращением к t разным IP-адресам за t последовательных минут; в минуту 1 атакующий обращался по адресу i_1; в минуту 2 он обращался по адресу i_2; и т. д. вплоть до адреса i_t в минуту t.

Кто же несет ответственность за атаку? Руководство компании проверяет файлы журналов и, к своему удивлению, выясняет, что нет ни одного пользователя u, который бы обращался по всем атакованным IP-адресам в соответствующее время; другими словами, нет такого u, что $I(u, m) = i_m$ для каждой минуты m от 1 до t.

Неужели атака была совместно проведена небольшой группой из k пользователей? Подмножество S пользователей будет называться *подозрительной группой*, если в каждую минуту m от 1 до t был по крайней мере один пользователь $u \in S$, для которого $I(u, m) = i_m$. (Другими словами, к каждому IP-адресу в соответствующий момент обращался по крайней мере один пользователь из группы.)

Итак, для заданного набора всех значений $I(u, m)$ и числа k существует ли подозрительная группа с размером не более k?

Решение

Прежде всего, задача о подозрительной группе очевидно принадлежит *NP*: если нам предъявят множество S пользователей, то мы можем проверить, что размер S не превышает k и в каждую минуту m от 1 до t по крайней мере один из пользователей S обращался к IP-адресу i_m.

Мы хотим найти заведомо *NP*-полную задачу и свести ее к задаче о подозрительной группе. Хотя эта задача имеет несколько характеристик (пользователи, минуты, IP-адреса), совершенно очевидно, что она является задачей покрытия (в соответствии с классификацией, описанной в этой главе): требуется найти объяснение для всех t подозрительных обращений, с ограничением количества пользователей (k). Когда мы решаем, что это задача покрытия, будет естественно попытаться свести к ней задачу о вершинном покрытии или покрытии множества. А для этого полезно вытеснить большинство усложняющих аспектов, оставив минимум, необходимый для кодирования задачи о вершинном покрытии или покрытии множества.

Сосредоточимся на сведении задачи о вершинном покрытии. В этой задаче необходимо найти покрытие для каждого ребра, при этом разрешено использовать всего k узлов. В задаче о подозрительной группе требуется «покрыть» все обращения и при этом разрешено использовать только k пользователей. Этот параллелизм дает веские основания полагать, что для заданного экземпляра задачи о вершинном покрытии, состоящего из графа $G = (V, E)$ и границы k, следует построить экземпляр задачи о подозрительной группе, в которой пользователи соответствуют узлам G, а подозрительные обращения — ребрам.

Итак, предположим, что граф G в экземпляре задачи о вершинном покрытии состоит из m ребер e_1, ..., e_m, и $e_j = (v_j, w_j)$. Экземпляр задачи о подозрительной группе строится следующим образом: для каждого узла G создается пользователь, а для каждого ребра $e_t = (v_t, w_t)$ создается минута t. (Так что всего будет m минут.) В минуту t пользователи, связанные с двумя концами e_t, обращаются к IP-адресу i_t, а все остальные пользователи не обращаются ни к чему. Наконец, атака состоит из обращений по адресам $i_1, i_2, ..., i_m$ в минуты 1, 2, ..., m соответственно.

Следующее утверждение устанавливает, что *Вершинное покрытие* $\leq_P$ *Подозрительная группа*, а следовательно, завершает доказательство *NP*-полноты задачи о по-

дозрительной группе. Учитывая, насколько близко наш построенный экземпляр воспроизводит исходный экземпляр задачи о вершинном покрытии, доказательство полностью прямолинейно.

(8.28) В построенном экземпляре задачи подозрительная группа с размером не более k существует в том, и только в том случае, если граф G содержит вершинное покрытие с размером не более k.

Доказательство. Сначала предположим, что G содержит вершинное покрытие C с размером не более k. Теперь рассмотрим соответствующее множество S пользователей в экземпляре задачи о подозрительной группе. Для всех t от 1 до m по крайней мере один элемент C является концом ребра e_t, и соответствующий пользователь в S обращался к IP-адресу i_t. Следовательно, множество S является подозрительной группой.

И наоборот, предположим, что существует подозрительная группа S с размером не более k, и рассмотрим соответствующее множество узлов C в G. Для всех t от 1 до m по крайней мере один пользователь из S обращался к IP-адресу i_t, и соответствующий узел в C является концом ребра e_t. Отсюда следует, что множество S является вершинным покрытием. ■

Упражнение с решением 2

Вам предложено организовать семинар начального уровня, который будет проводиться раз в неделю на протяжении следующего семестра. Предполагается, что первая часть семестра состоит из серии из ℓ лекций, а вторая часть семестра будет посвящена серии из p практических проектов, которые будут выполняться студентами.

Всего имеется n возможных лекторов, и в неделю с номером i (для $i = 1, 2, ..., \ell$) доступно подмножество L_i лекторов для проведения лекций.

С другой стороны, каждый проект требует наличия определенной подготовки, чтобы студенты могли успешно завершить его. В частности, для каждого проекта j (для $j = 1, 2, ..., p$) определено подмножество P_j лекторов; чтобы завершить проект, студент должен посетить лекцию хотя бы одного из лекторов подмножества P_j.

Итак, задача: для заданных множеств можно ли выбрать ровно одного лектора для каждой из первых ℓ недель семинара, чтобы выбирались только лекторы, свободные в соответствующую неделю, а для каждого проекта j студенты прослушали как минимум одного из лекторов соответствующего множества P_j? Назовем ее *задачей планирования лекций*.

Чтобы прояснить ситуацию, рассмотрим следующий пример. Допустим, $\ell = 2$, $p = 3$ и доступны $n = 4$ лектора, обозначенные A, B, C, D. Доступность лекторов определяется множествами $L_1 = \{A, B, C\}$ и $L_2 = \{A, D\}$. Необходимые докладчики для каждого проекта определяются множествами $P_1 = \{B,C\}$, $P_2 = \{A, B, D\}$ и $P_3 = \{C, D\}$. В этом случае ответ для экземпляра задачи будет положительным: в первую неделю выбирается лектор B, а во вторую лектор D; для каждого из трех проектов студенты посетят лекцию хотя бы одного из необходимых лекторов.

Докажите, что задача планирования лекций является *NP*-полной.

Доказательство. Задача принадлежит *NP*, так как для заданной последовательности лекторов мы можем проверить, что (a) все лекторы свободны в назначенные им недели и (b) что для каждого проекта был выбран хотя бы один из необходимых лекторов.

Теперь нужно найти заведомо *NP*-полную задачу, которая может быть сведена к задаче о планировании лекций. Выбор не столь очевиден, как в прошлом упражнении, потому что формулировка задачи о планировании лекций не имеет прямого соответствия в классификации из этой главы.

Однако существует менее очевидное представление задачи планирования лекций, типичное для широкого диапазона задач на соблюдение ограничений. Это представление было весьма живописно отражено в «Нью-Йоркере» при описании стиля перекрестного допроса адвоката Дэвида Бойеса: во время перекрестного допроса Дэвид словно неспешно прогуливается со свидетелем по коридору, одновременно незаметно закрывая двери. Дойдя до конца коридора, Дэвид поворачивается к свидетелю — и выясняется, что отступать уже некуда. Все двери закрыты[1].

Какое отношение соблюдение ограничений имеет к перекрестному допросу? Задача планирования лекций, как и многие похожие задачи, состоит из двух концептуальных фаз. В первой фазе вы перебираете набор вариантов, выбирая одни и «закрывая двери» для других; затем наступает вторая фаза, где можно проверить, приводят выбранные варианты к действительному решению или нет.

В случае планирования лекций первая фаза состоит из выбора лектора на каждую неделю, а вторая — из проверки того, что для каждого проекта был выбран необходимый лектор. Однако существует много *NP*-полных задач, которые соответствуют этому описанию на верхнем уровне, поэтому рассмотрение задачи с этой точки зрения упростит поиск возможного сведения. Мы опишем два сведения: от 3-SAT и от задачи о вершинном покрытии. Конечно, любое из них само по себе достаточно для доказательства *NP*-полноты, но оба примера будут полезны.

Задача 3-SAT — канонический пример задачи с двухфазной структурой, описанной выше: сначала мы перебираем переменные, присваивая каждой *true* или *false*; затем перебираем все условия и смотрим, выполняются ли они с выбранными значениями. Эта параллель с задачей планирования лекций уже предполагает естественное сведение, демонстрирующее, что 3-*SAT* $\leq_P$ *Планирование лекций*: мы определяем задачу так, чтобы выбор лекторов задавал переменные, после чего возможность реализации проектов представляет выполнение условий.

Допустим, имеется экземпляр задачи 3-SAT, состоящий из условий $C_1, ..., C_k$ по переменным $x_1, x_2, ..., x_n$. Построим экземпляр задачи планирования лекций следующим образом: для каждой переменной x_i создаются два лектора z_i и z'_i, соответствующие переменной x_i и ее отрицанию. Сначала идут n недель лекций; в неделю i доступны только лекторы z_i и z'_i. Далее следует серия из k проектов; для проекта j множество необходимых лекторов P_j состоит из трех лекторов, соответствующих литералам в условии C_j. Теперь, если для экземпляра 3-SAT существует выполняющее присваивание v, то в неделю i мы выбираем лектора между z_i и z'_i, соответствующего значению, присвоенному x_i в результате v; в этом случае будет выбран

[1] «The New Yorker», 16 августа 1999 года.

как минимум один лектор из каждого необходимого множества P_j. И наоборот, если можно найти вариант выбора лекторов, включающий как минимум одного лектора из каждого необходимого множества, мы сможем задать переменные x_i следующим образом: x_i присваивается 1, если выбран z_i, или присваивается 0, если выбран z'_i. В этом случае по крайней мере одна из трех переменных в каждом условии C_j получит значение, с которым это условие выполнено, поэтому данное присваивание является выполняющим. На этом завершается как сведение, так и доказательство его правильности.

Наше интуитивное представление задачи планирования лекций естественным образом приводит и к сведению от задачи о вершинном покрытии. (Приведенное описание легко изменяется для задачи покрытия множеств или трехмерного сочетания.) Суть в том, что задача о вершинном покрытии также может рассматриваться как имеющая аналогичную двухфазную структуру: сначала мы выбираем множество из k узлов во входном графе, а затем для каждого ребра проверяем, что выбранные варианты покрывают все ребра.

Для заданных входных данных для задачи о вершинном покрытии, состоящих из графа $G = (V, E)$ и числа k, создается лектор z_v для каждого узла v. Мы присваиваем $\ell = k$ и определяем $L_1 = L_2 = ... = L_k = \{z_v: v \in V\}$. Другими словами, для первых k недель доступны все лекторы. После этого мы создаем проект j для каждого ребра $e_j = (v, w)$, с множеством $P_j = \{z_v, z_w\}$.

Далее, если существует вершинное покрытие S, состоящее не более чем из k узлов, рассмотрим множество лекторов $Z_S = \{z_v: v \in S\}$. Для каждого проекта P_j по крайней мере один из необходимых лекторов принадлежит Z_S, так как S покрывает все ребра в G. Кроме того, мы можем планировать всех лекторов в Z_S на протяжении первых k недель. Из этого следует, что мы имеем действительное решение экземпляра задачи о планировании лекций.

И наоборот, предположим, что существует действительное решение экземпляра задачи о планировании лекций; пусть T — множество всех лекторов, выступающих в первые k недель. Обозначим X множество узлов G, соответствующих лекторам в T. Для каждого проекта P_j по крайней мере один из двух необходимых лекторов присутствует в T, а следовательно, по крайней мере один конец каждого ребра e_j входит в множество X. Таким образом, X — вершинное покрытие, содержащее не более k узлов.

Это завершает доказательство того, что *Вершинное покрытие* $\leq_P$ *Планирование лекций*.

Упражнения

1. Для каждого из следующих двух вопросов выберите вариант ответа: (i) «Да», (ii) «Нет» или (iii) «Неизвестно, потому что это привело бы к ответу на вопрос о $P = NP$». Приведите краткое объяснение своего ответа.

   (a) Определим версию задачи интервального планирования из главы 4 следующим образом: для заданного набора интервалов и границы k существует ли подмножество неперекрывающихся интервалов с размером не менее k?

Вопрос: Можно ли утверждать, что *Интервальное планирование* $\leq_P$ *Вершинное покрытие*?

(b) Вопрос: Можно ли утверждать, что
Независимое множество $\leq_P$ *Интервальное планирование*?

2. Для анализа поведения покупателей в магазинах часто ведется двумерный массив A, строки которого соответствуют клиентам, а столбцы — продаваемым товарам. Элемент $A[i, j]$ определяет количество единиц товара j, приобретенных покупателем i. Ниже приведен небольшой пример такого массива.

	Моющее средство	**Пиво**	**Пеленки**	**Наполнитель для кошачьего туалета**
Радж	0	6	0	3
Аланис	2	3	0	0
Челси	0	0	0	7

Один из способов обработки полученных данных выглядит так: допустим, подмножество S покупателей называется *разнородным*, если никакие два покупателя из S никогда не покупали один и тот же товар (то есть каждый товар был куплен не более чем одним покупателем из S). Например, разнородное множество покупателей может пригодиться для проведения маркетингового исследования.

Теперь мы можем определить задачу о разнородном множестве следующим образом: для заданного массива A размером $m \times n$, определенного выше, и числа $k \leq m$ существует ли разнородное подмножество, содержащее не менее k покупателей?

Покажите, что задача о разнородном множестве является *NP*-полной.

3. При организации летнего спортивного лагеря возникает следующая задача. В лагере должен быть как минимум один квалифицированный тренер для каждого из n видов спорта (баскетбол, волейбол и т. д.). Организаторы получают заявки от m потенциальных тренеров. Для каждого из n видов спорта существует подмножество из m претендентов, квалифицированных в этом в виде спорта. Вопрос заключается в следующем: для заданного числа $k < m$ возможно ли нанять не более k тренеров, чтобы для каждого из n видов спорта нашелся хотя бы один квалифицированный тренер? Назовем эту задачу *задачей эффективного найма*.

Покажите, что задача эффективного найма является *NP*-полной.

4. Предположим, вы занимаетесь администрированием высокопроизводительной системы реального времени, в которой асинхронные процессы работают с общими ресурсами. В системе существует множество из n процессов и множество из m ресурсов. В любой заданный момент времени каждый процесс определяет набор ресурсов, которые он намеревается использовать. Каждый ресурс может запрашиваться сразу несколькими процессами, но использоваться в любой момент времени он может только одним процессом. Ваша задача — распределить ресурсы между запрашивающими их процессами. Если процесс получает все

запрошенные им ресурсы, он остается активным; в противном случае его выполнение блокируется. Соответственно *задача резервирования ресурсов* формулируется следующим образом: для заданных множество процессов и ресурсов, множества запрашиваемых ресурсов для каждого процесса и числа k возможно ли распределить ресурсы между процессами так, чтобы не менее k процессов оставались активными?

Рассмотрите задачи из следующего списка и для каждой задачи либо приведите алгоритм с полиномиальным временем, либо докажите, что задача является *NP*-полной.

(a) Общая задача резервирования ресурсов, определенная выше.

(b) Частный случай задачи при $k = 2$.

(c) Частный случай задачи с двумя типами ресурсов (допустим, люди и оборудование); каждый процесс запрашивает максимум один ресурс каждого типа (другими словами, каждый процесс запрашивает одного конкретного человека и одно конкретное устройство).

(d) Частный случай задачи, в котором каждый ресурс запрашивается максимум двумя процессами.

5. Рассмотрим множество $A = \{a_1, ..., a_n\}$ и набор $B_1, B_2, ..., B_m$ подмножеств A (то есть $B_i \subseteq A$ для всех i).

Множество $H \subseteq A$ называется *множеством представителей* для набора $B_1, B_2, ..., B_m$, если H содержит минимум один элемент из каждого B_i — то есть если $H \cap B_i$ не пусто для каждого i (таким образом, H содержит «представителей» из всех множеств B_i).

Задача множества представителей определяется следующим образом: имеется множество $A = \{a_1, ..., a_n\}$, набор $B_1, B_2, ..., B_m$ подмножеств A и число k. Существует ли множество представителей $H \subseteq A$ для $B_1, B_2, ..., B_m$, размер которого не превышает k?

Докажите, что задача множества представителей является *NP*-полной.

6. Рассмотрим экземпляр задачи выполнимости, заданный условиями $C_1, ..., C_k$ по множеству булевых переменных $x_1, ..., x_n$. Экземпляр называется *монотонным*, если литерал в каждом условии состоит из неинвертированной переменной, то есть каждый литерал равен x_i для некоторого i, а не $\overline{x_i}$. Монотонные экземпляры задачи выполнимости решаются очень легко: они всегда выполнимы присваиванием каждой переменной 1.

Предположим, имеются три условия

$$(x_1 \vee x_2), (x_1 \vee x_3), (x_2 \vee x_3).$$

Экземпляр является монотонным, и присваивание, задающее все три переменные равными 1, действительно выполняет все условия. Но это не единственное выполняющее присваивание; также можно было присвоить x_1 и x_2 значение 1, а x_3 значение 0. В самом деле, для любого монотонного экземпляра естественно задать вопрос о минимальном количестве переменных, которым необходимо присвоить 1 для его выполнения.

Задача монотонной выполнимости с минимумом истинных переменных формулируется так: для заданного монотонного экземпляра задачи выполнимости и числа k существует ли выполняющее присваивание для экземпляра, в котором не более k переменных задано значение 1? Докажите, что задача является *NP*-полной.

7. Так как задача о трехмерном сочетании является *NP*-полной, естественно ожидать, что аналогичная задача о четырехмерном сочетании будет хотя бы не менее сложной. Определим *четырехмерное сочетание* следующим образом: для заданных множеств W, X, Y и Z, каждое из которых имеет размер n, и набора C упорядоченных четверок в форме (w_i, x_j, y_k, z_l) существуют ли n четверок из C, среди которых никакие два не имеют общих элементов?

   Докажите, что задача о четырехмерном сочетании является *NP*-полной.

8. Малолетняя дочь ваших знакомых Мэдисон учится читать. Чтобы помочь ей, родители купили набор цветных магнитов на холодильник с буквами алфавита (несколько экземпляров буквы A, несколько экземпляров буквы B и т. д.). При последней встрече вы строили из магнитов слова, которые она уже знает.

   Изначальные условия игры немного изменились. Вскоре вы пытались строить слова так, чтобы использовать все магниты из набора — то есть выбирали слова, которые она уже знает, но так, чтобы после выкладывания всех слов каждый магнит участвовал в записи ровно одного слова. (Разрешается повторение слов; например, если в набор магнитов входят по два экземпляра букв C, A и T, слово CAT можно построить дважды.)

   Задача оказалась довольно сложной, и только позднее вы поняли причину. Допустим, мы рассматриваем обобщенную версию *задачи использования всех магнитов*, в которой английский алфавит заменяется произвольным набором символов, а словарь Мэдисон моделируется произвольным набором строк из символов этого набора. Цель остается той же, что в предыдущем абзаце.

   Докажите, что задача использования всех магнитов является *NP*-полной.

9. Вы управляете сетью передачи данных, моделируемой направленным графом $G = (V, E)$. Имеется c пользователей, которые намерены использовать эту сеть. Пользователь i (для всех $i = 1, 2, ..., c$) выдает *запрос* на резервирование пути P_i в графе G, по которому должны передаваться данные.

   Вы заинтересованы в том, чтобы принять как можно больше таких запросов, со следующим ограничением: если приняты запросы на пути P_i и P_j, то P_i и P_j не могут содержать общих узлов.

   Итак, задача о выборе путей формулируется следующим образом: для направленного графа $G = (V, E)$, множества запросов $P_1, P_2, ..., P_c$ (каждый из которых определяет путь в G) и числа k возможно ли выбрать как минимум k путей, чтобы никакие два из выбранных путей не содержали общих узлов?

   Докажите, что задача о выборе путей является *NP*-полной.

10. Ваши друзья из фирмы WebExodus недавно консультировали компании с большими общедоступными сайтами (условия контракта не позволяют сказать, какими именно), и в ходе работы они столкнулись со следующей *задачей стратегического размещения рекламы*.

Компания предоставляет карту сайта, которая моделируется направленным графом $G = (V, E)$. Компания также предоставляет множество t типичных маршрутов перемещения по сайту; маршруты моделируются направленными путями $P_1, P_2, ..., P_t$ в графе G (то есть каждый маршрут P_i представляется путем в G).

Компания хочет получить от WebExodus ответ на следующий вопрос: для заданного графа G, путей $\{P_i\}$ и числа k возможно ли разместить рекламу не более чем на k узлах G, чтобы каждый путь P_i включал как минимум один узел с рекламой? Назовем эту задачу «задачей стратегического размещения рекламы» со входными данными G, $\{P_i: i = 1, ..., t\}$ и k.

Ваши друзья считают, что хороший алгоритм решения задачи их обогатит; к сожалению, все не так просто.

(a) Докажите, что задача стратегического размещения рекламы является *NP*-полной.

(b) Ваши друзья из WebExodus написали довольно быстрый алгоритм S, который выдает ответы «да/нет» для произвольного экземпляра задачи стратегического размещения рекламы. Будем считать, что алгоритм S всегда работает правильно.

Используя алгоритм S как «черный ящик», разработайте алгоритм, который получает входные данные G, $\{P_i\}$ и k из части (a) и делает одно из двух:

- выводит множество, содержащее не более k узлов из G, для которого каждый путь P_i содержит как минимум один такой узел, или
- выводит (обоснованно) сообщение о том, что такое множество из максимум k узлов не существует.

Ваш алгоритм должен использовать не более полиномиального количества вычислительных шагов, в сочетании с не более чем полиномиальным количеством вызовов алгоритма S.

11. Как еще помнят некоторые (а другие слышали), идея гипертекста появилась за десятилетия до Всемирной паутины. Даже гипертекстовая литература появилась относительно давно: вместо линейной печатной страницы автор пишет историю, состоящую из набора связанных виртуальных мест, соединенных виртуальными «проходами»[1]. Таким образом, гипертекстовое произведение в действительности строится на основе направленного графа; для конкретности (хотя и с сужением полного диапазона возможностей) мы воспользуемся следующей моделью.

Будем рассматривать структуру гипертекстового произведения как направленный граф $G = (V, E)$. Каждый узел $u \in V$ содержит некоторый текст; когда читатель достигает u, он может выбрать любое ребро, ведущее из u; и если было выбрано ребро $e = (u, v)$, читатель переходит к узлу v. Чтение начинается с узла $s \in V$ и заканчивается на узле $t \in V$; когда читатель впервые достигает t, история завершается. Следовательно, любой путь от s к t является действительным сюжетом произведения. Учтите, что в отличие от просмотра страниц

[1] Например, см. http://www.eastgate.com.

в браузере возможность возврата может отсутствовать; после перехода от u к v вы уже можете никогда не вернуться к u.

В такой модели гипертекстовая структура определяет огромное количество разных сюжетов с одним базовым контентом; и отношения между многочисленными вариантами порой становятся весьма нетривиальными. Пример задачи, встречающейся при таком подходе к структуре: рассмотрим фрагмент гипертекстового произведения на базе графа $G = (V, E)$, в котором содержатся некоторые ключевые тематические элементы: любовь, смерть, война, стремление получить ученую степень по информатике и т. д. Каждый тематический элемент i представляется множеством $T_i \subseteq V$, состоящим из всех узлов G, в которых встречается данная тема. Теперь для заданного множества тематических элементов можно задать вопрос: существует ли действительное развитие сюжета, в котором встречается каждый из таких элементов? Конкретнее, для направленного графа G с начальным узлом s и конечным узлом t и тематическими элементами, представленными множествами $T_1, T_2, ..., T_k$, *задача воплощения сюжета* формулируется так: существует ли путь из s в t, содержащий по крайней мере один узел каждого из множеств T_i?

Докажите, что задача воплощения сюжета является *NP*-полной.

12. Ваши друзья открыли популярный сайт с новостями и форумом. Трафик достиг уровня, при котором они хотят различать платных и бесплатных посетителей. Обычно в такой схеме весь контент сайта доступен для посетителей, оплачивающих ежемесячную подписку; посетители, которые не платят за пользование сайтом, могут просматривать некоторое подмножество страниц (и надоедливую рекламу, которая предлагает им перейти на платную подписку).

    Существуют два простых способа управления доступом для бесплатных посетителей: (1) фиксированное подмножество страниц помечается как разрешенное для просмотра бесплатными посетителями или (2) просматриваться могут все страницы, но ограничивается максимальное количество страниц, которые могут просматриваться бесплатным посетителем за один сеанс. (Будем считать, что сайт может отслеживать действия посетителя на сайте.)

    Ваши друзья экспериментируют с моделью ограничения доступа — более хитрой по сравнению с обоими вариантами. Они хотят, чтобы бесплатные посетители могли получить представление о разных разделах сайта, для чего подмножества страниц обозначаются как образующие некую *зону*: например, на сайте одна зона страниц может быть посвящена политике, другая — музыке и т. д. Страница может принадлежать сразу нескольким зонам. Когда бесплатный пользователь перемещается по сайту, политика доступа позволяет ему посетить одну страницу из каждой зоны, но попытка обратиться ко второй странице из той же зоны пресекается. (Вместо этого пользователь направляется на страницу с предложением оформить подписку.)

    Формально сайт может моделироваться направленным графом $G = (V, E)$, в котором ребра представляют веб-страницы, а ребра — направленные гиперссылки. Существует *входной узел* $s \in V$ и *зоны* $Z_1, ..., Z_k \subseteq V$. Путь P, выбираемый бесплатным пользователем, ограничивается включением одного узла из каждой зоны Z.

Один из недостатков усложненной политики доступа заключается в том, что с ней трудно ответить даже на самые простейшие вопросы относительно достижимости узлов, например: может ли бесплатный пользователь посетить заданный узел *t*? Более точная формулировка *задачи проблемного пути* выглядит так: для заданного графа $G, Z_1, ..., Z_k, s \in V$ и конечного узла $t \in V$ существует ли в G путь s–t, включающий не более одного узла из каждой зоны Z_i? Докажите, что задача проблемного пути является *NP*-полной.

13. Механизм *комбинаторного аукциона* был разработан экономистами для продажи множества объектов множеству потенциальных покупателей. (Федеральное агентство по связи США изучало эту разновидность аукционов для распределения радиочастот среди вещательных компаний.)

    Простой пример комбинаторного аукциона: имеются n продаваемых объектов с метками $I_1, ..., I_n$. Каждый объект является неделимым и может быть продан только одному покупателю. Далее, m разных покупателей делают ставки: i-я ставка задает подмножество S_i объектов, а x_i — цена, которую покупатель готов заплатить за объекты множества S_i как за единое целое. (Ставка будет представляться парой (S_i, x_i).)

    Теперь аукционист просматривает множество всех m заявок; он выбирает одни заявки и отклоняет другие. Каждый покупатель, ставка которого i будет принята, покупает все объекты соответствующего множества S_i. Следовательно, две принятые ставки не могут содержать множества, содержащие хотя бы один общий элемент, так как один объект не может быть продан двум разным людям.

    Аукционист получает сумму предложенных цен по всем принятым ставкам. (Обратите внимание: у каждого покупателя всего одна попытка, сделать новую ставку уже не получится.) Цель аукциониста — получить как можно больше денег.

    Задача определения победителя для комбинаторного аукциона формулируется так: для заданных объектов $I_1, ..., I_n$, ставок $(S_1, x_1), ..., (S_m, x_m)$ и границы B существует ли набор ставок, которые может принять аукционист, чтобы заработанная сумма была не меньше B?

    Пример. Предположим, аукционист решает воспользоваться описанным методом для продажи лишнего компьютерного оборудования. Продаются четыре объекта: «PC», «монитор», «принтер» и «сканер»; трое покупателей делают ставки. Определяем:

    $$S_1 = \{PC, монитор\}, S_2 = \{PC, принтер\}, S_3 = \{монитор, принтер, сканер\}$$

    и

    $$x_1 = x_2 = x_3 = 1$$

    Сделаны ставки (S_1, x_1), (S_2, x_2) и (S_3, x_3), и граница B равна 2.

    В данном экземпляре ответ будет отрицательным: аукционист сможет принять не более одной ставки (так как у любых двух ставок имеются общие объекты), поэтому общая выручка не может превысить 1.

Докажите, что задача определения победителя для комбинаторного аукциона является *NP*-полной.

14. Задача интервального планирования упоминалась в главах 1 и 4. Рассмотрим намного более сложную с вычислительной точки зрения ее разновидность, которую мы назовем *задачей множественного интервального планирования*. Как и прежде, имеется процессор, способный выполнять задания в течение некоторого периода времени (например, с 9:00 до 17:00).

    Пользователи поставляют задания, которые должны выполняться на процессоре; в любой момент времени на процессоре может выполняться только одно задание. Однако в этой модели задания имеют более сложную структуру: каждое задание определяет множество интервалов, в течение которых ему требуется использовать процессоры. Итак, например, одно задание может занимать процессор с 10:00 до 11:00, а потом с 14:00 до 15:00. Если принять это задание, оно занимает процессор в течение двух часов, но вы можете принимать задания на другие периоды (включая промежуток с 11:00 до 14:00).

    Имеется множество из n заданий, каждое из которых определяется множеством интервалов. Требуется найти ответ на следующий вопрос: для заданного числа k возможно ли принять не менее k заданий так, чтобы никакие два задания не перекрывались по времени?

    Покажите, что задача множественного интервального планирования является *NP*-полной.

15. Однажды во время работы вам приносят пакет FedEx. В пакете лежит сотовый телефон, который начинает звонить, — оказывается, это ваш друг Нео, от которого уже давно не было никаких известий. Все разговоры с Нео всегда проходят одинаково: сначала он долго и пафосно объясняет, почему он обратился к вам, но в итоге все сводится к просьбам потратить ваше время на то, чтобы помочь ему с решением некоторой проблемы.

    На этот раз по причинам, о которых он вынужден умолчать (что-то типа защиты подземного города от роботов-убийц), ему и его знакомым требуется отслеживать радиосигналы в разных точках электромагнитного спектра. Всего есть n разных частот, для наблюдения за которыми существует массив датчиков.

    Задача отслеживания состоит из двух компонентов:

    - множество L из m мест, в которых могут быть размещены датчики;
    - множество S из b источников помех, каждый из которых блокирует некоторые частоты в некоторых местах. А конкретно каждый источник помех i задается парой (F_i, L_i), где F_i — подмножество частот, а L_i — подмножество мест; это означает, что датчик, помещенный в любое из мест множества L_i, не сможет получать сигналы на любых частотах множества F_i (из-за воздействия помех).

    Подмножество $L' \subseteq L$ мест называется *достаточным*, если для каждой из n частот j существует некоторое место в L', в котором частота j не блокируется ни одним источником помех. Тогда, размещая датчик в каждом месте достаточного множества, вы сможете успешно отслеживать все n частот.

У соратников Нео есть k датчиков, и они хотят узнать, существует ли достаточное множество мест с размером не более k. Назовем следующую формулировку *задачей отслеживания частот*: для заданных частот, мест, источников помех и параметра k существует ли достаточное множество с размером не более k?

Пример. Допустим, заданы четыре частоты $\{f_1, f_2, f_3, f_4\}$ и четыре места $\{\ell_1, \ell_2, \ell_3, \ell_4\}$. Существуют три источника помех:

$$(F_1, L_1) = (\{f_1, f_2\}, \{\ell_1, \ell_2, \ell_3\})$$
$$(F_2, L_2) = (\{f_3, f_4\}, \{\ell_3, \ell_4\})$$
$$(F_3, L_3) = (\{f_2, f_3\}, \{\ell_1\})$$

В этой ситуации существует достаточное множество с размером 2: можно выбрать места ℓ_2 и ℓ_4. (Так как f_1 и f_2 не блокируются в ℓ_4, а f_3 и f_4 не блокируются в ℓ_2).

Докажите, что задача отслеживания частот является *NP*-полной.

16. Рассмотрим задачу характеристики множества по размерам его пересечений с другими множествами. Имеется конечное множество U размера n, а также набор $A_1, ..., A_m$ подмножеств U. Также заданы числа $c_1, ..., c_m$. Вопрос звучит так: существует ли такое множество $X \subset U$, что для всех $i = 1, 2, ..., m$ мощность $X \cap A_i$ равна c_i? Назовем его *задачей выведения пересечений* с входными данными U, $\{A_i\}$ и $\{c_i\}$.

    Докажите, что задача выведения пересечений является *NP*-полной.

17. Задан направленный граф $G = (V, E)$ с весами w_e ребер $e \in E$. Веса могут быть отрицательными или положительными. В задаче *цикла с нулевым весом* требуется решить, существует ли в G простой цикл, сумма весов ребер которого равна 0. Докажите, что эта задача является *NP*-полной.

18. Вас попросили помочь теоретикам в анализе данных, связанных с принятием решений. В частности, аналитики рассматривают набор данных по решениям, принятых некоторым правительственным комитетом, и пытаются выявить малое подмножество влиятельных членов комитета.

    Состав комитета представлен множеством $M = \{m_1, ..., m_n\}$ из n членов. За последний год в комитете проводилось голосование по t разным вопросам. По каждому вопросу каждый член комитета мог проголосовать «За», «Против» или «Воздержаться». В результате комитет принимает положительное решение по каждому вопросу, если количество голосов «За» строго превышает количество голосов «Против» (воздержавшиеся не считаются ни за одну из сторон); в противном случае принимается отрицательное решение.

    Имеется большая таблица, содержащая данные голосования каждого участника комитета по каждому вопросу. Рассматривается следующее определение: подмножество членов $M' \subseteq M$ называется *решающим*, если при проверке только голосов, поданных членами M', решение комитета по каждому вопросу будет тем же. (Иначе говоря, результат голосования членов M' по каждому вопросу

определяет общий результат голосования всего комитета.) Такое подмножество можно рассматривать как своего рода «ближнее окружение», которое отражает мнение комитета в целом.

Вопрос: для данных о голосовании каждым членом комитета по каждому вопросу и заданного параметра k требуется узнать, существует ли решающее подмножество, содержащее не более k членов. Назовем эту формулировку *задачей решающего подмножества*.

Пример. Допустим, имеются четыре члена комитета и три вопроса. Мы ищем решающее подмножество с размером не более $k = 2$, а голосование проходило так:

Вопрос	m_1	m_2	m_3	m_4
Вопрос 1	За	За	Воздержаться	Против
Вопрос 2	Воздержаться	Против	Против	Воздержаться
Вопрос 3	За	Воздержаться	За	За

В данном экземпляре результат будет положительным, поскольку члены m_1 и m_3 образуют решающее подмножество.

Докажите, что задача решающего подмножества является *NP*-полной.

19. Предположим, вы консультируете управление порта в маленькой стране на побережье Тихого океана. Ежегодный объем сделок исчисляется миллиардами, и доход ограничивается исключительно скоростью разгрузки кораблей, прибывающих в порт.

Работа с опасными грузами усложняет и без того непростую задачу. Допустим, утром в порт прибывает морской конвой, который привозит n контейнеров с разными видами опасных материалов. В доке стоят m машин, каждая из которых вмещает до k контейнеров.

Существуют две взаимосвязанные задачи, которые обусловлены разными типами ограничений, действующих при обработке опасных материалов. Для каждой из двух задач выберите один из двух ответов:

- Алгоритм с полиномиальным временем для решения задачи.
- Доказательство *NP*-полноты задачи.

(a) Для каждого контейнера существует заданное подмножество машин, на которых они могут перевозиться. Возможно ли погрузить все n контейнеров на m машин, чтобы ни одна машина не была перегружена, а каждый контейнер был погружен на машину, для которой разрешена его перевозка?

(b) В другой версии задачи любой контейнер может быть погружен на любую машину, однако некоторые пары контейнеров не могут быть погружены на одну машину. (При контакте содержащихся в них химикатов может произойти взрыв.) Возможно ли погрузить все n контейнеров на m машин так, чтобы ни одна машина не была перегружена и никакие два контейнера не грузились на одну машину, если это запрещено правилами?

20. Существует много разных способов формального представления задачи кластеризации, целью которой является разбиение набора объектов на «похожие» группы.

    Естественным способом представления входных данных для задачи кластеризации является множество объектов $p_1, p_2, ..., p_n$, в котором для каждой пары определяется числовое расстояние $d(p_i, p_j)$. (Для расстояний устанавливаются требования: $d(p_i, p_i) = 0$; $d(p_i, p_j) > 0$ для разных p_i и p_j и симметричность $d(p_i, p_j) = d(p_j, p_i)$.)

    Ранее в разделе 4.7 была рассмотрена одна из возможных формулировок задачи кластеризации: разбить объекты на k множеств так, чтобы максимизировать минимальное расстояние между парами объектов из разных кластеров. Как выясняется, задача решается удачным применением задачи о минимальном остовном дереве.

    Другой, но на первый взгляд похожий способ формализации может выглядеть следующим образом: разбить объекты на k множеств так, чтобы минимизировать максимальное расстояние между любой парой объектов в пределах одного кластера. Обратите внимание на отличия: если в формулировке из предыдущего абзаца нужно было найти кластеры, среди которых не было бы двух «слишком близких», в новой формулировке кластеры не должны быть слишком «широкими», то есть ни один кластер не должен содержать двух точек, находящихся на большом расстоянии друг от друга.

    Тот факт, что новая формулировка обладает слишком большой вычислительной сложностью для оптимального решения, может показаться удивительным (при таком сходстве задач). Чтобы задачу можно было проанализировать в контексте *NP*-полноты, сначала запишем ее в формулировке с принятием решения. Для заданных n объектов $p_1, p_2, ..., p_n$ с расстояниями, определенными выше, и границы B *задача кластеризации малого диаметра* формулируется следующим образом: возможно ли разбить объекты на k множеств так, чтобы расстояние между любыми двумя точками, принадлежащими одному множеству, не превышало B?

    Докажите, что задача кластеризации малого диаметра является *NP*-полной.

21. После знакомства с популярной книгой для начинающих предпринимателей вы осознаете, что компьютерная система в вашем офисе нуждается в обновлении. Однако это начинание сталкивается с некоторыми непростыми проблемами.

    При планировании новой системы необходимо выбрать k компонентов: операционная система, редактор текстов, почтовый клиент и т. д. Для j-го компонента системы существует множество A_j вариантов; а конфигурация системы состоит из выбора одного элемента из каждого из множеств $A_1, A_2, ..., A_k$.

    Проблемы возникают из-за того, что некоторые пары вариантов из разных множеств оказываются несовместимыми. Вариант $x_i \in A_i$ и вариант $x_j \in A_j$ образуют несовместимую пару, если они не могут содержаться в одной системе: например, Linux (вариант операционной системы) и Microsoft Word (вариант редактора текстов) образуют несовместимую пару. Конфигурация системы называется *полностью совместимой*, если она состоит из элементов $x_1 \in A_1$, $x_2 \in A_2, ... x_k \in A_k$, из которых ни одна пара (x_i, x_j) не является несовместимой.

Теперь мы можем определить *задачу полностью совместимой конфигурации*. Экземпляр такой задачи состоит из непересекающихся множеств вариантов $A_1, A_2, ..., A_k$, и множества P несовместимых пар (x,y), в котором x и y являются элементами разных множеств вариантов. Требуется решить, существует ли полностью совместимая конфигурация: выбор элемента из каждого множества вариантов, при котором ни одна пара выбранных элементов не принадлежит множеству P.

Пример. Допустим, $k = 3$, а множества A_1, A_2, A_3 обозначают варианты выбора операционной системы, редактора текстов и почтового клиента соответственно:

$A_1 = \{$`Linux`, `Windows NT`$\}$,

$A_2 = \{$`emacs`, `Word`$\}$,

$A_3 = \{$`Outlook`, `Eudora`, `rmail`$\}$.

Множество несовместимых пар выглядит так:

$P = \{$(`Linux`, `Word`), (`Linux`, `Outlook`), (`Word`, `rmail`)$\}$

В этом экземпляре ответ на версию с принятием решения будет положительным: например, выбор `Linux` $\in A_1$, `emacs` $\in A_2$, `rmail` $\in A_3$ образует полностью совместимую конфигурацию в соответствии с предшествующими определениями.

Докажите, что задача полностью совместимой конфигурации является *NP*-полной.

22. Предположим, вы получили алгоритм A — «черный ящик», который получает ненаправленный граф $G = (V, E)$ и число k и ведет себя следующим образом:
    - Если граф G не является связным, алгоритм просто возвращает сообщение о несвязности.
    - Если граф G является связным и в нем есть независимое множество с размером не менее k, алгоритм возвращает сообщение «Да».
    - Если граф G является связным, но в нем нет независимого множества с размером не менее k, алгоритм возвращает сообщение «Нет».

    Предположим, алгоритм A выполняется за время, полиномиальное по размеру G и k.

    Покажите, как с использованием обращений к A решить за полиномиальное время задачу о независимом множестве: для заданного графа G и числа k содержит ли граф G независимое множество с размером не менее k?

23. Для множества конечных двоичных строк $S = \{s_1, ..., s_k\}$ строка u называется *конкатенацией* по S, если она равна $s_{i_1} s_{i_2} \cdots s_{i_t}$ для некоторых индексов $i_1, ..., i_t \in \{1, ..., k\}$.

    Ваш знакомый рассматривает следующую задачу: для заданных двух множеств конечных двоичных строк $A = \{a_1, ..., a_m\}$ и $B = \{b_1, ..., b_n\}$ существует ли строка u, которая бы одновременно являлась конкатенацией по A и конкатенацией по B?

    Ваш знакомый заявляет: «По крайней мере задача принадлежит *NP*, поскольку для доказательства положительного ответа мне достаточно предъявить такую строку u». Вы указываете (разумеется, вежливо), что такое объяснение совершенно несостоятельно: как узнать, что кратчайшая из таких строк u не имеет

длины, экспоненциальной по размеру входных данных, — ведь в этом случае она не может служить сертификатом полиномиального размера?

Но как выясняется, это заявление можно преобразовать в доказательство принадлежности *NP*. А именно, докажите следующее утверждение:

Если существует строка u, которая является конкатенацией и по A, и по B, то существует строка, длина которой ограничивается полиномиально по сумме длин строк в $A \cup B$.

24. Имеется двудольный граф $G = (V, E)$; допустим, его узлы разбиты на множества X и Y так, что один конец каждого ребра принадлежит X, а другой принадлежит Y. Определим (a, b)-*скелет* графа G как множество ребер $E' \subseteq E$, для которого *не более a* узлов в X инцидентно ребру в E' и *не менее b* узлов в Y инцидентно ребру в E'.

    Покажите, что для заданного двудольного графа G и чисел a и b задача принятия решения о существовании (a, b)-скелета является *NP*-полной.

25. Для функций $g_1, ..., g_l$ функция $\max(g_1, ..., g_l)$ определяется по формуле

$$[\max(g_1, ..., g_l)](x) = \max(g_1(x), ..., g_l(x)).$$

    Рассмотрим следующую задачу: имеется n кусочно-линейных непрерывных функций $f_1, ..., f_n$, определенных на интервале $[0, t]$ для некоторого целого t. Также задано целое число B. Требуется решить: существуют ли k функций $f_{i_1}, ..., f_{i_k}$, для которых

$$\int_0^t \left[\max(f_{i_1}, ..., f_{i_k})\right](x)dx \geq B\,?$$

    Докажите, что эта задача является *NP*-полной.

26. Разъезжая по отдаленным частям света, вы со своим другом накопили большую груду сувениров. В то время вы не задумывались над тем, какие сувениры останутся вам, а какие — ему, но, похоже, пришло время разделить приобретения.

    Один из возможных способов дележа выглядит так: допустим, имеется n объектов $1, 2, ..., n$ и объект i имеет согласованную ценность x_i. (Например, это может быть сумма, за которую его можно продать; будем считать, что у вас с друзьями нет расхождений по поводу оценки). Можно попытаться найти разбиение объектов на два множества, чтобы общая ценность объектов в каждом множестве была одинаковой.

    Для этого следует решить следующую *задачу численного разбиения*. Даны положительные целые числа $x_1, ..., x_n$; требуется решить, возможно ли разбить числа на два множества S_1 и S_2 с одинаковой суммой:

$$\sum_{x_i \in S_1} x_i = \sum_{x_j \in S_2} x_j .$$

    Покажите, что задача численного разбиения является *NP*-полной.

27. Рассмотрим следующую задачу. Даны положительные целые числа $x_1, ..., x_n$, а также числа k и B. Требуется узнать, возможно ли разбить числа $\{x_i\}$ на k множеств $S_1, ..., S_k$ так, чтобы квадраты сумм множеств в совокупности давали не более B:

$$\sum_{i=1}^{k}\left(\sum_{x_j \in S_i} x_j\right)^2 \le B.$$

Покажите, что эта задача является *NP*-полной.

28. Ниже описана разновидность задачи о независимом множестве. Имеется граф $G = (V, E)$ и целое число k. В контексте этой задачи множество $I \subset V$ будет называться *строго независимым*, если для любых двух узлов $v, u \in I$ ребро (v, u) не принадлежит E, а также не существует пути из двух ребер из u в v — то есть не существует узла w, для которого $(u, w) \in E$ и $(w, v) \in E$. *Задача о строго независимом множестве* требует решить, существует ли в G строго независимое множество с размером не менее k.

Докажите, что задача о строго независимом множестве является *NP*-полной.

29. Вы строите большую сеть рабочих станций, которая будет моделироваться ненаправленным графом G; узлы G представляют отдельные рабочие станции, а ребра представляют прямые каналы связи. Всем рабочим станциям необходим доступ к центральной базе данных с информацией, необходимой для работы базовых функций операционной системы.

Конечно, базу данных можно реплицировать на каждой рабочей станции; в этом случае обращения с любой станции будут выполняться очень быстро, но вам придется поддерживать большое количество копий. Также можно сохранить одну копию базы данных на одной рабочей станции, к которой другие рабочие станции будут обращаться за информацией по сети, но это создаст большие задержки для рабочих станций, находящихся на расстоянии многих сетевых переходов от места хранения базы данных.

Похоже, стоит поискать компромисс: ограничиться небольшим количеством копий базы данных, но разместить их так, чтобы любая рабочая станция либо содержала собственную копию, либо была соединена прямым каналом с рабочей станцией, на которой имеется копия. В терминологии графов такое множество узлов называется *доминирующим*.

Задача о доминирующем множестве формулируется следующим образом: для заданной сети G и числа k возможно ли разместить k копий базы данных в k разных узлах так, чтобы каждый узел либо содержал собственную копию базы данных либо был соединен прямым каналом с узлом, содержащим копию?

Покажите, что задача о доминирующем множестве является *NP*-полной.

30. Порой даже в традиционных задачах «непрерывной» математики совершенно неожиданно проявляются дискретные, комбинаторные аспекты из изучаемой нами области, которые проникают даже во вполне стандартные на первый взгляд задачи.

Для примера рассмотрим традиционную задачу минимизации функции одной переменной — допустим, $f(x) = 3 + x - 3x^2$ на интервале $x \in [0,1]$. Производная рав-

на нулю в точке $x = 1/6$, но это точка максимума, а не минимума функции; чтобы получить минимум, следует выполнить стандартную проверку значений на границах интервала (причем минимум в данном случае достигается на границе $x = 1$).

Проверка границы не создает серьезных проблем для функций одной переменной; но предположим, что в задаче требуется минимизировать функцию n переменных $x_1, x_2, ..., x_n$ на единичном кубе, где все $x_1, x_2, ..., x_n \in [0,1]$. Минимум может достигаться как внутри куба, так и на его границах; и последняя перспектива выглядит устрашающе, так как куб состоит из 2^n «углов» (для которых каждое из значений x_i равно либо 0, либо 1), а также из различных фрагментов других измерений. Из-за этой сложности книги по матанализу подозрительно нечетко излагают этот вопрос при описании многофакторных задач оптимизации.

И тому есть веская причина: минимизация функции с n переменными в единичном n-мерном кубе относится к классу *NP*-полных задач. Чтобы это утверждение стало более конкретным, рассмотрим специальный случай многочленов с целочисленными коэффициентами по n переменным $x_1, x_2, ..., x_n$. На всякий случай освежим в памяти терминологию: *одночленом* называется произведение вещественного коэффициента c и каждой из переменных x_i, возведенной в неотрицательную целую степень a_i; это можно записать в виде $cx_1^{a_1}cx_2^{a_2}\cdots cx_n^{a_n}$. (Например, $2x_1^2x_2x_3^4$ является одночленом). *Многочленом* называется сумма конечного набора одночленов (например, $2x_1^2x_2x_3^4 + x_1x_3 - 6x_2^2x_3^2$ является многочленом).

Задача минимизации многофакторного многочлена формулируется так: для заданного многочлена по n переменным с целочисленными коэффициентами и заданной целой границей B возможно ли выбрать вещественные числа $x_1, x_2, ..., x_n \in [0, 1]$, с которыми многочлен достигает значения $\leq B$?

Выберите в этой главе задачу Y, которая заведомо является *NP*-полной, и покажите, что $Y \leq_P$ *Минимизация многофакторного многочлена*.

31. Для заданного ненаправленного графа $G = (V, E)$ *множеством обратной связи* называется множество $X \subseteq V$, обладающее тем свойством, что G-X не содержит циклов. Задача ненаправленного множества обратной связи формулируется так: для заданных G и k содержит ли G множество обратной связи с размером не более k? Докажите, что задача ненаправленного множества обратной связи является *NP*-полной.

32. Расшифровка генома сопряжена с множеством сложных вычислительных задач. На простейшем уровне каждая хромосома организма может рассматриваться как чрезвычайно длинная строка (возможно, состоящая из миллионов символов) четырехбуквенного алфавита $\{A, C, G, T\}$. Один из классов методов расшифровки генома базируется на генерировании большого количества коротких, перекрывающихся фрагментов хромосомы, с последующим построением полной длинной строки, представляющей хромосому, из этого набора перекрывающихся подстрок.

Хотя мы не будем рассматривать задачи сборки строк во всех подробностях, ниже приведена упрощенная задача, которая дает некоторое представление

о вычислительной сложности задач этой области. Допустим, имеется множество $S = \{s_1, s_2, ..., s_n\}$ коротких строк ДНК из q-буквенного алфавита; каждая строка s_i имеет длину 2ℓ для некоторого числа $\ell \geq 1$. Также имеется библиотека дополнительных строк $T = \{t_1, t_2, ..., t_m\}$ из символов того же алфавита; каждая из этих строк также имеет длину 2ℓ. Пытаясь определить, может ли строка s_b следовать прямо за строкой s_a в хромосоме, мы проверяем, содержит ли библиотека T строку t_k, чтобы первые ℓ символов t_k совпадали с последними ℓ символами в s_a, а последние ℓ символов t_k совпадали с первыми ℓ символами в s_b. Если это возможно, мы говорим, что t_k *скрепляет* пару (s_a, s_b). (Другими словами, t_k может быть фрагментом ДНК, «перекрывающим» точку перехода s_a в s_b.)

Нам хотелось бы сцепить все строки S в некотором порядке без перекрытий, так чтобы каждая последовательная пара скреплялась некоторой строкой из библиотеки T. Следовательно, мы хотим упорядочить строки S в виде $s_{i_1}, s_{i_2}, ..., s_{i_n}$, где $i_1, i_2, ..., i_n$ — такая перестановка $\{1, 2, ..., n\}$, что для всех $j = 1, 2, ..., n - 1$ существует строка t_k, скрепляющая пару $(s_{i_j}, s_{i_{j+1}})$. (Одна строка t_k может использоваться для нескольких последовательных пар в конкатенации.) Если это возможно, говорят, что у множества S существует *идеальная сборка*.

Для заданных множеств S и T задача об идеальной сборке формулируется так: существует ли для S идеальная сборка по отношению к T? Докажите, что задача об идеальной сборке является *NP*-полной.

Пример. Допустим, задан алфавит $\{A,C,G,T\}$, множество $S = \{AG,TC,TA\}$ и множество $T = \{AC,CA,GC,GT\}$ (так, что каждая строка имеет длину $2\ell = 2$). Тогда ответ на этот экземпляр задачи об идеальной сборке будет положительным: три строки S можно сцепить в порядке $TCAGTA$ (так, что $s_{i_1} = s_2$, $s_{i_2} = s_1$ и $s_{i_3} = s_3$). В этом порядке пара (s_{i_1}, s_{i_2}) скрепляется строкой CA из библиотеки T, а пара (s_{i_2}, s_{i_3}) сцепляется строкой GT из библиотеки T.

33. В натуральном хозяйстве люди обмениваются товарами и услугами напрямую, без участия денег как промежуточного шага. Обмен происходит тогда, когда каждая сторона рассматривает множество получаемых товаров как обладающее большей ценностью, чем множество товаров, которые отдаются взамен. Исторически человеческое общество обычно переходило от натурального хозяйства к денежной экономике; соответственно различные сетевые системы, экспериментирующие с бартером, могут рассматриваться как намеренные попытки вернуться к более ранней форме экономических взаимодействий. При этом обнаруживаются некоторые проблемы, изначально присущие натуральному обмену по сравнению с системами, использующими деньги. Одной из таких трудностей является сложность нахождения возможностей для обмена (даже когда такие возможности существуют).

Для моделирования этих сложностей мы введем концепцию *ценности*, которая назначается каждым человеком каждому объекту в реальном мире. Предположим, имеется множество из n людей $p_1, ..., p_n$ и множество m объектов $a_1, ..., a_m$. Каждый объект принадлежит одному человеку. Далее, у каждого человека p_i имеется оценочная функция v_i, которая определяется так, что $v_i(a_j)$ является

неотрицательным числом, которое определяет ценность объекта a_j для p_i, — чем больше число, тем большую ценность объект представляет для человека.

Обмен между двумя людьми в такой системе возможен в том случае, если существуют люди p_i и p_j, и подмножества объектов A_i и A_j, принадлежащих p_i и p_j соответственно, так что для каждого участника предпочтительными являются объекты того подмножества, которое ему пока не принадлежит. А точнее, суммарная ценность объектов A_j для участника p_i превышает суммарную ценность объектов A_i, и суммарная ценность объектов A_i для участника p_j превышает суммарную ценность объектов A_j. (Обратите внимание: подмножество A_i не обязано включать все объекты, принадлежащие p_i; то же относится к A_j.) A_i и A_j могут быть произвольными подмножествами принадлежащих им объектов, удовлетворяющим этим критериям.)

Допустим, вы получили экземпляр задачи натурального обмена, определяемый приведенными выше данными ценности объектов (чтобы избежать проблем с представлением вещественных чисел, будем считать, что ценности всех объектов представляют собой натуральные числа). Докажите, что задача определения возможности обмена между двумя людьми является *NP*-полной.

34. В 1970-е годы ученые, работающие в области математической социологии (включая Марка Грановеттера и Томаса Шеллинга), взялись за разработку моделей некоторых видов коллективного человеческого поведения: почему одни модные «фишки» приживаются, а другие быстро отмирают? Почему одни технологические новшества распространяются повсеместно, а другие не выходят за пределы малых групп пользователей? Какова динамика беспорядков и грабежей, которые иногда проявляются в поведении разъяренной толпы? Ученые предположили, что все эти ситуации являются примерами *каскадных процессов*, при которых поведение участника сильно зависит от поведения его близкого окружения; процесс, запущенный несколькими участниками, распространяется на все большее количество людей и в конечном итоге получает очень широкое воздействие. Этот процесс можно сравнить с распространением слухов или эпидемии, передаваемой от человека к человеку.

Рассмотрим простейшую версию модели каскадных процессов. Имеется некоторое поведение (собирание марок, покупка сотового телефона, участие в беспорядках), и в какой-то момент времени каждый человек либо принимает в нем участие, либо не принимает. Население представляется направленным графом $G = (V, E)$, в котором узлы соответствуют людям, а ребро (v, w) означает, что человек v влияет на поведение человека w; если человек v участвует в поведении, это также способствует принятию поведения человеком w. У каждого человека w также имеется порог $\theta(w) \in [0,1]$, который имеет следующий смысл: в любой момент, когда доля узлов, от которых выходят ребра в w, достигнет по меньшей мере $\theta(w)$, узел w также начинает участвовать в поведении.

Узлы с низким порогом быстрее «заражаются» поведением, узлы с более высоким порогом более консервативны. Узел w с порогом $\theta(w) = 0$ принимает поведение немедленно, без влияния окружения. Наконец, нужно договориться по поводу узлов без входящих ребер: будем считать, что они принимают поведение, если $\theta(w) = 0$, и не принимают его при более высоком пороге.

Для заданного экземпляра этой модели распространение поведения может быть смоделировано следующим образом.

```
Инициализировать все узлы w с θ(w) = 0 как участников поведения.
   (Все остальные узлы в исходном состоянии не участвуют в поведении.)
Пока множество участников поведения продолжает изменяться:
   Для всех людей, не участвующих в поведении, одновременно:
      Если среди узлов с ребрами к w доля участников не менее θ(w)
         w становится участником
      Конец Если
   Конец
Конец
Вывести итоговое множество участников
```

Этот процесс конечен, так как потенциальных участников всего n и при каждой итерации по крайней мере один человек становится участником поведения.

В последние годы исследователи, работающие в области маркетинга и анализа данных, изучали возможность применения этой модели для анализа эффектов «сарафанного радио» в успехе новых продуктов (так называемый *вирусный маркетинг*). Под поведением понимается использование нового продукта; нужно попытаться убедить нескольких ключевых представителей населения опробовать новый продукт и надеяться, что это вызовет по возможности значительный каскадный эффект.

Допустим, выбирается множество узлов $S \subseteq V$, а порог каждого узла в S обнуляется 0. (Убедив людей опробовать новый продукт, мы гарантируем их участие.) Затем запускается описанный выше процесс, и мы смотрим, насколько большим окажется итоговое множество участников. Обозначим размер итогового множества участников $f(S)$ (размер записывается в виде функции от S, так как он естественно зависит от выбора S). Величина $f(S)$ может рассматриваться как *влияние* множества S, так как она отражает распространение поведения, «посеянного» в S.

При планировании маркетинговой кампании целью является нахождение малого множества S, имеющего как можно большее влияние $f(S)$. *Задача максимизации влияния* определяется так: для заданного направленного графа $G = (V, E)$ с пороговыми значениями для каждого узла, параметров k и b существует ли множество S, содержащее не более k узлов, для которых $f(S) \geq b$?

Докажите, что задача максимизации влияния является *NP*-полной.

Пример. Допустим, граф $G = (V, E)$ состоит из пяти узлов $\{a, b, c, d, e\}$, четырех ребер (a, b), (b, c), (e, d), (d, c), а пороги всех узлов установлены равными 2/3. В этом случае ответ на экземпляр задачи максимизации влияния, определяемой графом G, с $k = 2$ и $b = 5$, будет положительным: если выбрать $S = \{a, e\}$, то все три остальных узла также станут участниками. (В этой задаче такой вариант выбора S только один; например, если выбрать $S = \{a, d\}$, то b и c станут участниками, но e не станет; а если выбрать $S = \{a, b\}$, то ни один из узлов c, d или e не станет участником.)

35. Трое ваших друзей работают в крупной компании, занимающейся компьютерными играми. Вот уже несколько месяцев они работают над идеей новой игры «Droid Trader!», одобренной руководством компании. За это время им пришлось выслушать множество обескураживающих замечаний, от «Вам придется основательно поработать с отделом маркетинга над названием» до «А может, стоит добавить побольше крови?»

    Но сейчас практически все уверены, что игра пойдет в производство, и друзья обращаются к вам за решением последней проблемы: а не получится ли игра слишком простой — игроки слишком быстро освоят ее и им станет скучно?

    Вы не сразу разбираетесь в происходящем по их подробному описанию; но если отбросить все космические сражения, поединки и псевдомистицизм в духе «Звездных войн», основная идея выглядит так. Игрок управляет космическим кораблем и пытается заработать деньги, покупая и продавая дроидов на разных планетах. Всего существуют n типов дроидов и k планет. Каждая планета p обладает следующими свойствами: на ней продаются $s(j, p) \geq 0$ дроидов типа j по фиксированной цене $x(j, p) \geq 0$ за штуку, для всех $j = 1, 2, ..., n$; также существует спрос на $d(j, p) \geq 0$ дроидов типа j по фиксированной цене $y(j, p) \geq 0$. (Будем считать, что на планете не может быть положительного спроса одновременно с положительным предложением на дроидов одного типа; таким образом, по крайней мере одно из значений $s(j, p)$ и $d(j, p)$ равно 0.)

    Игрок начинает на планете s с z единицами денег и заканчивает на планете t; существует направленный ациклический граф G для множества планет, в котором пути s–t соответствуют действительным маршрутам игрока. (Граф G выбран ациклическим, чтобы избежать бесконечных игр.) На протяжении пути s–t P в G игрок может участвовать в сделках следующим образом. Прибывая на планету p в пути P, он может купить до $s(j, p)$ дроидов типа j по цене $x(j, p)$ за штуку (при условии, что у него хватит денег) и/или продать до $d(j, p)$ дроидов типа j по цене $y(j, p)$ за штуку (для $j = 1, 2, ..., n$). Счет игрока определяется суммой денег, которую он имеет на руках при прибытии на планету t. (Также начисляются дополнительные очки за космические сражения и поединки, но для формулировки задачи они нас не интересуют.)

    Итак, задача фактически сводится к получению наибольшего результата. Другими словами, для заданного экземпляра игры с направленным ациклическим графом G на базе множества планет, всех остальных параметров, описанных выше, и целевого ограничения B существует ли путь P в G и последовательность сделок в P, при которой игрок заканчивает со счетом не менее B? Назовем этот экземпляр *задачей максимального счета в игре Droid Trader*!, или сокращенно ЗМСDT!.

    Докажите, что задача ЗМСDT! является *NP*-полной (в предположении $P \neq NP$). Тем самым вы гарантируете отсутствие простой стратегии для получения высокого счета в игре ваших друзей.

36. Даже если вы знаете людей много лет, понять их порой бывает нелегко. Для примера возьмем хотя бы Раджа и Аланис. Ни один из них не относится к «жаворонкам», но сейчас они ежедневно встают в 6 утра и отправляются на рынок за свежими фруктами и овощами для своего ресторана здорового питания.

Стремясь сэкономить на продуктах, они столкнулись с непростой проблемой. Существует большое множество из n ингредиентов $I_1, I_2, ..., I_n$ (пучки зелени, бутылки рисового уксуса и т. д.). Ингредиент I_j покупается в единицах по $s(j)$ граммов (с покупкой целого количества единиц) и стоит $c(j)$ долларов за единицу. Кроме того, он остается годным для использования в течение $t(j)$ дней с даты покупки.

За следующие k дней ваши друзья должны подать k фирменных блюд, по одному в день. (Выбор дня, в который должно готовиться то или иное блюдо, остается на их усмотрение.) Фирменное блюдо с номером i использует подмножество $S_i \subseteq \{I_1, I_2, ..., I_n\}$ ингредиентов, а именно $a(i,j)$ граммов ингредиента I_j.

Наконец, существует еще одно ограничение: постоянные клиенты ресторана останутся постоянными только в том случае, если они получают еду, приготовленную из самых свежих продуктов. Для каждого фирменного блюда ингредиенты S_i делятся на два подмножества: те, которые должны покупаться в день изготовления фирменного блюда, и те, которые могут быть куплены в любой день, но их срок годности еще не истек. (Например, салат-месклан с базиликом должен готовиться из базилика, купленного в этот же день, а в соусе песто с рукколой, базиликом и корнелльским козьим сыром можно использовать базилик, купленный несколько дней назад, если он еще не завял).

Здесь-то и появляется возможность сэкономить на продуктах. Часто при покупке единицы ингредиента I_j для приготовления фирменного блюда на этот день им не нужен весь запас. Если они смогут на следующей день приготовить другое блюдо, которое тоже использует Ij, но не требует, чтобы ингредиент был куплен в тот же день, можно сэкономить за счет использования ранее купленных продуктов. Конечно, если блюда с базиликом будут следовать друг за другом, это может усложнить изготовление рецептов с козьим сыром и т. д., — собственно, здесь и возникает сложность.

Итак, *задача планирования фирменных блюд* выглядит так: для заданной информации об ингредиентах и рецептах, а также бюджета x возможно ли спланировать k фирменных блюд так, чтобы общая сумма, потраченная на ингредиенты за k дней, не превышала x?

Докажите, что задача планирования фирменных блюд является *NP*-полной.

37. Как ни странно, даже в «Звездных войнах» кроется немало *NP*-полных задач.

Рассмотрим задачу, с которой столкнулись Люк, Лея и их друзья на пути со «Звезды Смерти» на скрытую базу повстанцев. Галактику можно рассматривать как ненаправленный граф $G = (V, E)$, в котором узлы представляют звездные системы, а ребро (u, v) обозначает возможность прямого перемещения из u в v. «Звезде Смерти» соответствует узел s, а скрытой базе повстанцев — узел t. Расстояния, представленные ребрами графа, не одинаковы; каждому ребру e назначается целочисленная длина $\ell_e \geq 0$. Кроме того, некоторые ребра представляют маршруты, активно патрулируемые имперскими кораблями; соответственно каждому ребру e также назначается целочисленный риск $r_e \geq 0$, который определяет прогнозируемые повреждения от насыщенных спецэффектами космических сражений в случае перемещения по этому ребру.

Самый безопасный маршрут пролегает по дальним секторам Галактики, от одной безопасной звездной системы к другой; но в этом случае топливо у корабля кончится задолго до того, как он доберется до места назначения. Самый быстрый маршрут — рвануть через населенный центр, но тогда придется отбиваться от слишком большого количества имперских кораблей. В общем случае общая длина любого пути P из s в t определяется как сумма длин всех его ребер, а общий риск — как сумма рисков всех его ребер.

Итак, Люк, Лея и компания пытаются решить для этого графа усложненную разновидность задачи о кратчайшем пути: им нужно добраться из s в t по пути с разумно малой общей длиной и общим риском. А конкретнее *задачу о кратчайшем галактическом пути* можно сформулировать так: для заданной конфигурации, описанной выше, и целочисленных ограничений L и R существует ли путь из s в t, в котором общая длина не превышает L и при этом общий риск не превышает R?

Докажите, что задача о кратчайшем галактическом пути является *NP*-полной.

38. Рассмотрим разновидность задачи о дереве Штейнера, которую мы будем называть *графическим деревом Штейнера*. Имеется ненаправленный граф $G = (V, E)$, множество вершин $X \subseteq V$ и число k. Требуется решить, существует ли множество $F \subseteq E$, содержащее не более k ребер, для которого в графе (V,F) X принадлежит одной компоненте связности.

Покажите, что задача о графическом дереве Штейнера является *NP*-полной.

39. *Задача о направленных непересекающихся путях* определяется следующим образом: имеется направленный граф G и k пар узлов (s_1, t_1), (s_2, t_2), ..., (s_k, t_k). Требуется решить, существуют ли непересекающиеся по узлам пути $P_1, P_2, ..., P_k$, такие, что P_i идет из s_i в t_i.

Покажите, что задача о направленных непересекающихся путях является *NP*-полной.

40. Рассмотрим задачу, встречающуюся при проектировании схем широковещательной рассылки в сети. Имеется направленный граф $G = (V, E)$ с узлом $r \in V$ и назначенным множеством «целевых узлов» $T \subseteq V - \{r\}$. С каждым узлом v связано *время переключения* s_v, которое представляет собой положительное целое число.

Во время 0 узел r генерирует сообщение, которое должно быть получено каждым узлом в T. Для этого нужно построить схему, в которой r передает информацию некоторым из своих соседей (последовательно), которые в свою очередь сообщают некоторым из своих соседей и т. д., пока сообщение не будет получено каждым узлом T. На более формальном уровне схема рассылки определяется следующим образом: узел r может отправить копию сообщения одному из своих соседей во время 0; сосед получит сообщение во время 1. В общем случае во время $t \geq 0$ любой узел v, уже получивший сообщение, может отправить копию сообщения одному из своих соседей — при условии, что он еще не отправлял копию сообщения в любой из моментов $t - s_v + 1$, $t - s_v + 2$, ..., $t - 1$. (В этом проявляется роль времени переключения: v необходима пауза в $s_v - 1$ шагов между последовательными отправками сообщения. Если $s_v = 1$, никакие дополнительные ограничения не добавляются.)

Временем завершения схемы рассылки называется минимальное время t, к которому все узлы T получат сообщение. Задача о времени рассылки определяется так: для входных данных, описанных выше, и границы b существует ли схема рассылки с временем завершения, не превышающим b?

Докажите, что задача о времени рассылки является *NP*-полной.

Пример. Допустим, имеется направленный граф $G = (V, E)$ с множеством $V = \{r, a, b, c\}$ и ребрами (r, a), (a, b), (r, c); множество $T = \{b, c\}$ и время переключения составляет $s_v = 2$ для всех $v \in V$. В этом случае схема рассылки с минимальным временем завершения будет выглядеть так: r отправляет сообщение a во время 0; a отправляет сообщение b во время 1; r отправляет сообщение c во время 2; схема завершается во время 3, когда c получает сообщение. (Обратите внимание: узел a может отправить сообщение при получении его во время 1, так как это его первая отправка сообщения; но узел r не может отправить сообщение во время 1, так как $s_r = 2$, а сообщение было отправлено во время 0.)

41. Для направленного графа G *циклическим покрытием* называется множество циклов, не пересекающихся по узлам, в котором каждый узел в G принадлежит циклу. В *задаче о циклическом покрытии* требуется узнать, имеет ли заданный направленный граф циклическое покрытие.

    (a) Покажите, что задача о циклическом покрытии может быть решена за полиномиальное время. (Подсказка: воспользуйтесь задачей о двудольном паросочетании.)

    (b) Предположим, каждый цикл должен содержать не менее трех ребер. Покажите, что проверка существования такого циклического покрытия для графа G является *NP*-полной.

42. Компания, занимающаяся проектированием коммуникационных сетей, обращается к вам со следующей задачей. Изучается конкретная сеть, состоящая из n узлов и моделируемая направленным графом $G = (V,E)$. По соображениям отказоустойчивости G требуется разбить на максимально возможное количество виртуальных «доменов»: *доменом* в G называется множество X узлов с размером не менее 2, так что для каждой пары узлов $u, v \in X$ существуют направленные пути из u в v и из v в u, полностью содержащиеся в X.

    Покажите, что следующая *задача доменной декомпозиции* является *NP*-полной. Имеется направленный граф $G = (V, E)$ и число k; возможно ли разбить V не менее чем на k множеств, каждое из которых является доменом?

Примечания и дополнительная литература

В примечаниях к главе 2 упоминаются некоторые ранние работы по формализации вычислительной эффективности при полиномиальном времени; концепция *NP*-полноты сформировалась на основе этих работ и заняла центральное место в вычислительной теории после работ Кука (Cook, 1971), Левина (Levin, 1973) и Карпа (Karp, 1972). Эдмондс (Edmonds, 1965) обратил особое внимание на класс задач

NP ∩ *co-NP* — то есть задач, обладающих «хорошей характеризацией». В его статье также содержится явно сформулированная гипотеза о том, что задача коммивояжера не решается за полиномиальное время, что можно считать предварительной формулировкой гипотезы $P \neq NP$. У Сипсера (Sipser, 1992) приводится полезное описание всего исторического контекста.

Книга Гэри и Джонсона (Garey, Johnson, 1979) предоставляет обширный материал по *NP*-полноте и завершается очень полезным каталогом известных *NP*-полных задач. Хотя в этом каталоге по понятным причинам присутствуют только задачи, известные на момент публикации книги, он остается очень полезным справочником для новых задач, которые могут оказаться *NP*-полными. В настоящее время пространство известных *NP*-полных задач продолжает стремительно расширяться; как сказал Кристос Пападимитриу в своей лекции, «ежегодно публикуется около 6000 статей, результатом которых является вывод о *NP*-полноте. Это означает, что после обеда была обнаружена новая *NP*-полная задача» (лекция проводилась в 14:00).

NP-полноту можно интерпретировать как утверждение о том, что в каждой отдельной *NP*-полной задаче кроется полная сложность *NP*. У этого факта имеется конкретное отражение: некоторым *NP*-полным задачам, рассматриваемым здесь, были посвящены целые книги; задача коммивояжера рассматривается у Лоулера и др. (Lawler, 1985); задача о раскраске графа является темой книги Дженсена и Тофта (Jensen, Toft, 1995); задаче о рюкзаке посвящена книга Мартелло и Тота (Martello, Toth, 1990). *NP*-полнота задач планирования обсуждается в обзоре Лоулера и др. (Lawler, 1993).

Примечания к упражнениям

Некоторые упражнения демонстрируют другие хрестоматийные задачи, встречавшиеся в ходе изучения *NP*-полноты; в частности, это упражнения 5, 26, 29, 31, 38, 39, 40 и 41. Упражнение 33 основано на обсуждении с Дэниелом Головиным, а в упражнении 34 используются результаты нашей работы с Дэвидом Кемпе. Упражнение 37 является примером задач о кратчайшем пути с двумя критериями; это конкретное применение было предложено Мэвериком Ву.

Глава 9
PSPACE: класс задач за пределами *NP*

До сих пор в этой книге одним из основных вычислительных ресурсов считалось время выполнения. Именно эта концепция стала основой для принятия полиномиального времени как рабочего определения эффективности; и неявно именно она определяет различия между *P* и *NP*. В некоторой степени также упоминались требования к пространству (то есть затратам памяти) алгоритмов. В этой главе рассматривается класс задач, который определяет пространство как фундаментальный вычислительный ресурс. Попутно будет разработан естественный класс задач, превышающих по сложности *NP* и со-*NP*.

9.1. PSPACE

Мы будем изучать класс PSPACE — множество всех задач, которые могут быть решены алгоритмом с полиномиальной пространственной сложностью (то есть алгоритмом с затратами пространства, полиномиальными по размерам входных данных).

Для начала рассмотрим отношение PSPACE к классам задач, которые рассматривались выше. Прежде всего, за полиномиальное время алгоритм может потреблять не более чем полиномиальное пространство; итак, можно утверждать:

(9.1) $P \subseteq PSPACE$

Однако множество PSPACE намного шире. Например, рассмотрим алгоритм, который просто считает от 0 до $2^n - 1$ в двоичной системе. Он просто реализует n-разрядный счетчик, который работает по тому же принципу, что и одометр в автомобиле. Таким образом, алгоритм выполняется за экспоненциальное время, после чего останавливается; при этом он использует полиномиальное пространство. И хотя этот алгоритм не делает ничего особенно интересного, он демонстрирует один важный принцип: пространство может повторно использоваться в вычислениях, тогда как для времени это по определению невозможно.

А вот более впечатляющее применение этого принципа.

(9.2) Существует алгоритм, решающий задачу 3-SAT с полиномиальными затратами пространства.

Доказательство. Просто используем алгоритм «грубой силы», который перебирает все возможные логические присваивания; каждое присваивание подается на множество условий для проверки того, выполняет ли оно эти условия. Важно реализовать это все с полиномиальными затратами пространства.

Для этого мы увеличиваем n-разрядный счетчик с 0 до $2^n - 1$, как описано выше. Значения счетчика ставятся в соответствие логическим присваиваниям: когда счетчик достигает значения q, оно интерпретируется как присваивание v, при котором x_i содержит значение i-го бита q.

Таким образом, полиномиальное пространство выделяется под перебор всех возможных вариантов логического присваивания v. Для каждого логического присваивания достаточно полиномиального пространства, чтобы подать данные на набор условий и проверить, обеспечивается ли их выполнение. Если условия выполняются, алгоритм можно немедленно остановить. Если условия не выполняются, мы удаляем промежуточные данные, задействованные в этой итерации, и заново используем память для следующего варианта присваивания. Таким образом, для проверки всех вариантов достаточно полиномиального пространства; тем самым доказывается граница для пространственных затрат алгоритма. ■

Так как 3-SAT является *NP*-полной задачей, у (9.2) имеется одно важное следствие.

(9.3) $NP \subseteq PSPACE$

Доказательство. Рассмотрим произвольную задачу Y из *NP*. Так как $Y \leq_P$ 3-SAT, существует алгоритм, который решает Y с полиномиальным количеством шагов плюс полиномиальное количество обращений к «черному ящику» для 3-SAT. Используя алгоритм в (9.2) для реализации «черного ящика», мы получаем алгоритм для Y, использующий только полиномиальное пространство. ■

Как и в случае с классом *P*, задача X принадлежит *PSPACE* в том, и только в том случае, если ее дополняющая задача также принадлежит *PSPACE*. Из этого следует, что *co-NP* $\subseteq$ *PSPACE*. Структурная диаграмма этих классов задач изображена на рис. 9.1.

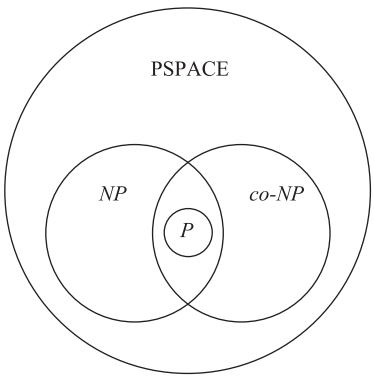


Рис. 9.1. Отношения между подмножествами различных классов задач. Учтите, что гипотезы о том, что все эти классы отличны друг от друга, до сих пор не доказаны

Так как *PSPACE* содержит невероятно огромный класс задач, включая как *NP*, так и *co-NP*, весьма вероятно, что в нем присутствуют задачи, которые не решаются за полиномиальное время. Но несмотря на это широко распространенное мнение, как ни удивительно, до сих пор не доказано, что $P \neq PSPACE$. Тем не менее почти общепринято мнение, что в *PSPACE* присутствуют задачи, не входящие даже в *NP* или *co-NP*.

9.2. Некоторые сложные задачи из PSPACE

Рассмотрим естественные примеры задач из *PSPACE*, которые, как принято считать, не принадлежат ни *NP*, ни *co-NP*.

Как и в случае с *NP*, чтобы разобраться в структуре *PSPACE*, можно рассмотреть конкретные задачи — самые сложные в классе. Задача X называется *PSPACE*-полной, если (i) она принадлежит *PSPACE* и (ii) для всех задач Y в *PSPACE* выполняется $Y \leq_P X$.

Оказывается, как и в случае с *NP*, существует широкий диапазон естественных задач, которые являются *PSPACE*-полными. Многие базовые задачи в области искусственного интеллекта являются *PSPACE*-полными; ниже описаны три категории таких задач.

Задачи построения плана

Задачи построения плана отражают процесс взаимодействия со сложной средой для достижения заданных целей. К каноническим примерам такого рода относятся крупные логистические операции, связанные с перемещением людей, оборудования и материалов. Например, при координации помощи при стихийных бедствиях может быть принято решение о том, что на определенной высоте над уровнем моря нужны двадцать машин «скорой помощи». Но чтобы решить эту задачу, нужно сначала подогнать десять снегоуборочных машин для расчистки дороги; этот этап, в свою очередь, требует заправки снегоуборочных машин и доставки водителей; но если израсходовать топливо для снегоуборочных машин, его может не хватить для «скорой помощи»; а если... в общем, вы поняли. Военные операции также требуют координации в большом масштабе, и методы автоматизированного планирования из области искусственного интеллекта успешно применялись для решения таких задач.

Очень похожие проблемы встречаются в сложных головоломках для одного игрока: например, в кубике Рубика или головоломке «15» — сетке 4×4 с 15 перемещаемыми плитками 1, 2, ..., 15 и одним пустым местом — перемещая плитки, игрок должен выстроить числа по возрастанию. (Вместо машин «скорой помощи» и снегоуборочных машин в этой задаче речь идет о таких операциях, как перемещение плитки 6 на одну позицию влево; но для этого нужно убрать с пути плитку 11, а для этого нужно переместить 9, которая находится в хорошем месте;

и т. д.) Даже эти «игрушечные» задачи порой оказываются весьма нетривиальными и часто используются в области искусственного интеллекта для тестирования координационных алгоритмов.

Как же с учетом сказанного определить задачу построения плана на достаточно общем уровне, включающем все эти примеры? И у головоломок, и у помощи при стихийных бедствиях имеются некоторые общие абстрактные черты: есть ряд условий, которые достигаются в результате принятия решений, и множество допустимых операций, которые к этим условиям применяются. Таким образом, окружение моделируется множеством $C = \{C_1, ..., C_n\}$ условий: общее «состояние окружающего мира» задается подмножеством условий, действующих на данный момент. Для взаимодействия с окружением используются операторы, образующие множество $\{O_1, ..., O_k\}$. Каждый оператор O_i определяется *множеством предусловий*, которые должны выполняться для выполнения O_i; *списком добавления* — множеством условий, которые станут истинными после выполнения O_i; и *списком удаления* — множеством условий, которые перестают действовать после выполнения O_i. Например, при моделировании головоломки «15» можно определить условие для каждого возможного положения каждой плитки и оператор для перемещения каждой плитки между двумя парами соседних положений; предусловие для оператора заключается в том, что в двух указанных позициях должны находиться плитка и пустое место.

Обобщенная задача выглядит так: дано множество C_0 начальных условий и множество C^* *целевых условий*; возможно ли применить последовательность операторов, начиная с C_0, и достичь ситуации, в которой выполняются только условия C^* (и никакие другие)? Назовем эту формулировку *задачей построения плана*.

Кванторы

Как вы убедились в задаче 3-SAT, проверка возможности одновременного выполнения множества дизъюнктивных условий создает немалые трудности. С добавлением кванторов задача только усложняется.

Пусть $\Phi(x_1, ..., x_n)$ — булева формула вида

$$C_1 \wedge C_2 \wedge ... \wedge C_k,$$

где каждый компонент C_i является дизъюнкцией трех литералов (другими словами, это экземпляр задачи 3-SAT). Для простоты будем считать, что n — нечетное число; задается вопрос:

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, ..., x_n)?$$

То есть мы хотим знать, существует ли для x_1 такой выбор, что для обоих вариантов выбора x_2 существует выбор x_3... и т. д., чтобы в итоге выполнялось Ф? Назовем эту задачу *задачей* 3-*SAT с кванторами*, или сокращенно *QSAT*.

Для сравнения, исходная задача 3-SAT задавала вопрос:

$$\exists x_1 \exists x_2 \cdots \exists x_{n-2} \exists x_{n-1} \exists x_n \Phi(x_1, ..., x_n)?$$

Другими словами, в задаче 3-SAT было достаточно найти одно значение для булевых переменных.

Следующий пример демонстрирует рассуждения, лежащие в основе экземпляра QSAT. Предположим, имеется формула

$$\Phi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$$

и задается вопрос:

$$\exists x_1 \forall x_2 \exists x_3 \Phi(x_1, x_2, x_3)?$$

Ответ на вопрос положителен: мы присваиваем x_1 такое значение, чтобы для обоих вариантов x_2 можно было задать x_3 так, чтобы выполнялось Ф. А именно: можно присвоить $x_1 = 1$; если затем x_2 присваивается 1, то x_3 можно присвоить 0 (с выполнением всех условий); а если x_2 присваивается 0, x_3 можно присвоить 1 (снова с выполнением всех условий).

Задачи такого типа с кванторами возникают естественным образом как разновидность *ситуационного планирования* — мы хотим знать, можно ли принять такое решение (выбор x_1), чтобы для всех возможных реакций (выбор x_2) можно было принять решение (выбор x_3)... и т. д.

Игры

В 1996 и 1997 годах в СМИ чемпиона мира по шахматам Гарри Каспарова называли защитником человеческой расы, когда он противостоял программе IBM Deep Blue в двух шахматных матчах. Даже один этот пример убедительно показывает, что интеллектуальные игры стали одним из самых заметных успехов в области современного искусственного интеллекта.

Многие игры для двух участников естественным образом укладываются в следующую схему: игроки ходят по очереди, и первый, кто достигает конкретной цели, выигрывает (такой целью может быть мат королю, захват всех шашек противника, построение ряда из четырех фишек и т. д.). Кроме того, часто существует естественная полиномиальная верхняя граница для максимально возможной длины игры.

Задача конкурентного размещения, упоминавшаяся в главе 1, естественно укладывается в эту структуру (а заодно показывает, что игры могут пригодиться не только для времяпрепровождения, но и в конкурентных ситуациях в повседневной жизни). Вспомните, что в задаче конкурентного размещения задается граф G с неотрицательным значением b_i, связанным с каждым узлом i. Два игрока поочередно выбирают узлы G, так что множество выбранных узлов в любой момент образует независимое множество. Игрок 2 побеждает, если ему удастся выбрать узлы с суммарным значением не менее B для заданной границы B; игрок 1 побеждает, если ему удастся этому помешать. Вопрос: существует ли для заданного графа G и границы B стратегия, с которой игрок 2 может обеспечить себе выигрыш?

9.3. Решение задач с кванторами и игровых задач в полиномиальном пространстве

В этом разделе мы поговорим о том, как все эти задачи решаются в полиномиальном пространстве. Как вы увидите, это будет сложнее (а в одном случае намного сложнее) той (простой) проблемы, с которой мы столкнулись при доказательстве принадлежности *NP* таких задач, как 3-SAT и задача о независимом множестве.

Начнем с задачи QSAT и задачи конкурентного размещения, а в следующем разделе будет рассмотрена задача построения плана.

Разработка алгоритма для QSAT

Сначала мы покажем, что задача QSAT может быть решена с полиномиальными затратами пространства. Как и в случае с 3-SAT, идея заключается в выполнении алгоритма «грубой силы», обеспечивающего экономное использование пространства в процессе вычислений.

Базовое решение методом «грубой силы» выглядит так: чтобы разобраться с первым квантором $\exists x_1$, мы последовательно рассмотрим оба возможных значения x_1. Сначала мы назначаем $x_1 = 0$ и рекурсивно проверяем, дает ли оставшаяся часть формулы значение 1. Затем для значения $x_1 = 1$ проводится рекурсивная проверка того, дает ли оставшаяся часть формулы значение 1. Полная формула дает результат 1 в том, и только в том случае, если один из этих рекурсивных вызовов возвращает 1 — просто по определению квантора $\exists$.

По сути, это алгоритм «разделяй и властвуй», который для входных данных с n переменными порождает два рекурсивных вызова с входными данными с $n - 1$ переменной каждый. Если сохранять всю работу, проделанную при рекурсивных вызовах, использование пространства $S(n)$ будет удовлетворять рекуррентному отношению

$$S(n) \leq 2S(n-1) + p(n),$$

где $p(n)$ — полиномиальная функция. Получается экспоненциальная граница, которая слишком велика.

К счастью, можно провести простую оптимизацию, которая значительно сократит затраты пространства. Разобравшись со случаем $x_1 = 0$, достаточно только сохранить один бит, представляющий результат рекурсивного вызова; всю остальную промежуточную работу можно отбросить. Таким образом, пространство, использованное при вычислениях для $x_1 = 0$, используется в вычислениях для $x_1 = 1$.

Ниже приведено компактное описание алгоритма.

```
Если первым квантором является ∃x_i
   Присвоить x_i = 0 и рекурсивно вычислить выражение под квантором
             для остальных переменных
   Сохранить результат (0 или 1) и удалить все промежуточные результаты
   Присвоить x_i = 1 и рекурсивно вычислить выражение под квантором
             для остальных переменных
```

```
   Если в одном из вычислений будет получено значение 1
      Вернуть 1
   Иначе вернуть 0
   Конец Если
Если первым квантором является ∀x_i
   Присвоить x_i = 0 и рекурсивно вычислить выражение под квантором
            для остальных переменных
   Сохранить результат (0 или 1) и удалить все промежуточные результаты
   Присвоить x_i = 1 и рекурсивно вычислить выражение под квантором
            для остальных переменных
   Если в обоих вычислениях будет получено значение 1
      Вернуть 1
   Иначе вернуть 0
   Конец Если
Конец Если
```

Анализ алгоритма

Так как рекурсивные вызовы для $x_1 = 0$ и $x_1 = 1$ используют одно пространство, затраты пространства $S(n)$ для задачи с n переменными просто полиномиальны по n плюс затраты пространства для одного рекурсивного вызова для задачи с $n - 1$ переменной.

$$S(n) \leq S(n-1) + p(n),$$

где $p(n)$ — полиномиальная функция. Раскручивая это рекуррентное отношение, получаем

$$S(n) \leq P(n) + p(n-1) + p(n-2) + \cdots + p(1) \leq n \cdot p(n).$$

Так как $p(n)$ — полиномиальная функция, то и $n \cdot p(n)$ тоже, а следовательно, затраты пространства полиномиальны по n, как и требовалось:

Получаем следующий результат.

(9.4) Задача QSAT решается с полиномиальными затратами пространства.

Расширения: алгоритм для задачи конкурентного размещения

Для определения того, кто из игроков может обеспечить себе форсированный выигрыш в игре типа конкурентного размещения, можно воспользоваться очень похожим алгоритмом.

Предположим, игрок 1 ходит первым. Мы последовательно рассматриваем все возможные ходы. Для каждого из ходов мы проверяем, кто из игроков имеет форсированный выигрыш в полученной игре с первым ходом игрока 2. Если игрок 1 имеет форсированный выигрыш в какой-либо из этих игр, то он имеет форсированный выигрыш и в исходной позиции. Здесь, как и в случае алгоритма QSAT, важно

то, что пространство может повторно использоваться при перемещении между кандидатами; достаточно хранить один бит, представляющий результат. В этом случае алгоритм потребляет полиномиальное пространство плюс пространство, необходимое для одного рекурсивного вывода в графе с меньшим количеством узлов. Как и в случае QSAT, мы получаем рекуррентное отношение

$$S(n) \leq S(n-1) + p(n)$$

для полиномиальной функции $p(n)$.

В итоге получаем следующий результат:

(9.5) Задача конкурентного размещения решается с полиномиальными затратами пространства.

9.4. Решение задачи построения плана с полиномиальным пространством

Теперь посмотрим, как решить базовую задачу построения плана с полиномиальными затратами пространства. Эта проблема заметно отличается от предыдущей и оказывается намного более сложной.

Задача

Вспомните, что в формулировке задачи используется множество условий $C = \{C_1, ..., C_n\}$ и множество операторов $\{O_1, ..., O_k\}$. Каждый оператор O_i имеет список предусловий P_i, список добавления A_i и список удаления D_i. Обратите внимание: оператор O_i может применяться и при наличии условий, не входящих в P_i, и он не влияет на условия, не входящие в A_i или D_i.

Конфигурация определяется как подмножество $C' \subseteq C$; состояние задачи в любой момент времени идентифицируется уникальной конфигурацией C', которая определяется условиями, действовавшими на тот момент. Для исходной конфигурации C_0 и целевой конфигурации C^* требуется определить, существует ли последовательность операций, которая переходит из C_0 в C^*.

Экземпляр задачи построения плана может рассматриваться в контексте очень большого, неявно определенного направленного графа G. В G существует узел для каждой из 2^n возможных конфигураций (то есть для каждого возможного подмножества C); и в G существует ребро от конфигурации C' к конфигурации C'', если на одном шаге один из операторов может преобразовать C' к C''.

В контексте этого графа задача построения плана имеет очень естественную формулировку: существует ли в G путь из C_0 в C^*? Такой путь точно соответствует последовательности операторов, приводящей от C_0 к C^*.

Экземпляр задачи может иметь короткое решение (как в случае с головоломкой «15»), но в общем случае на это не стоит рассчитывать. Другими словами, в G не

всегда существует короткий путь из C_0 в C^* — и это неудивительно, так как в G экспоненциальное количество узлов. Однако интуицию следует применять осторожно, поскольку граф G имеет особую структуру: он очень компактно определяется в контексте n условий и k операторов.

(9.6) Существуют экземпляры задачи построения плана с n условиями и k операторами, для которых существует решение, но кратчайшее решение имеет длину $2^n - 1$.

Доказательство. Приведем короткий пример такого экземпляра; по сути, он кодирует задачу увеличения n-разрядного счетчика из состояния «только нули» в состояние «только единицы».

- Имеются условия $C_1, C_2, ..., C_n$.
- Имеются операторы O_i для $i = 1, 2, ..., n$.
- Оператор O_1 не имеет предусловий или списка удаления; он просто добавляет C_1.
- Для $i > 1$ предусловиями O_i являются C_j для всех $j < i$. При вызове он добавляет C_i и удаляет C_j для всех $j < i$.

Вопрос: существует ли последовательность операторов, которая переводит от $C_0 = \varphi$ к $C^* = \{C_1, C_2, ..., C_n\}$?

Следующее утверждение доказывается индукцией по i:

Из любой конфигурации, не содержащей C_j для всех $j \le i$, существует последовательность операторов, которая достигает конфигурации, содержащей C_j для всех $j \le i$; но любая такая последовательность состоит не менее чем из $2^i - 1$ шагов.

Для $i = 1$ истинность утверждения очевидна. Для больших значений i одно из возможных решений выглядит так:

- По индукции достигнуть условий $\{C_{i-1}, ..., C_1\}$ с использованием операторов $O_1, ..., O_{i-1}$.
- Теперь вызовем оператор O_i, добавляя C_i, но удаляя все остальное.
- Снова по индукции достигнуть условий $\{C_{i-1}, ..., C_1\}$ с использованием операторов $O_1, ..., O_{i-1}$. Обратите внимание: условие C_i при этом сохраняется.

Теперь разберемся со второй частью шага индукции — что любая такая последовательность требует не менее $2^i - 1$ шагов. Рассмотрим момент первого добавления C_i. На этом шаге $C_{i-1}, ..., C_1$ должны присутствовать, и по индукции для этого должно было потребоваться не менее $2^{i-1} - 1$ шагов. Условие C_i может быть добавлено только оператором O_i, приводящим к удалению всех C_j для $j < i$. Теперь нужно снова достичь условий $\{C_{i-1}, ..., C_1\}$; по индукции это потребует еще $2^{i-1} - 1$ шагов, что в сумме дает не менее $2(2^{i-1} - 1) + 1 = 2^i - 1$ шагов.

Общая граница доказывается применением этого утверждения с $i = n$. ■

Конечно, если бы каждый «положительный» экземпляр задачи построения плана имел решение полиномиальной длины, то задача принадлежала бы NP — можно было бы просто представить решение. Но (9.6) показывает, что кратчайшее решение не обязательно является хорошим сертификатом для задачи построения плана, потому что оно может иметь длину, экспоненциальную по размеру входных данных.

Однако (9.6) фактически определяет худший случай. Граф G содержит 2^n узлов, и если существует путь из C_0 в C^*, то кратчайший такой путь не посещает ни один узел более одного раза. Из этого следует, что кратчайший путь может содержать не более $2^n - 1$ шагов после выхода из C_0.

(9.7) Если экземпляр задачи планирования с n условиями имеет решение, то он имеет и решение, использующее не более $2^n - 1$ шагов.

Разработка алгоритма

Как вы видели, кратчайшее решение задачи построения плана может иметь длину, экспоненциальную по n, и это неприятно: в конечном итоге из этого следует, что в полиномиальном пространстве невозможно даже сохранить явное представление решения. Но это факт не обязательно лишает нас надежды на решение произвольного экземпляра задачи построения плана с полиномиальными затратами пространства. Ведь может существовать алгоритм, который определяет ответ на экземпляр задачи построения плана без проверки всего решения.

Собственно, в данном случае дело обстоит именно так: мы проектируем алгоритм для решения задачи построения плана с полиномиальным пространством.

Некоторые экспоненциальные методы

Чтобы задача стала более понятной, мы сначала рассмотрим алгоритм «грубой силы». Построим граф G и воспользуемся любым алгоритмом связности графа (поиском в глубину или поиском в ширину) для принятия решения о том, существует ли в графе путь от C_0 к C^*.

Конечно, для наших целей этот алгоритм слишком примитивен; построение графа G потребует экспоненциальных затрат пространства. Можно опробовать метод, в котором граф G реально не строится, а для него просто моделируется поведение поиска в ширину или глубину. Впрочем, это решение тоже неприемлемо; для поиска в глубину неизбежно придется вести список всех узлов в текущем исследуемом пути, а он может разрастись до экспоненциального размера. Для поиска в ширину потребуется список всех узлов текущего «фронта» поиска, который тоже может вырасти до экспоненциального размера.

Похоже, мы оказались в тупике. Наша задача по сути эквивалентна поиску пути в G, а известные нам стандартные алгоритмы поиска путей слишком неэкономны в расходовании пространства. Не существует ли принципиально иного алгоритма поиска пути?

Более эффективный по затратам пространства способ построения путей

Оказывается, принципиально иная разновидность алгоритмов поиска пути существует и обладает как раз нужными нам свойствами. Основная идея, предложенная Савичем в 1970 году, заключается в умном применении принципа «разделяй

и властвуй». Впоследствии этот прием был заложен в основу сокращения затрат пространства в задаче выравнивания последовательностей; по этой причине общий подход может напомнить то, что обсуждалось ранее в разделе 6.7. Наш план, как и прежде, основан на умном повторном использовании пространства — откровенно говоря, за счет возрастания затрат времени. Как поиск в глубину, так и поиск в ширину не проявляет достаточного рвения в повторном использовании пространства; оба должны постоянно вести большой исторический список. Нам нужен способ решить половину задачи, отбросить почти всю промежуточную работу, а затем решить другую половину задачи.

Ключевая роль в этом алгоритме отводится процедуре, которую мы назовем Path(C_1, C_2, L). Процедура определяет, существует ли последовательность операторов, состоящая не более чем из L шагов, которая приводит из конфигурации C_1 в конфигурацию C_2. Итак, исходной задачей является определение результата (да или нет) для Path(C_0, C^*, 2^n). Поиск в ширину может рассматриваться как следующая реализация этой процедуры средствами динамического программирования: чтобы определить Path(C_1, C_2, L), мы сначала определяем все C', для которых выполняется Path(C_1, C', $L-1$); затем мы проверяем для всех таких C', ведет ли какой-нибудь оператор напрямую из C' в C_2.

Это отчасти демонстрирует неэффективность в отношении затрат пространства, присущую поиску в ширину. Мы генерируем множество промежуточных конфигураций только для того, чтобы уменьшить параметр L на 1. Эффективнее было бы попытаться определить, существует ли какая-либо конфигурация C', которая служит срединной точкой пути из C_1 в C_2. Можно сначала сгенерировать все возможные срединные точки C'. Затем для всех C' можно рекурсивно проверить, можно ли перейти из C_1 в C' не более чем за $L/2$ шагов и можно ли перейти из C' в C_2 не более чем за $L/2$ шагов. Для этого потребуются два рекурсивных вызова, но нас интересуют только результаты «да/нет» для каждого из них; в остальном пространство можно использовать повторно.

Позволит ли этот прием сократить затраты пространства до полиномиальной величины? Сначала тщательно сформулируем процедуру, а затем проанализируем ее. Для наших целей L будет считаться степенью 2.

```
Path(C1, C2, L)
   Если L = 1
      Если существует оператор O, преобразующий C1 в C2
         Вернуть "да"
      Иначе
         Вернуть "нет"
      Конец Если
   Иначе (L > 1)
      Перебрать все конфигурации C' с использованием n-разрядного счетчика
      Для каждого C' сделать следующее:
         Вычислить x = Path(C1, C', L/2)
         Удалить все промежуточные результаты, сохранить только
            возвращаемое значение x
         Вычислить y = Path(C',C2, L/2)
         Удалить все промежуточные результаты, сохранить только
```

```
        возвращаемое значение y
      Если x и y равны "да", вернуть "да"
    Конец
    Если ни для одного C' не был получен результат "да"
      Вернуть "нет"
    Конец Если
  Конец Если
```

И снова обратите внимание на то, что эта процедура решает проблему обобщения исходного вопроса, относившегося к Path(C_0, C^*, 2^n). Однако это означает, что L следует рассматривать как экспоненциально большой параметр: $\log L = n$.

Анализ алгоритма

Следующее утверждение показывает, что задача построения плана может быть решена с полиномиальными затратами пространства.

(9.8) Path(C_1, C_2, L) возвращает «да» в том, и только в том случае, если существует последовательность операторов длины не более L, ведущая из C_1 в C_2. Ее использование пространства полиномиально по n, k и $\log L$.

Доказательство. Утверждение доказывается индукцией по L. Оно очевидным образом выполняется при $L = 1$, так как все операторы рассматриваются явно. Теперь рассмотрим большее значение L. Если существует последовательность операторов из C_1 в C_2 длины $L' \le L$, то существует конфигурация C', находящаяся в позиции $\lceil L'/2 \rceil$ в этой последовательности. По индукции оба вызова Path(C_1, C', $\lceil L/2 \rceil$) и Path(C', C_2, $\lceil L/2 \rceil$) вернут «да», поэтому Path(C_1, C_2, L) вернет «да». И наоборот, если существует конфигурация C', при которой Path(C_1, C', $\lceil L/2 \rceil$) и Path(C',C_2, $\lceil L/2 \rceil$) вернут «да», то из гипотезы индукции следует, что существуют соответствующие последовательности операторов; объединяя эти две последовательности, получаем последовательность операторов из C_1 в C_2 длины не более L.

Теперь рассмотрим требования к пространству. Помимо затрат в рекурсивных вызовах, каждый вызов Path требует затрат пространства, полиномиальных по n, k и $\log L$. Но в любой заданный момент времени активен только один рекурсивный вызов, и промежуточные результаты всех остальных рекурсивных вызовов были удалены. Следовательно, для полиномиальной функции p требования к пространству $S(n, k, L)$ удовлетворяют рекуррентному отношению

$$S(n, k, L) \le P(n, k, \log L) + S(n, k, L/2).$$

$$S(n, k, 1) \le P(n, k, 1).$$

Раскручивая рекуррентное отношение для $O(\log L)$ уровней, мы получаем границу $S(n, k, L) = O(\log L \cdot p(n, k, \log L)$, полиномиальную по n, k и $\log L$. ■

Если у динамического программирования существует противоположность, то это она и есть. При решении задач методами динамического программирования принципиально сохранять все промежуточные результаты, чтобы их не прихо-

дилось вычислять заново. Теперь, когда мы стремимся к экономии пространства, действуют прямо противоположные приоритеты: отбросить все промежуточные результаты, так как они только занимают место и их всегда можно вычислить заново.

Как было показано при разработке алгоритма выравнивания последовательностей, эффективного по пространству, лучшая стратегия часто лежит где-то в середине и базируется на том, чтобы отбросить часть промежуточной работы, но без значительного ущерба для времени выполнения.

9.5. Доказательство PSPACE-полноты задач

В начале изучения *NP* нам пришлось доказать *NP*-полноту первой задачи прямо из определения *NP*. После того как Кук и Левин сделали это для выполнимости, ученые могли применить эту схему сведения ко многим другим *NP*-полным задачам.

Вскоре после публикации результатов для *NP* аналогичные события происходили и для *PSPACE*. Вспомните, что мы определили PSPACE-полноту по прямой аналогии с *NP*-полнотой в разделе 9.1. Естественной аналогией для задачи выполнимости булевых схем или 3-SAT для *PSPACE* является задача QSAT; Стокмейер и Мейер (1973) доказали:

(9.9) Задача QSAT является PSPACE-полной.

Базовая PSPACE-полная задача служит хорошим «корнем», от которого начинается поиск других PSPACE-полных задач. По четкой аналогии с *NP* из определения легко увидеть, что если задача Y является PSPACE-полной и задача X в *PSPACE* обладает свойством $Y \leq_P X$, то задача X также является PSPACE-полной.

В этом разделе мы приведем пример такого доказательства PSPACE-полноты для задачи конкурентного размещения; для этого задача QSAT будет сведена к задаче конкурентного размещения. Помимо установления сложности задачи конкурентного размещения, сведение также дает представление о том, как следует подходить к демонстрации PSPACE-полноты для игр вообще на основании их близкой связи с кванторами.

Заметим, что PSPACE-полнота задачи построения плана также может быть продемонстрирована сведением от QSAT, но здесь это доказательство не приводится.

Связь задач с кванторами с игровыми задачами

Не приходится удивляться тому, что между задачами с кванторами и игровыми задачами существует тесная связь. В самом деле, задачу QSAT можно было бы эквивалентно определить как задачу принятия решения о том, имеет ли первый игрок форсированный выигрыш в следующем «игровом варианте» 3-SAT: допустим, мы фиксируем формулу $\Phi(x_1, \cdots, x_n)$, которая состоит, как и в QSAT, из конъюнкции

условий длины 3. Два игрока поочередно выбирают значения переменных: первый игрок выбирает значение x_1, затем второй игрок выбирает значение x_2, потом первый выбирает значение x_3 и т. д. Считается, что первый игрок побеждает, если при вычислении $\Phi(x_1, \cdots, x_n)$ будет получен результат 1, а второй — если результат равен 0.

Когда у первого игрока имеется форсированная победа в этой игре (то есть когда наш экземпляр конкурентной задачи 3-SAT имеет положительный ответ)? Тогда, когда существует такой выбор x_1, что при всех вариантах выбора x_2 существует такой выбор x_3, что... и т. д., при котором $\Phi(x_1, \cdots, x_n)$ дает результат 1. То есть форсированная победа первого игрока возможна в том, и только в том случае, если (в предположении, что n является нечетным числом)

$$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, \ldots, x_n).$$

Другими словами, конкурентная задача 3-SAT эквивалентна экземпляру QSAT, определяемому по той же булевой формуле Ф, поэтому мы доказали следующее утверждение:

(9.10) *QSAT* $\leq_P$ *Конкурентная задача* 3-*SAT* и *Конкурентная задача* 3-*SAT* $\leq_P$ *QSAT*.

Доказательство PSPACE-полноты задачи конкурентного размещения

Утверждение (9.10) открывает путь в мир игровых задач. Мы воспользуемся этой связью для доказательства PSPACE-полноты задачи конкурентного размещения.

(9.11) Задача конкурентного размещения является PSPACE-полной.

Доказательство. Мы уже показали, что задача конкурентного размещения принадлежит *PSPACE*. Чтобы доказать ее PSPACE-полноту, мы теперь покажем, что *Конкурентная задача* 3-*SAT* $\leq_P$ *Конкурентное размещение*, и тем самым установим ее PSPACE-полноту.

Имеется экземпляр конкурентной задачи 3-SAT, определяемый формулой Ф. Ф представляет собой конъюнкцию условий

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k.$$

Каждое условие C_j имеет длину 3 и может быть записано в виде $C_j = t_{j1} \vee t_{j2} \vee t_{j3}$. Как и прежде, будем считать, что количество переменных n нечетно. Также будем полагать (по естественным причинам), что никакое условие не содержит литерал одновременно с его отрицанием; в конце концов, такое условие автоматически выполняется любым логическим присваиванием. Мы должны показать, как закодировать эту логическую структуру в графе, лежащем в основе задачи конкурентного размещения.

Экземпляр конкурентной задачи 3-SAT можно закодировать в следующем виде. Игроки поочередно выбирают значения в логическом присваивании, начиная и заканчивая игроком 1; в конце игрок 2 выигрывает, если он выберет условие C_j,

в котором ни одному литералу не было задано значение 1. Игрок 1 выигрывает, если игроку 2 это сделать не удастся.

В такой формулировке нам хотелось бы закодировать экземпляр задачи конкурентного размещения: игроки поочередно делают фиксированное количество ходов, и в конце существует вероятность того, что игрок 2 победит. Но в общей формулировке задача конкурентного размещения выглядит намного более открытой. Если игроки в конкурентной задаче 3-SAT должны задавать по одной переменной в строгом порядке, игроки в задаче конкурентного размещения могут перемещаться по всему графу, выбирая любые узлы на свое усмотрение.

Для решения этой проблемы мы воспользуемся значениями b_i узлов, которые будут ограничивать возможности перемещения игроков при любой «разумной» стратегии. Иначе говоря, все будет настроено так, что при отклонении любого из игроков от конкретного узкого пути он моментально проигрывает.

Как и в случае с более сложными сведениями *NP*-полноты в главе 8, наше построение будет содержать регуляторы, представляющие присваивание переменным, и другие регуляторы для представления условий. Переменные будут кодироваться следующим образом: для каждой переменной x_i в графе G определяются два узла v_i, v'_i, а также ребро (v_i, v'_i), как показано на рис. 9.2. Выбор v_i представляет присваивание $x_i = 1$; выбор v'_i представляет $x_i = 0$. Ограничение, согласно которому выбранные переменные должны образовать независимое множество, естественным образом предотвращают одновременный выбор v_i и v'_i. В этой точке никакие другие ребра не определяются.

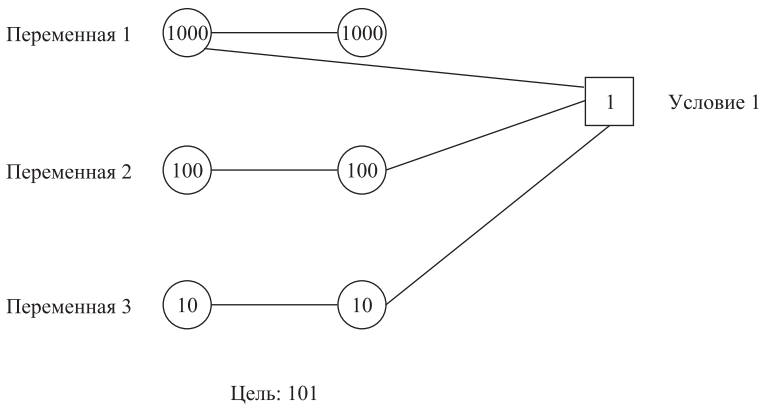


Рис. 9.2. Сведение конкурентной задачи 3-SAT к задаче конкурентного размещения

Как заставить игроков задавать переменные по порядку — сначала x_1, затем x_2 и т. д.? Значения v_1 и v'_1 задаются настолько высокими, что если игрок 1 не выберет их, он немедленно проиграет. Для v_2 и v'_2 устанавливаются значения поменьше, и т. д. А именно: для $c \geq k + 2$ мы определяем значения узлов b_{v_i} и $b_{v'_i}$ равными c^{1+n-i}. Граница, которую игрок 2 пытается достичь, определяется равной

$$B = c^{n-1} + c^{n-3} + \ldots + c^2 + 1.$$

Прежде чем переходить к условиям, остановимся на игре, которая играется на этом графе. Первым ходом игрок 1 должен выбрать один из узлов v_1 или v'_1 (тем самым отказываясь от другого); в противном случае игрок 2 немедленно выберет один из этих узлов для своего следующего хода и немедленно выиграет. Аналогичным образом вторым ходом игрок 2 должен выбрать один из узлов v_2 или v'_2, иначе игрок 1 выберет его следующим ходом; и тогда, даже если игрок 2 заберет все остальные узлы графа, он не сможет выполнить границу B. Продолжая индукцию, мы видим, что для предотвращения неизбежного проигрыша игрок, делающий i-й ход, должен выбрать один из узлов v_i или v'_i. С таким выбором узлов мы добиваемся ровно того эффекта, который был нужен: игроки должны задавать переменные по очереди. И каким же будет результат для этого графа? Игрок 2 завершает игру с суммой $c^{n-1} + c^{n-3} + ... + c^2 = B - 1$: он проиграл на одну единицу!

Завершим аналогию с конкурентной задачей 3-SAT, предоставляя игроку 2 один последний ход, которым он может попытаться выиграть. Для каждого условия C_j определяется узел c_j со значением $b_{c_j} = 1$ и ребром, связанным с каждым из его литералов, следующим образом: если $t = x_i$, добавляется ребро (c_j, v_i); если $t = \overline{x}_i$, добавляется ребро (c_j, v'_i). Другими словами, мы присоединяем c_j к узлу, который представляет литерал t.

Так определяется полный граф G. Мы можем убедиться в том, что из-за малых значений добавление узлов условий не изменило то свойство, что игроки начинают с выбора узлов переменных $\{v_i, v'_i\}$ в правильном порядке. Однако после того, как это будет сделано, игрок 2 победит в том, и только в том случае, если он сможет выбрать узел условия c_j, не смежный ни с одним узлом выбранной переменной, — другими словами, в том, и только в том случае, если логическое присваивание, попеременно определяемое игроками, не выполняет никакого условия.

Таким образом, игрок 2 может победить в определенном нами экземпляре задачи конкурентного размещения в том, и только в том случае, если он сможет победить в исходном экземпляре конкурентной задачи 3-SAT. Сведение завершено. ■

Упражнения с решениями

Упражнение с решением 1

Самоизбегающие блуждания изучаются в области статистической физики; их можно определить следующим образом. Пусть $\mathcal{L}$ — множество всех точек R^2 с целочисленными координатами (своего рода «сетка» на плоскости). Самоизбегающее блуждание W длины n представляет собой последовательность точек $(p_1, p_2, ..., p_n)$ из $\mathcal{L}$, для которых

(i) $p_1 = (0,0)$ (перемещение начинается от начала координат);

(ii) в последовательности нет двух одинаковых точек («маршрут» не повторяется);

(iii) для всех $i = 1, 2, ..., n - 1$ точки p_i и p_{i+1} находятся на расстоянии 1 друг от друга (перемещение происходит между соседними точками в $\mathcal{L}$).

Самоизбегающие блуждания (в двух и трех измерениях) используются в физической химии как простая геометрическая модель возможных конфигураций длинноцепочечных полимерных молекул. Такие молекулы можно рассматривать как гибкую цепь, которая может изгибаться, принимая разные геометрические структуры; самоизбегающие блуждания всего лишь предоставляют простую комбинаторную абстракцию для таких структур.

В этой области существует знаменитая нерешенная задача: для натурального числа $n \geq 1$ количество разных самоизбегающих блужданий длины n обозначается $A(n)$. Обратите внимание: блуждания рассматриваются как последовательности точек, а не как множества; даже если два блуждания проходят через одно множество точек, но в разном порядке, они считаются разными. (Формально блуждания $(p_1, p_2, ..., p_n)$ и $(q_1, q_2, ..., q_n)$ считаются разными, если для некоторого i $(1 \leq i \leq n)$ $p_i \neq q_i$.) Пример изображен на рис. 9.3. В полимерных моделях, основанных на самоизбегающих блужданиях, величина $A(n)$ напрямую связана с *энтропией* цепочечной молекулы и поэтому встречается в теориях, относящихся к частоте некоторых метаболических реакций и реакций органического синтеза.

Несмотря на важность величины $A(n)$, простая формула для нее неизвестна. Алгоритм для вычисления $A(n)$ с временем, полиномиальным по n, пока не найден.

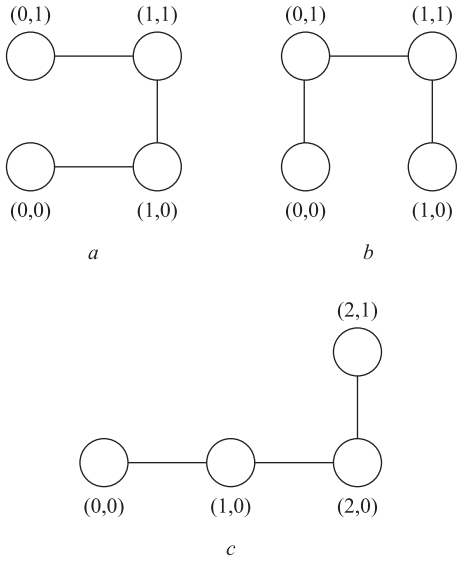


Рис. 9.3. Три разных самоизбегающих блуждания длины 4. Обратите внимание: блуждания (a) и (b) состоят из одинаковых множеств точек, но считаются разными, потому что они обходятся в разном порядке

(a) Покажите, что $A(n) \geq 2^{n-1}$ для натуральных чисел $n \geq 1$.

(b) Предложите алгоритм, который получает число n и выводит $A(n)$ как число в двоичной системе счисления с затратами пространства (то есть памяти), полиномиальными по n. (Таким образом, время выполнения алгоритма может быть экспоненциальным при условии полиномиальных затрат пространства.

Также обратите внимание на то, что под полиномиальностью имеется в виду «полиномиальность по n», а не «полиномиальность по $\log n$». В самом деле, из (a) следует, что для записи значения $A(n)$ потребуется не менее $n - 1$ битов, поэтому очевидно, что $n - 1$ является нижней границей для пространства, необходимого для получения правильного ответа.)

Решение

Для начала рассмотрим часть (b). На первый взгляд перебор всех самоизбегающих блужданий кажется сложной задачей; поиск естественно представляется как рост цепочки, растущей из одного звена, с анализом возможных конфигураций и отступлениями в том случае, когда продолжить рост без самопересечений не удается. Все мы видели экранные заставки, которые рисуют подобные картинки, но точно сформулировать алгоритм будет непросто.

Итак, сделаем шаг назад; полиномиальное пространство — очень свободная граница, и мы можем сделать шаг в сторону «грубой силы». Допустим, вместо того, чтобы пытаться перебрать все самоизбегающие блуждания длины n, мы просто переберем все блуждания длины n, а затем проверим, какие из них окажутся самоизбегающими. Преимущество такого решения в том, что пространство всех блужданий описывается намного проще, чем пространство самоизбегающих блужданий.

В самом деле, любое блуждание $(p_1, p_2, ..., p_n)$ для множества $\mathcal{L}$ точек сетки на плоскости может быть описано последовательностью выбираемых направлений. Каждый шаг от p_i к p_{i+1} может рассматриваться как перемещение в одном из четырех направлений: север, юг, восток или запад. Следовательно, любое блуждание длины n отображается на строку длины $n - 1$ в алфавите $\{N, S, E, W\}$. (Три блуждания на рис. 9.3 соответствуют строкам ENW, NES и EEN). Каждая такая строка соответствует блужданию длины n, но не все такие строки соответствуют самоизбегающим блужданиям: например, NESW возвращается к точке (0, 0).

Этот способ кодирования блужданий для части (b) вопроса можно использовать следующим образом. Используя счетчик по модулю 4, мы перебираем все строки длины $n - 1$ по алфавиту $\{N, S, E, W\}$, рассматривая этот алфавит в эквивалентном виде $\{0, 1, 2, 3\}$. Для каждой такой строки мы строим соответствующее блуждание и проверяем с полиномиальными затратами пространства, является ли оно самоизбегающим. Наконец, мы увеличиваем второй счетчик A (инициализируемый 0), если текущее блуждание является самоизбегающим. В конце алгоритма в A хранится значение $A(n)$.

Теперь мы можем ограничить пространство, используемое алгоритмом, следующим образом. Первый счетчик, используемый для перебора строк, состоит из $n - 1$ позиции, каждая из которых занимает два бита (для четырех возможных значений). Второй счетчик, содержащий A, может увеличиваться не более 4^{n-1} раз, поэтому для него понадобится не более $2n$ битов. Наконец, полиномиальное пространство используется для проверки того, является ли каждое сгенерированное блуждение самоизбегающим; но одно и то же пространство может использоваться для всех блужданий, поэтому необходимое пространство также является полиномиальным.

Описанная схема кодирования также помогает ответить на часть (a). Заметим, что все блуждания, которые могут быть закодированы с использованием только букв {*N*, *E*}, являются самоизбегающими, так как перемещение на плоскости осуществляется только вверх и вправо. Для этих двух букв существуют 2^{n-1} строки длины $n - 1$, поэтому существуют не менее 2^{n-1} самоизбегающих блужданий; другими словами, $A(n) \geq 2^{n-1}$.

(Ранее было показано, что наш метод кодирования также предоставляет верхнюю границу, из которой немедленно следует, что $A(n) \leq 4^{n-1}$.)

Упражнения

1. Рассмотрим частный случай задачи 3-SAT с кванторами, в котором булева формула не содержит инвертированных переменных. А именно, пусть $\Phi(x_1, ..., x_n)$ — булева формула вида

   $$C_1 \wedge C_2 \wedge ... \wedge C_k,$$

   где каждое условие C_i является дизъюнкцией трех литералов. *Монотонность* Φ означает, что каждый литерал в каждом условии состоит из неинвертированной переменной, то есть что каждый литерал равен x_i для некоторого i, а не $\overline{x_i}$.

   Монотонная задача QSAT определяется как задача принятия решения

   $$\exists x_1 \forall x_2 \cdots \exists x_{n-2} \forall x_{n-1} \exists x_n \Phi(x_1, ..., x_n),$$

   в которой формула Φ является монотонной.

   Сделайте одно из двух: (a) докажите, что монотонная задача QSAT является PSPACE-полной; или (b) предложите алгоритм для решения произвольных экземпляров монотонной задачи QSAT за время, полиномиальное по n. (Обратите внимание: в (b) целью является полиномиальное *время*, а не полиномиальное пространство.)

2. Рассмотрим следующую игру. Имеется множество географических названий — например, названий столиц. Первый игрок называет столицу c страны, в которой находится компания; второй игрок должен выбрать город c', начинающийся с буквы, которой заканчивается c. Далее игроки попеременно выбирают города, начинающиеся с последней буквы предыдущего города. Проигрывает тот, кто не может назвать город, еще не встречавшийся ранее в игре. Например, в начале игры, проходящей в Венгрии, первый игрок говорит «Будапешт»; далее следует продолжение (например): «Токио, Оттава, Анкара, Амстердам, Москва...»

   Конечно, игра помогает проверить знания географии, но даже с полным списком мировых столиц возникает серьезная стратегическая задача. Какое слово выбрать следующим, чтобы попытаться загнать противника в тупик, в котором у него не будет хода?

Чтобы выделить стратегический аспект, определим абстрактную версию игры, которая будет называться *географической игрой на графе*. Имеется направленный граф $G = (V, E)$ с начальным узлом $s \in V$. Игроки поочередно делают ходы, начиная с s; каждый игрок должен по возможности перейти по ребру от текущего узла к узлу, который ранее не посещался. Проигрывает тот игрок, который первым не сможет перейти к узлу, ранее посещавшемуся в игре (по прямой аналогии с игрой, только узлы соответствуют словам). Другими словами, игрок проигрывает, если в игре в данный момент текущим является узел v и для всех ребер вида (v, w) узел w уже посещался.

Докажите, что задача принятия решения о том, может ли первый игрок обеспечить себе победу в географической игре на графе, является PSPACE-полной.

3. Предложите алгоритм с полиномиальным временем для принятия решения о том, может ли первый игрок обеспечить себе победу в географической игре на графе в особом случае, когда граф G не имеет направленных циклов (иначе говоря, когда G является направленным ациклическим графом).

Примечания и дополнительная литература

PSPACE — всего лишь один класс неразрешимых задач за пределами *NP*; изучением этой области занимается теория сложности. По теме теории сложности написан ряд книг, таких как книги Пападимитриу (Papadimitriou, 1995) и Сэвиджа (Savage, 1998).

PSPACE-полноту задачи QSAT доказали Стокмейер и Мейер (Stockmeyer, Meyer, 1973).

Некоторые базовые результаты PSPACE-полноты для игр с двумя участниками приведены у Шефера (Schaefer, 1978), а также у Стокмейера и Чандры (Stockmeyer, Chandra, 1979). Задача конкурентного размещения, рассматриваемая здесь, является адаптированным примером класса задач, изучаемых более общей областью размещения объектов; например, обзорную информацию по этой теме можно найти в книге под редакцией Дрезнера (Drezner, 1995).

Игры для двух игроков постоянно порождали сложные вопросы для ученых в области математики и искусственного интеллекта. Берлекамп, Конуэй и Гай (Berlekamp, Conway, Guy, 1982) и Новаковски (Nowakowski, 1998) обсуждают некоторые математические вопросы в этой области. Разработка шахматной программы уровня чемпиона мира в течение 50 лет считалась основной задачей прикладного программирования в области компьютерных интеллектуальных игр. Известно, что Алан Тьюринг работал над алгоритмами для игры в шахматы, как и многие ведущие специалисты в области искусственного интеллекта тех лет. Ньюборн (Newborn, 1996) рассказывает об истории работы над задачей вплоть до момента за год до того, как компьютеру IBM Deep Blue наконец-то удалось победить чемпиона мира.

Построение плана относится к числу фундаментальных задач в области искусственного интеллекта; она занимает видное место в работе Рассела и Норвига (Russell, Norvig, 2002), а также является темой книги Галлаба, Нау и Траверсо (Ghallab, Nau, Traverso, 2004). Обоснование возможности решения задачи построения плана с полиномиальным пространством приводится в работе Савича (Savitch, 1970), посвященной вопросам теории сложности, а не задаче построения плана как таковой.

Примечания к упражнениям

Упражнение 1 основано на задаче, о которой мы узнали от Мэверика Ву и Райана Уильямса; упражнение 2 основано на результате Томаса Шефера.

Глава 10

Расширение пределов разрешимости

Хотя в начале книги изучались методы эффективного решения задач, в последних главах мы какое-то время занимались классами задач, для которых, как считается, эффективных решений не существует (*NP*-полными и PSPACE-полными задачами). И на основании того, что мы при этом узнали, был неявно сформирован двусторонний подход к решению новых вычислительных задач: сначала мы пытаемся найти эффективный алгоритм; а если попытка оказывается неудачной, мы пытаемся доказать его *NP*-полноту (или даже PSPACE-полноту). Если один из этих двух методов оказывается успешным, вы получаете либо решение проблемы (алгоритм), либо веское «обоснование» возникших трудностей: эта задача по крайней мере не проще многих знаменитых задач из области вычислительной теории.

К сожалению, на этой стратегии далеко не уедешь. Если существует задача, которую действительно важно решить, вряд ли кто-нибудь удовлетворится вашим заявлением о том, что задача является *NP*-сложной[1] и на нее не стоит тратить время. От вас хотят увидеть как можно лучшее решение — даже если оно и не является точным или оптимальным. Например, в задаче о независимом множестве, даже если мы не можем найти самое большое независимое множество в графе, все равно будет естественно проводить вычисления в пределах возможного времени и выдать независимое множество наибольшего размера, которое удастся найти.

Следующая часть книги будет посвящена различным аспектам этой проблемы. В главах 11 и 12 мы рассмотрим алгоритмы, предоставляющие приближенные ответы с гарантированными границами ошибки за полиномиальное время; также будут рассмотрены *эвристики локального поиска*, которые на практике часто оказываются очень эффективными, даже если мы не можем предоставить никаких доказуемых гарантий относительно их поведения.

Но для начала рассмотрим несколько ситуаций, в которых возможно точное решение экземпляров *NP*-полных задач с разумной эффективностью. Как появляются такие ситуации? Вспомните основной смысл *NP*-полноты: экземпляры этих задач в худшем случае очень сложны и, скорее всего, не решаются за полиномиаль-

[1] Под «*NP*-сложностью» мы понимаем задачу «по крайней мере не проще *NP*-полной задачи». Задачи оптимизации обычно не называют *NP*-полными, потому что формально этот термин относится только к задачам принятия решений.

ное время. Однако конкретный экземпляр задачи может и не оказаться «худшим случаем» — более того, может оказаться, что рассматриваемый экземпляр имеет особую структуру, которая упрощает задачу. Итак, в этой главе мы рассмотрим ситуации, в которых можно дать количественную оценку «простоты» данного экземпляра по сравнению с худшим случаем, чтобы использовать эти ситуации при их возникновении.

Этот принцип будет рассмотрен в нескольких конкретных ситуациях. Сначала мы займемся задачей о вершинном покрытии, для экземпляров которой существуют два естественных параметра «размера»: размер графа и размер искомого вершинного покрытия. *NP*-полнота вершинного покрытия предполагает, что решение должно быть экспоненциальным (по крайней мере) по одному из этих параметров; но разумный выбор того, какого именно, может оказать огромное влияние на время выполнения.

Затем мы исследуем идею о том, что многие *NP*-полные задачи графов становятся решаемыми за полиномиальное время, если потребовать, чтобы входные данные представляли собой дерево. Это конкретная иллюстрация того, как ввод с «особой структурой» позволяет избежать многих сложностей, из-за которых худший случай становится неразрешимым. Вооружившись этой информацией, концепцию дерева можно обобщить до более общего класса графов (имеющих малую ширину) и показать, что многие *NP*-полные задачи также разрешимы для этого более общего класса.

Впрочем, при этом следует подчеркнуть, что основное положение остается неизменным: экспоненциальные алгоритмы очень плохо масштабируются. В этой главе представлены некоторые способы обойти эту проблему, которые бывают очень эффективными в разных ситуациях, но, очевидно, для полностью обобщенного случая обходного пути не существует. Именно по этой причине мы займемся изучением алгоритмов аппроксимации и локального поиска в следующих главах.

10.1. Поиск малых вершинных покрытий

Припомним задачу о вершинном покрытии, представленную в главе 8 при изучении *NP*-полноты. Для заданного графа $G = (V, E)$ и целого k нам хотелось бы найти вершинное покрытие с размером не более k, то есть множество узлов $S \subseteq V$ с размером $|S| \leq k$, для которого по крайней мере один конец каждого ребра $e \in E$ принадлежит S.

Как и многие *NP*-полные задачи принятия решений, задача о вершинном покрытии имеет два параметра: n (количество узлов в графе) и k (допустимый размер вершинного покрытия). Это означает, что диапазон возможных границ времени выполнения становится менее тривиальным, поскольку в нем приходится учитывать взаимодействие этих двух параметров.

Задача

Рассмотрим это взаимодействие между параметрами n и k более подробно. Прежде всего замечаем, что если k является фиксированной константой (например, $k = 2$ или $k = 3$), задача о вершинном покрытии решается за полиномиальное время: мы просто проверяем все подмножества V с размером k и смотрим, образует ли какое-нибудь из них вершинное покрытие. Всего таких подмножеств $\binom{n}{k}$, и на проверку того, является ли каждое из них вершинным покрытием, требуется время $O(kn)$; общее время составляет $O(kn\binom{n}{k}) = O(kn^{k+1})$. Из этого мы видим, что неразрешимость задачи о вершинном покрытии реально проявляется только при росте k как функции n. Однако даже для умеренно малых значений k время выполнения $O(kn^{k+1})$ не особо практично. Например, если $n = 1000$ и $k = 10$, на компьютере, выполняющем миллион высокоуровневых команд в секунду, решение о том, существует ли в G вершинное покрытие из k узлов, займет 10^{24} секунд — на несколько порядков больше возраста Вселенной. И это для малого значения k, с которым задача должна быть более разрешимой! Естественно спросить себя, можно ли сделать что-то более практичное в том случае, если k является малой константой.

Оказывается, можно разработать гораздо лучший алгоритм с временем выполнения $O(2^k \cdot kn)$. Следует обратить внимание на два обстоятельства. Во-первых, подставляя $n = 1000$ и $k = 10$, мы видим, что наш компьютер сможет выполнить алгоритм всего за несколько секунд. Во-вторых, с ростом k время выполнения продолжает стремительно расти — просто экспоненциальная зависимость от k переместилась из экспоненты n в отдельную функцию. С практической точки зрения такое решение выглядит более привлекательно.

Разработка алгоритма

Прежде всего заметим, что если граф имеет малое вершинное покрытие, он не может иметь очень много ребер. Вспомните, что *степенью* узла графа называется количество инцидентных ему ребер.

(10.1) Если граф $G = (V, E)$ содержит n узлов, максимальная степень любого узла не превышает d и существует вершинное покрытие с размером не более k, то G содержит не более kd ребер.

Доказательство. Пусть S — вершинное покрытие G с размером $k' \le k$. У каждого ребра G по крайней мере один конец принадлежит S; но каждый узел в S может покрывать не более d ребер. Следовательно, в G может быть не более $k'd \le kd$ ребер. ■

Так как степень любого узла в графе не превышает $n - 1$, у (10.1) имеется простое следствие.

(10.2) Если граф $G = (V, E)$ содержит n узлов и вершинное покрытие с размером k, то G содержит не более $k(n - 1) \le kn$ ребер.

На первом шаге алгоритма мы можем проверить, содержит ли G более kn ребер; если содержит, то можно утверждать, что ответ на задачу принятия решения — существует ли вершинное покрытие с размером не более k? — отрицателен. После этой проверки будем считать, что G содержит не более kn ребер.

Идея, заложенная в основу алгоритма, концептуально очень проста. Сначала мы рассматриваем любое ребро $e = (u, v)$ в G. В любом k-узловом вершинном покрытии S графа G либо u, либо v принадлежит S. Предположим, u принадлежит вершинному покрытию S. Если удалить u и все его инцидентные ребра, остальные ребра должны покрываться не более чем $k - 1$ узлами. Иначе говоря, если $G - \{u\}$ — граф, полученный удалением u и всех инцидентных ребер, в $G - \{u\}$ должно существовать вершинное покрытие с размером не более $k - 1$. Аналогичным образом, если v принадлежит S, из этого следует, что в $G - \{v\}$ существует вершинное покрытие с размером не более $k - 1$.

Ниже приведена конкретная формулировка этой идеи.

(10.3) Пусть $e = (u, v)$ — произвольное ребро G. Граф G имеет вершинное покрытие с размером не более k в том, и только в том случае, если по крайней мере один из графов $G - \{u\}$ и $G - \{v\}$ имеет вершинное покрытие с размером не более $k - 1$.

Доказательство. Сначала предположим, что G имеет вершинное покрытие S с размером не более k. Тогда S содержит по крайней мере один из узлов u или v; предположим, это u. Тогда множество $S - \{u\}$ должно покрывать все ребра, у которых ни один из концов не равен u. Следовательно, $S - \{u\}$ является вершинным покрытием с размером не более $k - 1$ для графа $G - \{u\}$.

И наоборот, предположим, что один из графов $G - \{u\}$ и $G - \{v\}$ имеет вершинное покрытие с размером не более $k - 1$, — допустим, $G - \{u\}$ имеет такое вершинное покрытие T. Тогда множество $T \cup \{u\}$ покрывает все ребра в G, поэтому оно является вершинным покрытием для G с размером не более k. ■

Утверждение (10.3) напрямую устанавливает правильность следующего рекурсивного алгоритма для принятия решения о наличии у G вершинного покрытия из k узлов.

```
Чтобы провести поиск k-узлового вершинного покрытия для G:
   Если G не содержит ребер, то пустое множество является вершинным покрытием
   Если G содержит > k |V| edges, то G не имеет k-узлового вершинного покрытия
   Иначе для ребра e = (u, v)
      Рекурсивно проверить, имеет ли G − {u} или G − {v}
               вершинное покрытие с размером k − 1
      Если ни один подграф вершинного покрытия не имеет,
               то G не имеет k-узлового вершинного покрытия
         Иначе один из подграфов (допустим, G − {u})
               имеет вершинное покрытие T из (k − 1) узла
            В этом случае T ∪ {u} – k-узловое вершинное покрытие для G
      Конец Если
   Конец Если
```

Анализ алгоритма

Найдем границу для времени выполнения алгоритма. На интуитивном уровне поиск производится по «дереву возможностей»; рекурсивное выполнение алгоритма можно представить как рост дерева, в котором каждый узел соответствует отдельному рекурсивному вызову. Дочерними узлами, соответствующими рекурсивному вызову с параметром k, являются два узла, соответствующие рекурсивным вызовам с параметром $k - 1$. Таким образом, дерево содержит не более 2^{k+1} узлов. При каждом рекурсивном вызове затраты времени составляют $O(kn)$.

Итак, мы можем доказать следующее утверждение:

(10.4) Время выполнения алгоритма вершинного покрытия для графа из n узлов с параметром k равно $O(2^k \cdot kn)$.

Для доказательства можно воспользоваться индукцией. Если обозначить $T(n, k)$ время выполнения для графа из n узлов с параметром k, то $T(\cdot,\cdot)$ удовлетворяет следующему рекуррентному отношению для некоторой абсолютной константы c:

$$T(n, 1) \leq cn,$$

$$T(n,k) \leq 2T(n, k - 1) + ckn$$

Индукцией по $k \geq 1$ легко доказывается, что $T(n, k) \leq c \cdot 2^k kn$. В самом деле, если это утверждение истинно для $k - 1$, то

$$T(n,k) \leq 2T(n - 1,k - 1) + ckn$$

$$\leq 2c \cdot 2^{k-1}(k - 1)n + ckn$$

$$= c \cdot 2^k kn - c \cdot 2^k n + ckn$$

$$\leq c \cdot 2^k kn.$$

Таким образом, этот алгоритм значительно улучшает простой метод «грубой силы». Тем не менее ни один экспоненциальный алгоритм не может нормально масштабироваться в течение долгого времени; это относится и к нашему алгоритму. Предположим, мы хотим знать, существует ли вершинное покрытие, содержащее до 40 узлов вместо 10; на той же машине выполнение алгоритма займет много лет.

10.2. Решение *NP*-сложных задач для деревьев

В разделе 10.1 мы разработали алгоритм для задачи о вершинном покрытии, который хорошо работает при не слишком большом размере покрытия. Было показано, что найти относительно малое вершинное покрытие намного проще, чем в полностью обобщенной задаче вершинного покрытия.

В этом разделе рассматриваются особые случаи *NP*-полных задач графов другого типа — не с малым естественным параметром «размера», но со структурно

«простым» входным графом. Вероятно, простейшей разновидностью графа является дерево. Вспомните, что ненаправленный граф называется деревом, если он является связным и не содержит циклов. Дело даже не только в структурной простоте таких деревьев, но и в том, что многие *NP*-сложные задачи графов эффективно решаются в том случае, если нижележащий граф является деревом. На качественном уровне это объясняется следующим образом: если рассмотреть поддерево входных данных с корнем в некотором узле v, решение задачи, ограничиваемое этим поддеревом, «взаимодействует» с остатком дерева через v. Итак, рассматривая разные варианты расположения v в общем решении, мы фактически можем отделить задачу из поддерева v от задачи из остального дерева.

Чтобы формализовать этот общий подход и преобразовать его в эффективный алгоритм, потребуются определенные усилия. Сейчас вы увидите, как это делается для разновидностей задачи о независимом множестве; однако важно помнить, что этот принцип достаточно общий, и с таким же успехом можно было рассмотреть для деревьев многие другие *NP*-полные задачи графов.

Сначала мы покажем, что сама задача о независимом множестве для дерева может быть решена жадным алгоритмом. Затем будет рассмотрено обобщение — *задача о независимом множестве с максимальным весом*, в котором узлам назначаются веса, а ищется независимое множество с максимальным весом. Вы увидите, что задача о независимом множестве с максимальным весом решается для деревьев средствами динамического программирования с использованием довольно прямолинейной реализации описанных выше рассуждений.

Жадный алгоритм для задачи о независимом множестве для деревьев

Наше рассмотрение жадного алгоритма для дерева начнется с анализа того, как решение выглядит с точки зрения одного ребра (разновидность идеи из раздела 10.1). Возьмем ребро $e = (u, v)$ в графе G. В любом независимом множестве S в G не более чем один из узлов u или v может принадлежать S. Нам хотелось бы найти ребро e, для которого жадный алгоритм мог бы решить, какой из двух концов следует включить в независимое множество.

Для этого мы воспользуемся важнейшим свойством деревьев: у каждого дерева должен быть как минимум один *лист* — узел со степенью 1. Возьмем лист v; пусть (u, v) — уникальное ребро, инцидентное v. Как «жадно» оценить относительные преимущества включения u или v в независимое множество S? Если включить v, то из всех остальных узлов только u напрямую «блокируется» от присоединения к независимому множеству. Если включить u, то блокируется не только v, но и все остальные узлы, присоединенные к u. Итак, если мы стремимся к максимизации размера независимого множества, похоже, включение v должно быть лучше (или по крайней мере не хуже) включения u.

(10.5) Если $T = (V, E)$ — дерево, а v — лист этого дерева, то существует независимое множество максимального размера, содержащее v.

Доказательство. Рассмотрим независимое множество максимального размера S. Пусть $e = (u, v)$ — уникальное ребро, инцидентное узлу v. Очевидно, по

крайней мере один из узлов u или v принадлежит S; в противном случае мы могли бы добавить v в S с увеличением размера. Теперь, если $v \in S$, то работа закончена; а если $u \in S$, то мы можем получить другое независимое множество S' того же размера, удалив u из S и вставив v. ■

Мы можем использовать (10.5) многократно для выявления и удаления узлов, которые могут быть включены в независимое множество. При удалении дерево T может потерять связность. Чтобы избежать лишних проблем, мы опишем алгоритм для более общего случая, при котором используемый граф представляет собой *лес* — граф, в котором каждая компонента связности представляет собой дерево. Задачу нахождения независимого множества максимального размера для леса можно рассматривать как соответствующую задаче для деревьев: оптимальное решение для леса представляет собой простое объединение оптимальных решений для каждого дерева, и при этом можно использовать (10.5) для анализа задачи в каждом дереве.

Допустим, имеется лес F; тогда (10.5) позволяет принять первое решение по следующему жадному правилу. Снова рассмотрим ребро $e = (u, v)$, где v является листом. В независимое множество S включается узел v и не включается узел u. С этим решением мы можем удалить узел v (так как он уже включен) и узел u (так как он не может быть включен) с получением меньшего леса. Далее выполнение рекурсивно продолжается для уменьшенного леса для получения решения.

```
Чтобы найти независимое множество максимального размера для леса F:
   Пусть S — создаваемое независимое множество (изначально пусто)
   Пока F содержит хотя бы одно ребро
      Пусть e = (u, v) — ребро F, в котором v является листом
      Добавить v в S.
      Удалить из F узлы u и v, а также все инцидентные им ребра.
   Конец Пока
Вернуть S
```

(10.6) Приведенный алгоритм находит независимое множество максимального размера для лесов (а следовательно, и для деревьев).

Хотя утверждение (10.5) кажется очень простым, в действительности оно представляет применение одного из принципов разработки жадных алгоритмов, приведенных в главе 4: *метода замены*. В частности, в нашем алгоритме независимого множества центральное место занимает тот факт, что любое решение, не содержащее конкретный лист, может быть «преобразовано» в не худшее решение, содержащее этот лист.

Чтобы реализовать этот алгоритм для быстрого выполнения, необходимо хранить текущий лес F так, чтобы способ хранения обеспечивал эффективное нахождение ребра, инцидентного листу. Реализовать этот алгоритм с линейным временем несложно: лес должен храниться так, чтобы это можно было сделать за одну итерацию цикла *Пока* за время, пропорциональное количеству удаляемых ребер при исключении u и v.

Жадный алгоритм для более общих графов

Работоспособность жадного алгоритма, приведенного выше, для общих графов не гарантирована, потому что не гарантируется нахождение листа при каждой

итерации. Однако утверждение (10.5) относится к любому графу: если имеется произвольный граф G с ребром (u, v), в котором u является единственным соседом v, то всегда безопасно поместить v в независимое множество, удалить u и v и повторить с уменьшенным графом.

Итак, если многократное удаление узлов со степенью 1 и их соседей позволит исключить весь граф, мы гарантированно находим независимое множество с максимальным размером — даже если исходный граф не был деревом. А если исключить весь граф не удастся, возможно, мы последовательно выполним несколько итераций алгоритма, отчего размер графа уменьшится, а другие методы станут более приемлемыми. Таким образом, жадный алгоритм является полезной эвристикой, которую можно с пользой применить к произвольному графу для продвижения к цели нахождения большого независимого множества.

Независимое множество с максимальным весом для деревьев

Теперь обратимся к более сложной задаче — нахождению независимого множества с максимальным весом. Как и прежде, предполагается, что граф представляет собой дерево $T = (V, E)$, но теперь с каждым узлом связывается положительный *вес* w_v. Задача нахождения независимого множества с максимальным весом заключается в нахождении в графе $T = (V, E)$ такого независимого множества S, что общий вес $\sum_{v \in S} w_v$.

Сначала мы опробуем идею, которая использовалась ранее для построения жадного решения в случае без весов. Рассмотрим ребро $e = (u, v)$, для которого v является листом. Включение v блокирует вхождение в независимое множество меньшего числа узлов; следовательно, если вес v хотя бы не меньше веса u, мы можем принять жадное решение, как это делалось в случае без весов. Однако если $w_v < w_u$, мы сталкиваемся с дилеммой: включение u обеспечивает больший вес, но включение v оставляет больше вариантов на будущее. Похоже, простого способа принять решение локально, без учета оставшейся части графа, не существует. Впрочем, кое-что сказать все же можно. Если узел u имеет много соседей $v_1, v_2, \ldots$, которые являются листьями, это же решение следует принять для всех: принимая решение о том, что u не включается в независимое множество, мы с таким же успехом можем включить все соседние листья. Таким образом, для поддерева, состоящего из u и соседних листьев, стоит рассматривать только два «разумных» решения: включать u или включать все листья.

Эти идеи будут использоваться для разработки алгоритма с полиномиальным временем средствами динамического программирования. Как вы помните, динамическое программирование позволяет сформировать несколько разных решений, которые строятся через последовательность подзадач, и только в конце решить, какие из этих возможностей будут использоваться в общем решении.

Первое решение, которое следует принять для алгоритма динамического программирования, — выбор подзадач. Для независимого множества с макси-

мальным весом подзадачи будут строиться размещением корня дерева T в произвольном узле r; вспомните, что это операция «ориентирует» все ребра дерева в направлении от r. А именно: для каждого узла $u \neq r$ родителем $p(u)$ узла u является узел, смежный с u на пути от корня r. Другими соседями u становятся его дочерние узлы; для обозначения множества дочерних узлов u будет использоваться запись $children(u)$. Узел u и все его потомки образуют поддерево T_u, корнем которого является u.

Наши подзадачи будут базироваться на этих поддеревьях T_u. Дерево T_r соответствует исходной задаче. Если $u \neq r$ является листом, то T_u состоит из одного узла. Для узла u, все дочерние узлы которого являются листьями, T_u является как раз таким поддеревом, о котором говорилось выше.

Чтобы решить эту проблему средствами динамического программирования, мы начнем с листьев и постепенно будем двигаться вверх по дереву. Для узла u подзадача, связанная с деревом T_u, решается после решения подзадач всех его дочерних узлов. Чтобы получить независимое множество с максимальным весом S для дерева T_u, рассмотрим два случая: узел u либо включается в S, либо не включается. Если узел u включается, то мы не сможем включить его дочерние узлы; если же u не включается, то дочерние узлы могут включаться, а могут и не включаться. Это наводит на мысль о том, что для каждого поддерева T_u должны определяться две подзадачи: подзадача $\mathtt{OPT}_{in}(u)$ для максимального веса независимого множества T_u, включающего u, и подзадача $\mathtt{OPT}_{out}(u)$ для максимального веса независимого множества T_u, не включающего u.

После того как подзадачи будут выбраны, легко понять, как происходит рекурсивное вычисление этих значений. Для листа $u \neq r$ имеем $\mathtt{OPT}_{out}(u) = 0$ и $\mathtt{OPT}_{in}(u) = w_u$. Для всех остальных узлов u действует следующее рекуррентное отношение, определяющее $\mathtt{OPT}_{out}(u)$ и $\mathtt{OPT}_{in}(u)$ с использованием значений дочерних узлов u.

(10.7) Для узла u, имеющего дочерние узлы, следующее рекуррентное отношение определяет значения подзадач:

- $$\mathrm{OPT}_{in}(u) = w_u + \sum_{v \in children(u)} \mathrm{OPT}_{out}(v);$$
- $$\mathrm{OPT}_{out}(u) = \sum_{v \in children(u)} \max\left(\mathrm{OPT}_{out}(v), \mathrm{OPT}_{in}(v)\right).$$

Используя это рекуррентное отношение, мы получаем алгоритм динамического программирования, основанный на построении оптимальных решений по поддеревьям все большего и большего размера. Массивы $M_{out}[u]$ и $M_{in}[u]$ используются для хранения значений $\mathtt{OPT}_{out}(u)$ и $OPT_{in}(u)$ соответственно. Чтобы построить решения, необходимо обработать все дочерние узлы некоторого узла, прежде чем обрабатывать сам узел; в терминологии обхода дерева узлы посещаются в *обратном порядке* (post-order).

```
Чтобы найти независимое множество с максимальным весом для дерева T:
   Расположить корень дерева в узле r
   Для всех узлов u дерева T с обходом в обратном порядке
      Если u является листом, то присвоить значения:
    M_out[u] = 0
```

$M_{in}[u] = w_u$
Иначе присвоить значения:

$$M_{out}[u] = \sum_{v \in children(u)} \max\left(M_{out}[u], M_{in}[u]\right)$$

$$M_{in}[u] = w_u \sum_{v \in children(u)} M_{out}[u].$$

Конец Если
Конец цикла
Вернуть max($M_{out}[r]$, $M_{in}[r]$)

Так мы получаем *значение* независимого множества с максимальным весом. Теперь, по стандартной практике алгоритмов динамического программирования, которая уже встречалась ранее, можно легко восстановить само независимое множество с максимальным весом; для этого мы сохраняем решение, принятое для каждого узла, а потом посредством обратного отслеживания по этим решениям определяем, какие узлы следует включить в множество.

(10.8) Приведенный выше алгоритм находит независимое множество с максимальным весом для деревьев за линейное время.

10.3. Раскраска множества дуг

Несколько лет назад, когда телекоммуникационные компании начали интенсивно заниматься *технологией спектрального мультиплексирования*, у исследователей в этих компаниях проявился глубокий интерес к алгоритмическому вопросу, который ранее почти не изучался: *задаче о раскраске множества дуг*.

Сначала мы объясним, как проявилась связь между этими темами, а потом разработаем алгоритм для этой задачи. Алгоритм представляет собой более сложную вариацию на тему раздела 10.2: вычислительно сложная задача решается методами динамического программирования, с построением решений на множестве подзадач, «взаимодействующих» друг с другом по очень малым фрагментам входных данных. Ограничение таких взаимодействий позволяет удержать под контролем сложность алгоритма.

Задача

Сначала в двух словах о том, как проблемы сетевой маршрутизации привели к вопросу раскраски дуг. Технология спектрального мультиплексирования позволяет нескольким коммуникационным потокам совместно использовать одну часть оптоволоконного кабеля при условии, что потоки передаются по кабелю на разных длинах волн. Смоделируем коммуникационную сеть в виде графа $G = (V, E)$; каждый *коммуникационный поток* представляется путем P_i в G; считается, что данные передаются по потоку от одного конца P_i на другой конец. Если пути P_i

и P_j имеют общее ребро в G, одновременная передача данных по этим потокам возможна только на разных длинах волн. Итак, наша цель заключается в следующем: для заданного множества из k длин волн (1, 2, ..., k) каждому потоку P_i требуется назначить длину волны так, чтобы каждой паре потоков, имеющих общее ребро в графе, были назначены разные длины волн. Назовем эту задачу экземпляром *задачи о раскраске путей*, а ее решение — действительное распределение длин волн между путями — *k-раскраской*.

Это вполне естественная задача, которую можно рассматривать прямо в таком виде; но с точки зрения контекста оптоволоконной маршрутизации будет полезно внести одно упрощение. На практике спектральное мультиплексирование часто работает в сетях G, имеющих чрезвычайно простую структуру, поэтому будет естественно ограничить экземпляры задачи о раскраске путей за счет некоторых предположений относительно структуры сети. Один из самых важных частных случаев на практике также оказывается одним из простейших: используемая сеть имеет кольцевидную структуру, то есть моделируется графом G, который представляет собой цикл из n узлов.

Именно этим случаем мы сейчас займемся: имеется граф $G = (V, E)$, который представляет собой цикл из n узлов, и имеется множество путей P_1, ..., P_m в этом цикле. Целью, как и прежде, является назначение одной из k заданных длин волн каждому пути P_i так, чтобы перекрывающимся путям назначались разные длины волн. Мы будем называть такое назначение *действительным* распределением длин волн. На рис. 10.1 изображен примерный экземпляр такой задачи. В этом экземпляре существует действительное распределение с $k = 3$ длинами волн: длина волны 1 назначается путям a и e, длина волны 2 — путям b и f, длина волны 3 — путям c и d. Из рисунка видно, что циклическая сеть может рассматриваться как круг, а пути — как дуги на этом круге; соответственно этот частный случай задачи о раскраске путей будет называться задачей о раскраске дуг.

Сложность задачи о раскраске дуг

Нетрудно заметить, что задача о раскраске дуг напрямую сводится к задаче о раскраске графа. Для заданного экземпляра задачи раскраски дуг мы определим граф, содержащий узел z_i для каждого пути P_i, и соединим узлы z_i и z_j в H, если у путей P_i и P_j существует общее ребро в G. Тогда маршрутизация всех потоков с использованием k длин волн представляет собой задачу о раскраске H с использованием не более k цветов. (На самом деле эта задача представляет собой очередное практическое применение раскраски графа, в которой абстрактные «цвета» действительно являются цветами, так как они представляют световые волны разной длины.)

Из этого не следует, что задача о раскраске дуг является *NP*-полной, — мы всего лишь свели ее к известной *NP*-полной задаче, что ничего не говорит о ее сложности. Для задачи о раскраске путей на обобщенных графах легко выполнить сведение задачи о раскраске графа к раскраске путей, тем самым устанавливая *NP*-полноту задачи о раскраске путей. Однако это прямолинейное сведение не работает, когда используемый граф представляет собой простой цикл. Итак, что же можно сказать о сложности задачи о раскраске дуг?

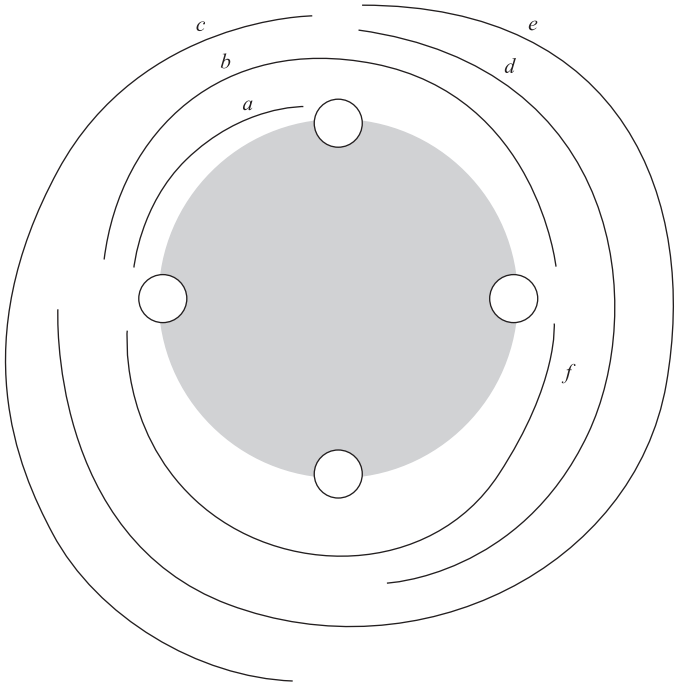


Рис. 10.1. Экземпляр задачи о раскраске дуг с шестью дугами (a, b, c, d, e, f) для цикла из четырех узлов

Как выясняется, *NP*-полнота задачи о раскраске дуг доказывается с использованием очень сложного сведения. Это огорчит людей, работающих с оптоволоконными сетями, потому что означает, что задача оптимального распределения длин волн вряд ли имеет эффективное решение. Однако все известные сведения, демонстрирующие *NP*-полноту задачи о раскраске дуг, обладают одним интересным свойством: сложные экземпляры задачи о раскраске дуг, создаваемые ими, всегда имеют достаточно большое множество длин волн. Итак, эти сведения не доказывают, что задача о раскраске дуг сложна при малом количестве длин волн; они оставляют возможность того, что для любого фиксированного, постоянного количества длин волн k задача распределения длин волн может быть решена за время, полиномиальное по n (размер цикла) и m (количество путей). Другими словами, мы можем надеяться на время выполнения в форме, приведенной для задачи о вершинном покрытии в разделе 10.1: $O(f(k) \cdot p(n, m))$, где $f(\cdot)$ может быть быстрорастущей функцией, но $p(\cdot,\cdot)$ имеет полиномиальную скорость роста.

Такое время выполнения выглядит заманчиво (если $f(\cdot)$ не растет с совсем запредельной скоростью), так как оно делает распределение длин волн потенциально приемлемым при малом их количестве. Чтобы понять, насколько непросто добиться такого времени выполнения, приведем аналогию: общая задача о раскраске графа сложна уже для трех цветов. Если бы задача о раскраске дуг была разрешима для любого фиксированного количества длин волн (то есть цветов) k,

это означало бы, что она является частным случаем раскраски графа с качественно иной сложностью.

В этом разделе мы постараемся создать алгоритм с временем выполнения такого типа, $O(f(k) \cdot p(n, m))$. Как упоминается в начале раздела, сам алгоритм строится на принципах, приводившихся в разделе 10.2 при решении задачи о независимом множестве с максимальным весом для деревьев. Сложность поиска, присущая поиску независимого множества с максимальным весом, была сокращена тем фактом, что для каждого узла v в дереве T задачи в компонентах $T - \{v\}$ становились полностью изолированными, когда мы принимали решение о включении (или невключении) v в независимое множество. Это конкретный пример общего принципа фиксирования малого множества решений, а следовательно, разделения задачи на меньшие подзадачи, которые могут решаться независимо.

В данном случае аналогичная идея заключается в выборе конкретной точки цикла и решении о том, как раскрашивать дуги, проходящие через эту точку; закрепление этих степеней свободы позволяет определять серии меньших подзадач для оставшихся дуг.

Разработка алгоритма

Для начала определимся с обозначениями. Имеется граф G, который представляет собой цикл из n узлов; узлы обозначаются $v_1, v_2, ..., v_n$, для каждого i существует ребро (v_i, v_{i+1}), а также существует ребро (v_n, v_1). Существует множество путей $P_1, P_2, ..., P_m$ в графе G и множество из k доступных цветов; пути нужно раскрасить так, чтобы путям P_i и P_j, имеющим общее ребро, назначались разные цвета.

Простой частный случай: раскраска интервалов

Чтобы построить алгоритм для раскраски дуг, мы сначала рассмотрим более простую задачу о раскраске интервалов на линии. Ее можно рассматривать как частный случай задачи о раскраске дуг, в котором дуги лежат только в одном полушарии; стоит избавиться от сложностей, обусловленных «возвратом к началу», как задача намного упрощается. Итак, в этом частном случае имеется множество интервалов, которые необходимо пометить так, чтобы каждые два перекрывающихся интервала получали разные метки.

Именно такая задача уже встречалась ранее: это задача интервального планирования, оптимальный жадный алгоритм для которой приводился в разделе 4.1. Анализ в этом разделе не только показывает, что для раскраски интервалов существует эффективный оптимальный алгоритм, но и дает много полезной информации о структуре задачи. А именно: если определить *глубину* множества интервалов как максимальное число интервалов, проходящих через одну точку, то жадный алгоритм из главы 4 показал бы, что минимальное количество необходимых цветов всегда равно глубине. Обратите внимание: количество необходимых цветов очевидно не менее глубины, потому что интервалы, содержащие общую точку, должны быть окрашены в разные цвета; здесь принципиально то, что необходимое количество цветов не может быть больше глубины.

Интересно, что точное отношение между количеством цветов и глубиной не выполняется для наборов дуг в круге. На рис. 10.2, например, изображен набор дуг с глубиной 2, для которого необходимы три цвета. Это происходит из-за того, что при попытке раскраски набора дуг возникают «отдаленные» препятствия, которые существенно усложняют задачу по сравнению с раскраской интервалов на линии. Несмотря на это, анализ упрощенной задачи раскраски интервалов будет полезен для разработки алгоритма раскраски дуг.

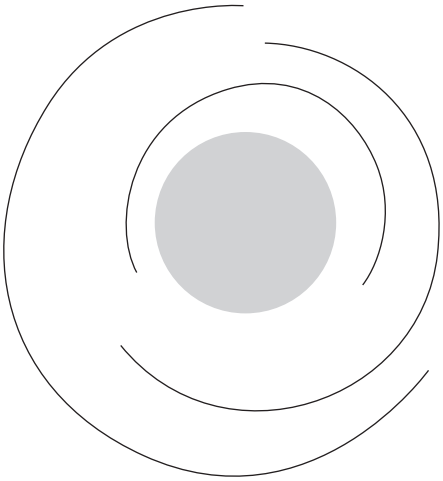


Рис. 10.2. Набор дуг, для раскраски которого необходимы три цвета, хотя через любую точку окружности проходят не более двух дуг

А теперь вернемся к задаче о раскраске дуг. Для начала рассмотрим частный случай, при котором для каждого ребра e в цикле существуют ровно k путей, содержащих e. Назовем эту задачу частным случаем *с постоянной глубиной*. Этот частный случай может показаться ограниченным, но, по сути, в нем отражена вся сложность задачи; и когда мы получим алгоритм для случая с постоянной глубиной, он легко преобразуется в алгоритм решения обобщенной задачи.

Разработка алгоритма начинается с преобразования экземпляра задачи в измененную форму задачи раскраски интервалов: цикл «разрезается» по ребру (v_n, v_1) и «разворачивается» в путь G'. Этот процесс изображен на рис. 10.3. «Разрезанный и развернутый» граф G' содержит те же узлы, что и G, а также два лишних узла в точке разреза: узел v_0, смежный с v_1 (и никакими другими узлами), и узел v_{n+1}, смежный с v_n (и никакими другими узлами). Кроме того, множество путей также немного изменилось. Допустим, $P_1, P_2, ..., P_k$ — пути, содержащие ребро (v_n, v_1) в G. Каждый из этих путей P_i разделяется на два: P'_i (от v_0) и P''_i (до v_{n+1}).

Перед нами экземпляр задачи о раскраске интервалов с глубиной k. Следовательно, из приведенного выше упоминания связи между глубиной и цветами видно, что интервалы

$$P'_1, P'_2, \ldots, P'_k, P_{k+1}, \ldots, P_m, P''_1, P''_2, \ldots, P''_k$$

могут быть окрашены с использованием k цветов. И что же, работа закончена? Можно ли преобразовать это решение в решение для путей графа G?

Оказывается, это не так просто; проблема в том, что при раскраске интервалов путям P'_i и P''_i могут быть назначены разные цвета. Так как это два фрагмента одного пути P_i в G, непонятно, как взять разные цвета P'_i и P''_i и определить по ним цвет P_i в G. Например, при разрезании цикла на рис. 10.3, *a* будет получено множество интервалов, изображенное на рис. 10.3, *b*. Предположим, была вычислена такая раскраска, что интервалам в первой строке присваивается цвет 1, интервалам во второй строке — цвет 2, а интервалам в третьей строке — цвет 3. Тогда очевидного способа определить цвет для a и c не существует.

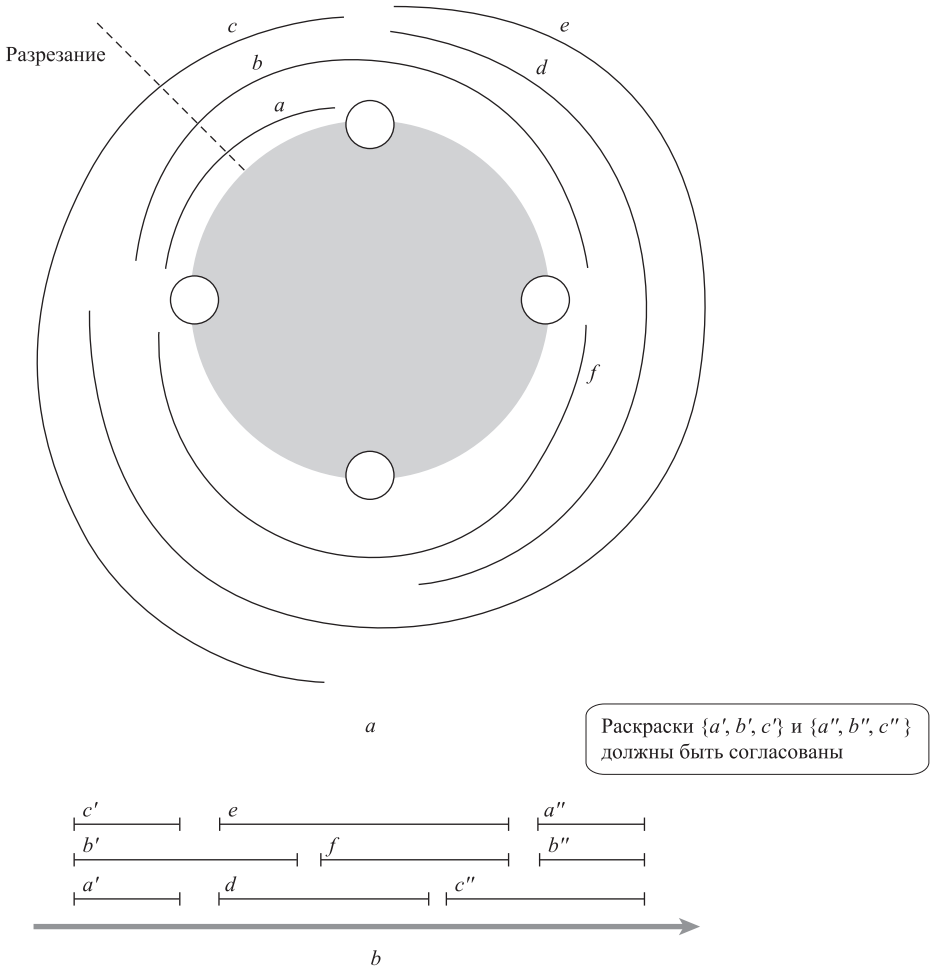


Рис. 10.3. Разрезание цикла (*a*) в экземпляре раскраски дуг и его преобразование в (*b*) набор интервалов на линии

Это подсказывает способ формализации отношений между экземпляром раскраски дуг в G и экземпляром раскраски интервалов в G'.

(10.9) Пути в G имеют k-раскраску в том, и только в том случае, если пути в G' имеют k-раскраску с тем дополнительным ограничением, что P'_i и P''_i назначается одинаковый цвет для всех $i = 1, 2, ..., k$.

Доказательство. Если пути в G могут быть раскрашены в k цветов, то мы просто используем k-раскраску как цвета G', назначая каждому из путей P'_i и P''_i цвет P_i. В полученной раскраске никакие два пути не имеют общего ребра.

И наоборот, предположим, пути в G' могут иметь k-раскраску с тем дополнительным ограничением, что P'_i и P''_i назначается один цвет для всех $i = 1, 2, ..., k$. Тогда пути P_i (для $i \le k$) назначается общий цвет P'_i и P''_i, а пути P_j (для $j > k$) назначается цвет, который P_j получает в G'. И снова при такой раскраске никакие два пути, имеющие общий цвет, не имеют общих ребер. ■

Теперь мы успешно преобразовали задачу в поиск раскраски путей G', подчиняющейся условию из (10.9): путям P'_i и P''_i ($1 \le i \le k$) назначается один цвет.

Прежде чем двигаться дальше, мы введем дополнительную терминологию, которая упрощает обсуждение алгоритмов в этой задаче. Во-первых, так как названия цветов выбираются произвольно, можно считать, что пути P'_i назначен цвет i для каждого $i = 1, 2, ..., k$. Теперь для каждого ребра $e_i = (v_i, v_{i+1})$ мы обозначаем S_i множество путей, содержащих это ребро. K-раскраска только для путей S_i имеет очень простую структуру: это просто способ назначения ровно одного из цветов $\{1, 2, ..., k\}$ каждому из k путей в S_i. Мы будем рассматривать такую k-раскраску как взаимно однозначную функцию $f: S_i \to \{1, 2, ..., k\}$.

А теперь важное определение: k-раскраска f для S_i и k-раскраска g для S_j называются *согласованными*, если существует одна k-раскраска всех путей, которая равна f для S_i, а также равна g для S_j. Иначе говоря, k-раскраски f и g для ограниченных частей экземпляра могут быть получены из одной k-раскраски всего экземпляра. В контексте согласованности наша задача формулируется следующим образом: если обозначить f' k-раскраску S_0, которая назначает цвет i пути P'_i, а f'' — k-раскраску S_n, которая назначает цвет i пути P''_i, требуется решить, являются ли согласованными f' и f''.

Поиск допустимой раскраски интервалов

Неясно, как принять решение о согласованности f' и f'' напрямую. Вместо этого мы воспользуемся методом динамического программирования и построим решение на основе серии подзадач.

Подзадачи выглядят так: для каждого множества S_i, последовательно обрабатывая $i = 0, 1, 2, ..., n$, мы вычисляем множество F_i для всех k-раскрасок S_i, согласованных с f'. После вычисления множества F_n остается только проверить, содержит ли оно f'', чтобы ответить на общий вопрос о согласованности f' и f''.

В начале выполнения алгоритма $F_0 = \{f'\}$: так как f' определяет цвет для каждого интервала в S_0, очевидно, что никакая k-раскраска S_0 не может быть согласована

с ней. Теперь допустим, что мы вычислили F_0, F_1, ..., F_i, и покажем, как вычислить F_{i+1} по F_i.

Вспомните, что S_i состоит из путей, содержащих ребро $e_i = (v_i, v_{i+1})$, а S_{i+1} состоит из путей, содержащих следующее последовательное ребро $e_{i+1} = (v_{i+1}, v_{i+2})$. Пути в S_i и S_{i+1} можно разделить на три типа:

- содержащие e_i и e_{i+1}; лежат в S_i и S_{i+1};
- заканчивающиеся в узле v_{i+1}; лежат в S_i, но не в S_{i+1};
- начинающиеся в узле v_{i+1}; лежат в S_{i+1}, но не в S_i.

Для любой раскраски $f \in F_i$ раскраска g множества S_{i+1} называется *расширением f*, если все пути в $S_i \cap S_{i+1}$ имеют одни и те же цвета в отношении f и g. Легко проверить, что если g является расширением f, а раскраска f согласована с f', то согласована и раскраска g. С другой стороны, предположим, некоторая раскраска g множества S_{i+1} согласована с f'; иначе говоря, существует раскраска h всех путей, равная f' для S_0, и равная g для S_{i+1}. Тогда, если рассмотреть цвета, назначаемые h путям в S_i, мы получаем раскраску $f \in F_i$ и g является расширением f.

Это доказывает следующий факт.

(10.10) Множество F_{i+1} равно множеству всех расширений k-раскрасок в F_i.

Итак, чтобы вычислить F_{i+1}, достаточно просто составить список всех расширений всех раскрасок в F_i. Для всех $f \in F_i$ это означает, что нам нужен список всех раскрасок g для S_{i+1}, согласующихся с f для $S_i \cap S_{i+1}$. Для этого мы перечисляем все возможные способы назначения цветов $S_i - S_{i+1}$ (в отношении f) путям $S_{i+1} - S_i$. Слияние этих списков для всех $f \in F_i$ дает F_{i+1}.

Итак, общий алгоритм выглядит так:

```
Чтобы определить, являются ли f′ и f″ согласованными:
   Определить F_0 = {f′}
   Для i = 1, 2, ..., n
      Для всех f ∈ F_i
         Добавить все расширения f в F_{i+1}
      Конец цикла
   Конец цикла
   Проверить, входит ли f″ в F_n
```

На рис. 10.4 представлены результаты выполнения этого алгоритма для примера на рис. 10.3. Как и во всех алгоритмах динамического программирования, которые встречались в этой книге, сама раскраска вычисляется обратным отслеживанием по этапам построения множеств $F_1, F_2, \ldots, F_n$.

Вскоре мы проанализируем время выполнения этого алгоритма. Но сначала посмотрим, как снять предположение о том, что входной экземпляр имеет постоянную глубину.

Вспомните, что только что разработанный нами алгоритм предполагает, что каждое ребро e содержится ровно в k путях. В общем случае каждое ребро может быть задействовано в разном количестве путей, вплоть до максимума k. (Если бы было ребро, содержащееся в k+1 путях, то все эти пути должны были быть окрашены в разные цвета, поэтому нам пришлось бы немедленно заключить, что входной экземпляр не окрашивается в k цветов.)

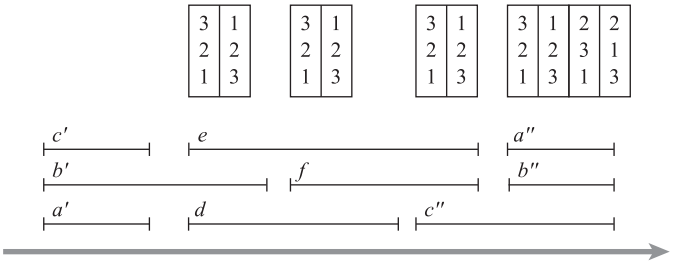


Рис. 10.4. Выполнение алгоритма раскрашивания. Исходная раскраска f' присваивает цвет 1 интервалу a', цвет 2 — интервалу b' и цвет 3 — интервалу c'. Над каждым ребром e_i (для $i > 0$) расположена таблица, представляющая множество всех согласованных раскрасок в Fi: каждая раскраска представляется одним из столбцов таблицы. Так как раскраска $f''(a'') = 1$, $f''(b'') = 2$ и $f''(c'') = 3$ присутствует в последней таблице, у экземпляра существует решение

Алгоритм нетрудно изменить напрямую для общего случая, но так же просто свести общий случай к случаю с постоянной глубиной. Для каждого ребра e_i, несущего только $k_i < k$ путей, добавляются $k - k_i$ путей, состоящих из единственного ребра e_i. Теперь мы имеем экземпляр с постоянной глубиной и можем доказать следующее утверждение:

(10.11) Исходный экземпляр может быть раскрашен k цветами в том, и только в том случае, если измененный экземпляр (полученный добавлением однореберных путей) может быть раскрашен k цветами.

Доказательство. Очевидно, если измененный экземпляр имеет k-раскраску, то эта же k-раскраска может быть применена к исходному экземпляру (цвета, назначенные только что добавленным однореберным путям, просто игнорируются). И наоборот, предположим, что исходный экземпляр имеет k-раскраску f. Тогда мы можем построить k-раскраску измененного экземпляра, начиная с f и последовательно рассматривая дополнительные однореберные пути, назначая любой свободный цвет каждому из этих путей при их рассмотрении. ■

Анализ алгоритма

Наконец, найдем границу для времени выполнения алгоритма. В нем доминирует время вычисления множеств F_1, F_2, ..., F_n. Чтобы построить одно из таких множеств F_{i+1}, нам понадобится рассмотреть каждую раскраску $f \in F_i$ и перечислить все перестановки цветов, которые f присваивает путям в $S_i - S_{i+1}$. Так как S_i содержит k путей, то количество раскрасок в F_i не превышает $k!$. Перечисление всех перестановок цветов, которые f назначает $S_i - S_{i+1}$, также включает перебор по множеству размера $\ell!$, где $\ell \le k$ — размер $S_i - S_{i+1}$.

Следовательно, общее время вычисления F_{i+1} от F_i записывается в форме $O(f(k))$ для функции $f(\cdot)$, зависящей только от k. По n итерациям внешнего цикла для вычисления F_1, F_2, ..., F_n получаем общее время выполнения $O(f(k) \cdot n)$, как и требовалось.

На этом завершается описание и анализ алгоритма. Основные его свойства перечислены в следующем утверждении.

(10.12) Алгоритм, описанный в этом разделе, правильно определяет возможность раскраски набора путей в n-узловом в k цветов, а его время выполнения составляет $O(f(k) \cdot n)$ для функции $f(\cdot)$, зависящей только от k.

Оглядываясь назад, мы видим, что время выполнения алгоритма обусловлено фактом, упомянутым в начале раздела: для всех i подзадачи, основанные на вычислении F_i и F_{i+1}, совместно «проходят» через «узкий интерфейс», состоящий из путей в S_i и S_{i+1}, размер каждого из которых не превышает k. Таким образом, мы можем сделать так, чтобы время, необходимое для перехода между ними, зависело только от k, а не от размера цикла G или от количества путей.

10.4.* Декомпозиция графа в дерево

В предыдущих двух разделах мы видели, как конкретные *NP*-сложные задачи (а именно задачи о независимом множестве с максимальным весом и раскраске графа) могут решаться при ограниченной структуре входных данных. Когда вы оказываетесь в такой ситуации — возможности решения *NP*-полной задачи в более или менее естественном частном случае, — стоит спросить себя, почему этот метод не работает в общем случае. Как упоминалось в разделах 10.2 и 10.3, наши алгоритмы в обоих случаях использовали особенности конкретной разновидности структуры: тот факт, что входные данные можно было разбить на подзадачи с очень ограниченным взаимодействием.

Например, чтобы решить задачу о независимом множестве с максимальным весом для дерева, мы используем специальное свойство (корневых) деревьев: после принятия решения о том, должен ли узел u включаться в независимое множество или нет, подзадачи в каждом поддереве становятся полностью изолированными; мы можем решать каждую из них так, словно остальных не существует. В обобщенных графах, где может не быть узла, «разрывающего связь» между подзадачами в других частях графа, такая удобная возможность отсутствует. В задаче о независимом множестве для обобщенного графа решения, принятые в одном месте, приводят к сложным вторичным эффектам по всему графу.

Итак, мы можем задать себе «ослабленную» версию нашего вопроса: насколько общим должен быть класс графов, чтобы для него можно было использовать концепцию «ограниченного взаимодействия» — рекурсивной фрагментации входных данных с использованием малых множеств узлов — для проектирования эффективных алгоритмов для таких задач, как задача о независимом множестве с максимальным весом?

Как выясняется, существует естественный и обладающий широкими возможностями класс графов, поддерживающих алгоритмы такого типа; фактически такие графы представляют собой «обобщенные деревья», и по причинам, которые вскоре станут ясны, мы будем называть их *графами с ограниченной древовидной шириной*. Как и для многих деревьев, многие *NP*-полные задачи разрешимы для графов

с ограниченной древовидной шириной; при этом класс графов с ограниченной древовидной шириной имеет существенное практическое значение, потому что к нему относятся многие реальные сети, в которых возникают *NP*-полные задачи графов. Итак, в каком-то смысле этот тип графов служит примером нахождения «правильного» частного случая задачи, который одновременно позволяет построить эффективный алгоритм и включает графы, встречающиеся на практике.

В этом разделе мы определим понятие древовидной ширины, а также предоставим общий подход к решению задач на графах с ограниченной древовидной шириной. В следующем разделе речь пойдет о том, как определить, имеет ли заданный граф ограниченную древовидную ширину.

Определение древовидной ширины

Дадим точное определение класса графов, предназначенного для обобщенного представления деревьев. В основу определения заложены два соображения. Во-первых, нам нужны графы, которые можно разложить на несвязанные части удалением небольшого количества узлов; это позволит реализовать динамические алгоритмы того типа, о котором говорилось ранее. Во-вторых, мы хотим формализовать интуитивное представление о «древовидных» изображениях графов наподобие изображенного на рис. 10.5, *b*.

Нам хотелось бы утверждать, что граф *G*, изображенный на рисунке, может быть разложен в древовидный формат по упомянутым выше соображениям. Если взглянуть на граф *G* в том виде, в каком он изображен на рис. 10.5, вероятно, не будет очевидно, что это возможно. Но из рис. 10.5, *b* видно, что *G* на самом деле состоит из десяти переплетенных треугольников; и семь из десяти треугольников обладают тем свойством, что при их удалении остаток *G* распадется на несвязанные части, рекурсивно обладающие той же структурой из сцепленных треугольников. Еще три треугольника прикреплены в крайних точках, и их удаление можно сравнить с удалением листовых узлов из дерева.

Итак, граф *G* древовиден, если рассматривать его не как состоящий из двенадцати узлов, как обычно, а как состоящий из десяти треугольников. И хотя *G* очевидным образом содержит много циклов, при рассмотрении на уровне этих десяти треугольников циклов словно бы и нет; на этом основании граф наследует многие полезные декомпозиционные свойства дерева.

Древовидная структура из треугольников должна представляться так, чтобы каждый треугольник соответствовал узлу дерева, как показано на рис. 10.5, *c*. Интуитивно понятно, что дерево на рисунке соответствует графу (каждый узел дерева представляет один из треугольников). Однако обратите внимание на то, что одни узлы графа присутствуют в разных треугольниках — даже в треугольниках, не являющихся смежными в структуре дерева, и некоторые ребра соединяют узлы треугольников, находящихся очень далеко в древовидной структуре, — например, центральный треугольник соединяется ребрами с узлами всех остальных треугольников. Как точно сформулировать соответствие между деревом и графом? Для этого мы введем понятие *древовидной декомпозиции* графа *G*, которая называется так потому, что мы хотим провести декомпозицию *G* по древовидной схеме.

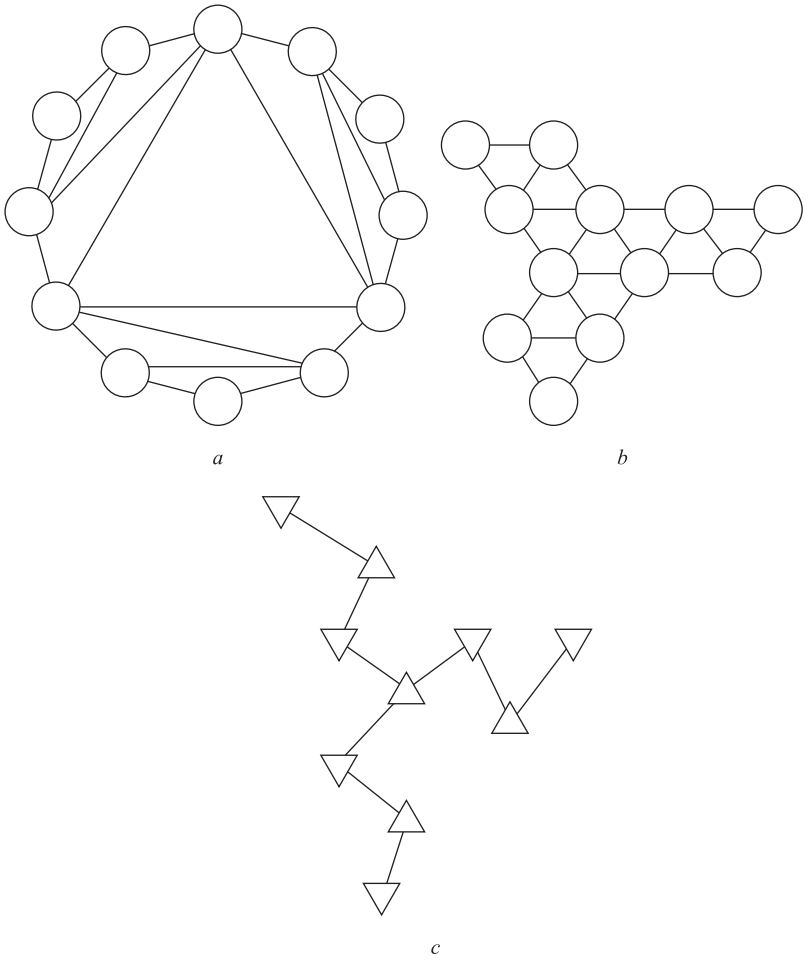


Рис. 10.5. На рисунке *a* и *b* изображен один граф в разных представлениях. Рисунок *b* подчеркивает, что граф состоит из десяти сплетенных треугольников. На рисунке *c* схематично показано, как эти треугольники «сцепляются» друг с другом

Формально декомпозиция $G = (V, E)$ состоит из дерева T (множество узлов которого отличается от G) и подмножества $V_t \subseteq V$, связанного с каждым узлом t дерева T (мы будем называть эти подмножества V_t «фрагментами» декомпозиции). Иногда мы будем записывать ее в виде упорядоченной пары $(T,\{V_t: t \in T\})$. Дерево T и множество фрагментов $\{V_t: t \in T\}$ должны удовлетворять следующим трем условиям:

- (покрытие узлов): каждый узел G принадлежит минимум одному фрагменту V_t;
- (покрытие ребер): для каждого ребра e графа G, существует фрагмент V_t, содержащий оба конца e;

✦ (согласованность): пусть t_1, t_2 и t_3 — такие три узла T, что t_2 лежит на пути из t_1 в t_3. Если узел v графа G принадлежит как V_{t_1}, так и V_{t_3}, то он также принадлежит V_{t_2}.

Следует заметить, что дерево на рис. 10.5, *c* является древовидной декомпозицией графа, которая использует десять треугольников в качестве фрагментов.

Теперь рассмотрим случай, при котором граф G является деревом. Древовидная декомпозиция строится следующим образом: декомпозиционное дерево T содержит узел t_v для каждого узла v графа G и узел t_e для каждого ребра e графа G. Дерево T содержит ребро (t_v, t_e), где v — конец e. Наконец, если v является узлом, мы определяем фрагмент $V_{t_v} = \{v\}$, а если $e = (u, v)$ — ребро, то определяется фрагмент $V_{t_e} = \{u, v\}$. Теперь мы можем убедиться в том, что три свойства определения древовидной декомпозиции выполнены.

Свойства декомпозиции

Присмотревшись к определению, мы видим, что свойства покрытия узлов и ребер просто гарантируют минимальное соответствие между набором фрагментов и графом G. Важнейшим аспектом определения является свойство согласованности. Хотя из его формулировки не очевидно, что согласованность приводит к свойствам древовидного разбиения, на самом деле это происходит вполне естественным образом. Деревья обладают двумя полезными и очень часто используемыми свойствами разбиения, которые тесно связаны друг с другом. Одно свойство утверждает, что при удалении ребра e дерево распадается ровно на две компоненты связности. Другое свойство утверждает, что при удалении узла t из дерева с последующим удалением всех инцидентных ему ребер дерево распадается на компоненты, число которых равно степени t. Свойство согласованности гарантирует, что разбиения T обоего типа имеют естественное соответствие среди разбиений G.

Если T' — подграф T, мы будем использовать обозначение $G_{T'}$ для подграфа G, определяемого всеми узлами всех фрагментов, связанных с узлами T', то есть множества $\cup_{t \in T'} V_t$.

Для начала рассмотрим удаление узла t из T.

(10.13) Предположим, $T - t$ содержит компоненты $T_1, ..., T_d$. В этом случае подграфы

$$G_{T_1} - V_t, G_{T_2} - V_t, ..., G_{T_d} - V_t$$

не имеют общих узлов и не соединяются ребрами.

Доказательство. На рис. 10.6 показано, как примерно выглядит разбиение. Сначала мы докажем, что подграфы $G_{T_i} - V_t$ не имеют общих узлов. В самом деле, любой такой узел v должен принадлежать как $G_{T_i} - V_t$, так и $G_{T_j} - V_t$ для некоторого $i \neq j$, поэтому такой узел v принадлежит некоторому фрагменту V_x для $x \in T_i$, и некоторому фрагменту V_y для $y \in T_j$. Так как t лежит на пути x-y в T, из свойства согласованности следует, что v лежит в V_t, а следовательно, не принадлежит ни $G_{T_i} - V_t$, ни $G_{T_j} - V_t$.

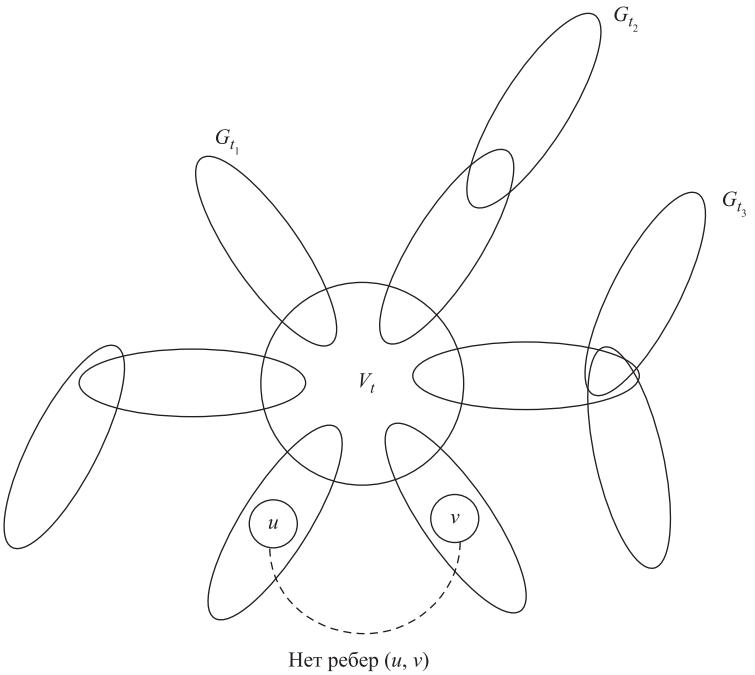


Рис. 10.6. Разбиения дерева T преобразуются в разбиения графа G

Затем мы должны показать, что в G нет ребра $e = (u, v)$, один конец которого u принадлежит подграфу $G_{T_i} - V_t$, а другой конец v принадлежит подграфу $G_{T_j} - V_t$ для некоторого $j \neq i$. Если бы такое ребро существовало, то по свойству покрытия ребер должен был существовать фрагмент V_x, соединяющий u и v. Узел x не может входить одновременно в подграфы T_i и T_j. Допустим, с учетом симметрии $x \notin T_i$. Узел u принадлежит подграфу G_{T_i}, поэтому узел u должен принадлежать множеству V_y для некоторого y в T_i. Тогда узел u принадлежит как V_x, так и V_y, а поскольку t лежит на пути x-y в T, из этого следует, что u также принадлежит V_t, поэтому он не лежит в $G_{T_i} - V_t$, как и требовалось. ■

Свойство разбиения по ребрам доказывается аналогично. Если удалить ребро (x, y) из T, то T разбивается на две компоненты: X (содержит x) и Y (содержит Y). Установим соответствующий способ разбиения G этой операцией.

(10.14) Пусть X и Y — две компоненты T после удаления ребра (x, y). Тогда удаление множества $V_x \cap V_y$ из V разбивает G на два подграфа $G_X - (V_x \cap V_y)$ и $G_Y - (V_x \cap V_y)$. А точнее, эти два подграфа не имеют общих узлов и не существует ребра, один конец которого принадлежал бы каждому из них.

Доказательство. Общая схема разбиения изображена на рис. 10.7. Доказательство этого свойства аналогично доказательству (10.13). Сначала мы доказываем, что два подграфа $G_X - (V_x \cap V_y)$ и $G_Y - (V_x \cap V_y)$ не имеют общих узлов, показывая, что узел v, который принадлежит как G_X, так и G_Y, должен принадлежать как V_x, так и V_y, а следовательно, он не входит ни в $G_Y - (V_x \cap V_y)$, ни в $G_X - (V_x \cap V_y)$.

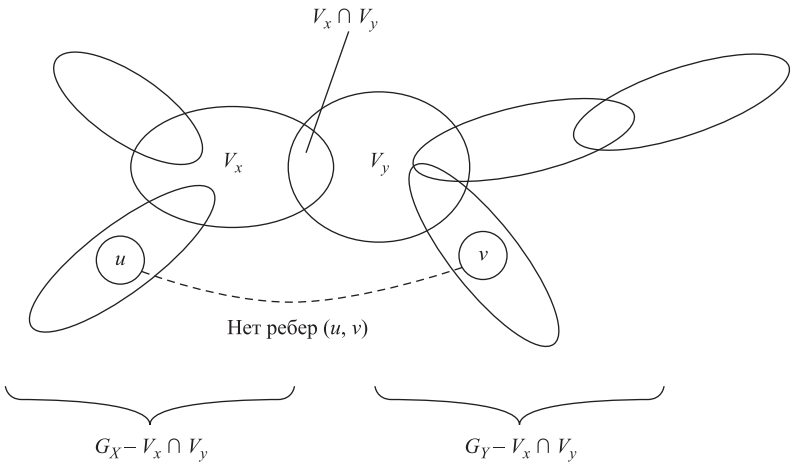


Рис. 10.7. Удаление ребра дерева T преобразуется в разбиение графа G

Теперь нужно показать, что в G не существует ребра $e = (u, v)$, один конец которого u принадлежит $G_X - (V_x \cap V_y)$, а другой конец v принадлежит $G_Y - (V_x \cap V_y)$. Если бы такое ребро существовало, то по свойству покрытия ребер должен существовать фрагмент V_z, содержащий u и v. Допустим, с учетом симметрии $z \in X$. Узел v также принадлежит некоторому фрагменту V_w для $w \in Y$. Поскольку x и y лежат на пути w-z в T, из этого следует, что V принадлежит V_x и V_y. Следовательно, $v \in V_x \cap V_y$ не принадлежит $G_Y - (V_x \cap V_y)$, как и требуется. ■

Итак, древовидные декомпозиции полезны тем, что свойства разбиения T переносятся на G. Возможно, кому-то покажется, что дальше нужно найти ответ на ключевой вопрос: какие графы имеют древовидные декомпозиции? Но в действительности это не так, потому что если подумать, станет ясно, что у любого графа существует древовидная декомпозиция. Для любого графа G можно выбрать в качестве T дерево, состоящее из одного узла t, а единственным фрагментом V_t назначить все множество узлов G. Такое определение легко выполняет три свойства, требуемых в определении; тем не менее пользы от такой декомпозиции ничуть не больше, чем от исходного графа.

Следовательно, важно найти древовидную декомпозицию, в которой все фрагменты малы. Именно это свойство мы пытаемся перенести с деревьев, требуя, чтобы удаление очень малых множеств узлов разбивало граф на несвязные подграфы. Соответственно мы определяем ширину древовидной декомпозиции $(T,\{V_t\})$ на 1 менее максимального размера любого фрагмента V_t:

$$\text{width}(T, \{V_t\}) = \max_t |V_t| - 1.$$

Древовидная ширина G определяется как минимальная ширина любой древовидной декомпозиции G. Из-за свойства покрытия ребер все древовидные декомпозиции должны содержать фрагменты не менее чем с двумя узлами, а следовательно, иметь древовидную ширину не менее 1. Вспомните, что наша древовидная декомпозиция для дерева G имеет ширину 1, так как каждое из множеств V_t

содержит либо один, либо два узла. Несколько странное «–1» в этом определении нужно для того, чтобы деревья имели ширину 1 вместо 2. Кроме того, все графы с нетривиальной декомпозицией, обладающие древовидной шириной w, имеют разделители с размером w, потому что если (x, y) — ребро дерева, согласно (10.14), удаление $V_x \cap V_y$ разбивает G на две компоненты.

Итак, мы можем говорить о множестве всех графов с древовидной шириной 1, множестве всех графов с древовидной шириной 2 и т. д. Следующий факт устанавливает, что из всех графов только деревья имеют древовидную ширину 1, а следовательно, наши определения действительно обобщают концепцию дерева. Доказательство также предоставляет хорошую возможность для проявления некоторых базовых свойств декомпозиций. Также заметим, что граф на рис. 10.5 в соответствии с определением древовидной ширины относится ко второму по «простоте» классу графов после деревьев: это граф с древовидной шириной 2.

(10.15) Древовидная ширина связного графа G равна 1 в том, и только в том случае, если он является деревом.

Доказательство. Ранее уже было показано, что если граф G является деревом, то для него можно построить декомпозицию с древовидной шириной 1.

Чтобы доказать обратное, сначала заметим один полезный факт: если H — подграф G, то древовидная ширина H не превышает древовидной ширины G. Это доказывается хотя бы тем, что для заданной декомпозиции $(T, \{V_t\})$ графа G можно определить декомпозицию H с тем же деревом T, заменив каждый фрагмент V_t на $V_t \cap H$. Легко проверить, что три обязательных свойства продолжают выполняться. (Тот факт, что некоторые фрагменты могут быть равны пустому множеству, не создает проблем.)

Теперь предположим от противного, что G — связный граф с древовидной шириной 1, который не является деревом. Так как G не является деревом, он содержит подграф, состоящий из простого цикла C. Из предыдущего абзаца следует, что теперь нам достаточно доказать, что граф C не имеет древовидной ширины 1. В самом деле, предположим, что у него существует декомпозиция $(T, \{V_t\})$, в которой каждый фрагмент имеет размер не более 2. Выберем любые два ребра (u, v) и (u', v') в C; по свойству покрытия ребер существуют содержащие их фрагменты V_t и V_t'. На пути в T из t в t' должно существовать такое ребро (x, y), что фрагменты V_x и V_y не равны. Из этого следует, что $|V_x \cap V_y| \leq 1$. Теперь мы воспользуемся (10.14): определяя X и Y как компоненты $T - (x, y)$, содержащие x и y соответственно, мы видим, что удаление $V_x \cap V_y$ разделяет C на $C_X - (V_x \cap V_y)$ и $C_Y - (V_x \cap V_y)$. Ни один из этих двух подграфов не может быть пустым, так как один содержит $\{u, v\} - (V_x \cap V_y)$, а другой содержит $\{u', v'\} - (V_x \cap V_y)$. Но цикл невозможно разбить на два непустых подграфа удалением всего одного узла, поэтому мы приходим к противоречию. ■

Применяя декомпозицию в контексте алгоритмов динамического программирования, по соображениям эффективности было бы желательно, чтобы количество фрагментов было относительно небольшим. Этого можно добиться простым способом: если имеется древовидная декомпозиция $(T,\{V_t\})$ графа G и в T имеется такое ребро (x, y), что $V_x \subseteq V_y$, то мы можем свернуть ребро (x, y) («складывая»

фрагмент V_x в V_y) и получить декомпозицию G для меньшего дерева. Повторяя этот процесс необходимое количество раз, мы получим древовидную декомпозицию без избыточности: в полученном дереве не существует такого ребра (x, y), что $V_x \subseteq V_y$.

После получения декомпозиции можно проверить, что она не содержит слишком большого количества фрагментов:

(10.16) Любая неизбыточная древовидная декомпозиция n-узлового графа содержит не более n узлов.

Доказательство. Мы воспользуемся индукцией по n; случай n=1 очевиден. Рассмотрим случай n>1. Для имеющейся декомпозиции $(T,\{V_t\})$ n-узлового графа без избыточности мы сначала найдем лист t в T. Согласно условию неизбыточности, в V_t должен быть как минимум один узел, не входящий в соседний фрагмент, а значит (по свойству согласованности), не входящий в любой другой фрагмент. Пусть U — множество таких узлов в V_t. Заметим, что при удалении t из T и удалении V_t из множества фрагментов будет получена древовидная декомпозиция $G–U$ без избыточности. Согласно индукционной гипотезе, эта декомпозиция имеет не более $n - |U| \leq n - 1$ фрагментов, поэтому $(T, \{V_t\})$ содержит не более n фрагментов. ■

Хотя (10.16) помогает убедиться в том, что древовидная декомпозиция относительно невелека, чаще в процессе анализа графа проще начать с построения избыточной декомпозиции, чтобы позднее «сжать» ее до неизбыточной. Например, наша декомпозиция графа G, который представляет собой дерево, строит избыточный результат; с ходу описать неизбыточную декомпозицию было бы не так просто.

Итак, фундамент заложен, и мы можем обратиться к алгоритмическим применениям древовидных декомпозиций.

Динамическое программирование и древовидная декомпозиция

Мы начали с утверждения о том, что задача о независимом множестве с максимальным весом может быть эффективно решена для любого графа с ограниченной древовидной шириной. Пришло время выполнить обещание, а именно: мы разработаем алгоритм, который точно следует алгоритму с линейным временем для деревьев. Для заданного n-узлового графа, с которым связана древовидная декомпозиция с шириной w, этот алгоритм выполняется за время $O(f(w) \cdot n)$, где $f(\cdot)$ — экспоненциальная функция, которая зависит только от ширины w, а не от количества узлов n. И хотя мы будем заниматься задачей о независимом множестве с максимальным весом, как и в случае с деревьями, примененный метод пригодится при решении многих *NP*-сложных задач.

Итак, в очень конкретном смысле сложность задачи была переведена из размера графа в древовидную ширину, которая может быть намного меньше. Как упоминалось ранее, большие сети в реальном мире часто имеют очень малую древовидную ширину; и часто эта особенность проявляется не случайно, а вследствие структурированного или модульного подхода к их планированию. Итак, если вы имеете дело с сетью из 1000 узлов с древовидной декомпозицией ширины 4, рассматриваемый

метод позволяет преобразовать безнадежно неразрешимую задачу в потенциально выполнимую.

Конечно, ситуация отчасти напоминает алгоритм вершинного покрытия из раздела 10.1. Тогда экспоненциальная сложность была переведена в параметр k, размер искомого вершинного покрытия. На этот раз очевидного параметра, кроме n, не видно, поэтому нам пришлось изобрести неочевидный: древовидную ширину.

Чтобы разработать алгоритм, вспомним, что было сделано в случае дерева T. После определения корня T мы построили независимое множество, двигаясь вверх от листьев. В каждом внутреннем узле u мы перебираем возможные решения относительно узла u (включать или не включать), так как после фиксирования этого решения задачи разных поддеревьев под u становятся независимыми.

Обобщение для графа G с древовидной декомпозицией $(T, \{V_t\})$ ширины w выглядит очень похоже. Мы закрепляем корень T и строим независимое множество, рассматривая фрагменты V_t от листьев вверх. Во внутреннем узле t из T мы сталкиваемся со следующим основным вопросом: оптимальное независимое множество пересекает фрагмент V_t в некотором подмножестве U, но мы не знаем, какое это множество U. Соответственно мы перебираем все возможности для этого подмножества U, то есть все возможности относительно того, какие узлы из V_t включать, а какие не включать. Поскольку V_t может иметь размер до $w + 1$, появляется до 2^{w+1} рассматриваемых возможностей. Но теперь можно использовать два ключевых факта: во-первых, величина 2^{w+1} намного разумнее 2^n, когда w намного меньше n; и во-вторых, после фиксирования конкретной возможности 2^{w+1} (когда мы решим, какие узлы в фрагменте V_t будут включены), свойства разбиения (10.13) и (10.14) гарантируют, что задачи разных поддеревьев T под t могут решаться независимо. Таким образом, хотя мы довольствуемся поиском «грубой силы» на уровне *одного* фрагмента, в целом алгоритм достаточно эффективно работает на глобальном уровне при малом количестве отдельных фрагментов.

Определение подзадач

Корень дерева T помещается в узел r. Для любого узла t обозначим T_t поддерево с корнем t. Вспомните, что G_{T_t} обозначает подграф G, сформированный узлами всех фрагментов, ассоциированных с узлами T_t; для упрощения записи мы будем обозначать этот подграф G_t. Для подмножества U множества V общий вес узлов в U будет обозначаться $w(U)$; то есть $w(U) = \sum_{u \in U} w_u$.

Для каждого поддерева T_t определяется множество подзадач, соответствующих каждому возможному подмножеству U множества V_t, которое может представлять пересечение оптимального решения с V_t. Таким образом, для каждого независимого множества $U \subseteq V_t$ мы используем запись $f_t(U)$ для обозначения максимального веса независимого множества S в G_t, соответствующего требованию $S \cap V_t = U$. Величина $f_t(U)$ не определена, если U не является независимым множеством, поскольку в этом случае известно, что U не может представлять пересечение оптимального решения с V_t.

С каждым узлом t дерева T может быть связано не более 2^{w+1} подзадач, так как это максимально возможное число независимых подмножеств V_t. Согласно (10.16)

можно считать, что мы работаем с древовидной декомпозицией, имеющей не более n фрагментов; следовательно, общее количество подзадач не превышает $2^{w+1}n$. Очевидно, при наличии решений всех этих подзадач максимальный вес независимого множества в G можно определить по подзадачам, связанным с корнем r: нужно просто взять максимум по всем независимым множествам $U \subseteq V_r$ из $f_r(U)$.

Построение решений

Теперь мы должны показать, как строить решения этих подзадач посредством рекурсии. Начать несложно: когда t является листом, $f_t(U)$ равно $w(U)$ для каждого независимого множества $U \subseteq V_t$.

Предположим, что дочерние узлы t обозначены t_1, ..., t_d и мы уже определили значения $f_{t_i}(W)$ для каждого дочернего узла t_i и каждого независимого множества $W \subseteq V_{t_i}$. Как определить значение $f_t(U)$ для независимого множества $U \subseteq V_t$?

Пусть S — независимое множество с максимальным весом в G_t, для которого $S \cap V_t = U$; то есть $w(S) = f_t(U)$. Важно понять, как выглядит это множество S при пересечении с каждым из подграфов G_{t_i} (рис. 10.8). Обозначим S_i пересечение S с узлами G_{t_i}.

(10.17) S_i является независимым множеством с максимальным весом для G_{t_i} при условии, что $S_i \cap V_t = U \cap V_{t_i}$.

Доказательство. Предположим, существует независимое множество S'_i для G_{t_i} с тем свойством, что $S'_i \cap V_t = U \cap V_{t_i}$ и $w(S'_i) > w(S_i)$. Тогда рассмотрим множество $S' = (S - S_i) \cup S'_i$. Очевидно, $w(S') > w(S)$. Также легко убедиться в том, что $S' \cap V_t = U$.

Утверждается, что S' является независимым множеством в G; это будет противоречить нашему выбору S как независимого множества с максимальным весом в G_t, соответствующего условию $S \cap V_t = U$. Например, предположим, что S' не является независимым; пусть $e = (u, v)$ — ребро, оба конца которого принадлежат S'. Однако u и v одновременно S принадлежать не могут, иначе они будут принадлежать S'_i, так как оба являются независимыми множествами. Следовательно, $u \in S - S'_i$ и $v \in S'_i - S$, из чего следует, что u не является узлом G_{t_i}, а $v \in G_{t_i} - (V_t \cap V_{t_i})$. Но тогда согласно (10.14) не может быть ребра, соединяющего u и v. ■

Утверждение (10.17) — именно то, что необходимо для определения рекуррентного отношения для подзадач. Оно сообщает, что информация, необходимая для вычисления $f_t(U)$, неявно содержится в значениях, уже вычисленных для поддеревьев. А именно: для всех дочерних t_i достаточно просто определить значение независимого множества с максимальным весом S_i для G_{t_i} с учетом ограничения $S_i \cap V_r = U \cap V_{t_i}$. Это ограничение не определяет точно, что собой представляет $S_i \cap V_{t_i}$; оно лишь говорит, что это может быть любое независимое множество $U_i \subseteq V_{t_i}$, для которого $Ui \cap V_t = U \cap V_{t_i}$. Следовательно, вес оптимального варианта S_i равен

$$\max \{ f_{t_i}(U_i) : U_i \cap V_t = U \cap V_{t_i} \text{ и } U_i \subseteq V_{t_i} \text{ независимое} \}.$$

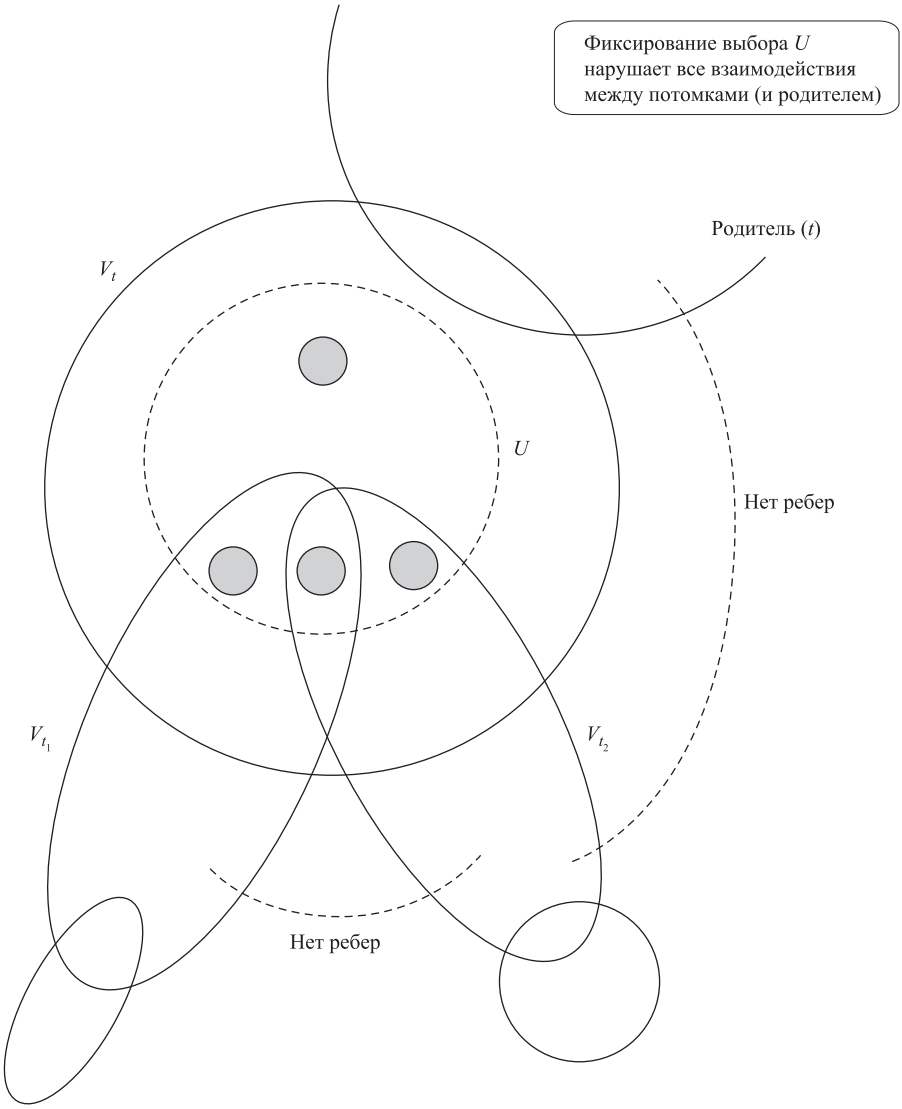


Рис. 10.8. Подзадача $f_t(U)$ в подграфе G_t. В оптимальном решении этой подзадачи мы рассматриваем независимые множества S_i в подграфах-потомках G_{t_i}, для которых $S_i \cap V_t = U \cap V_{t_i}$

Наконец, значение $f_t(U)$ равно просто $w(U)$ в сумме с максимумами, добавленными по d дочерним узлам t, — не считая того, что для предотвращения повторного подсчета узлов U мы исключаем их из вклада дочерних узлов. Таким образом:

(10.18) Значение $f_t(U)$ задается следующим рекуррентным отношением:

$$f_t(U) = w(U) + \sum_{i=1}^{d} \max\left\{f_{t_i}(U) - w(U_i \cap U)\right\}:$$

$U_i \cap V_t = U \cap V_{t_i}$ и $U_i \subseteq V_{t_i}$ независимое}.

Обобщенный алгоритм просто строит значения всех подзадач по листьям T и вверх.

```
Чтобы найти независимое множество с максимальным весом в графе G
для заданной древовидной декомпозиции (T, {V_t}) графа G:
   При необходимости изменить декомпозицию так, чтобы
   она не имела избыточности
   Разместить корень T в узле r
   Для каждого узла t из T в обратном порядке обхода
      Если t является листом
         Для каждого независимого множества U из V_t
            f_t(U) = w(U)
      Иначе
         Для каждого независимого множества U из V_t
            f_t(U) определяется рекуррентным отношением из (10.18)
      Конец Если
   Конец цикла
   Вернуть max {fr (U):U ⊆ V_r независимое}
```

Как обычно, независимое множество с максимальным весом находится обратным отслеживанием по последовательности выполнения.

Время, необходимое для вычисления $f_t(U)$, находится следующим образом: для каждого из d дочерних узлов t_i и для каждого независимого множества U_i в V_{t_i} мы за время $O(w)$ проверяем условие $U_i \cap V_t = U \cap V_{t_i}$, чтобы узнать, должно ли оно рассматриваться при вычислении (10.18).

Для $f_t(U)$ общее время составляет $O(2^{w+1}wd)$; так как с t связано не более 2^{w+1} множеств U, общие затраты времени на узел t составляют $O(4^{w+1}wd)$. Наконец, эти величины суммируются по всем узлам t для получения общего времени выполнения. Мы видим, что сумма по всем узлам t количества дочерних узлов t равна $O(n)$, так как каждый узел считается как дочерний один раз. Следовательно, общее время выполнения составляет $O(4^{w+1}wn)$.

10.5.* Построение древовидной декомпозиции

В предыдущем разделе были представлены концепции декомпозиции графа в дерево и древовидной ширины, а также рассмотрен канонический пример решения *NP*-сложной задачи для графов с ограниченной древовидной шириной.

Задача

В нашем алгоритмическом использовании древовидной ширины все еще не хватает одной важной составляющей. Пока что мы просто предоставили алгоритм для нахождения независимого множества с максимальным весом для графа G для заданной древовидной декомпозиции G с малой шириной. А если граф G просто появляется «на ровном месте» и никто не побеспокоился предоставить хорошую древовидную декомпозицию? Можно ли вычислить ее самостоятельно, а затем перейти к алгоритму динамического программирования?

В общих чертах — да, можно, но не без проблем. Сначала необходимо предупредить о том, что для заданного графа G определение его древовидной ширины является *NP*-сложной задачей. Впрочем, все не так плохо, потому что нас интересуют только графы, для которых древовидная ширина является малой константой. А в этом случае алгоритм описывается со следующей гарантией: для заданного графа G с древовидной шириной менее w он строит декомпозицию G в дерево с шириной менее $4w$ за время $O(f(w)\cdot mn)$, где m и n — количество ребер и узлов в G, а $f(\cdot)$ — функция, зависящая только от w. Итак, по сути при небольшой древовидной ширине существует достаточно быстрый способ построения древовидной декомпозиции с почти минимально возможной шириной.

Разработка и анализ алгоритма

Первым шагом при разработке алгоритма для этой задачи является выявление разумного «препятствия» к тому, чтобы граф G имел малую древовидную ширину. Другими словами, когда мы пытаемся построить древовидную декомпозицию малой ширины для $G = (V, E)$, может ли существовать некая «локальная» структура, обнаружение которой укажет на то, что древовидная ширина на самом деле должна быть большой?

Оказывается, следующая идея помогает найти такое препятствие. Во-первых, два множества $Y, Z \subseteq V$ с одинаковым размером называются *разделимыми*, если некоторое строго меньшее множество может полностью нарушить их связность, а именно если существует такое множество $S \subseteq V$, что $|S| < |Y| = |Z|$ и не существует пути от $Y - S$ к $Z - S$ в $G - S$. (В этом определении Y и Z не обязаны быть непересекающимися.) Далее множество X узлов в G будет называться *w-связным*, если $|X| \geq w$ и X не содержит разделяемые подмножества Y и Z, такие что $|Y| = |Z| \leq w$.

Для последующего алгоритмического применения w-связных множеств обратите внимание на следующий факт:

(10.19) Пусть граф $G = (V, E)$ имеет m ребер, X — множество из k узлов в G, а $w \leq k$ — заданный параметр. Тогда можно определить, является ли множество X w-связным, за время $O(f(k)\cdot m)$, где $f(\cdot)$ зависит только от k. Более того, если множество X не является w-связным, доказательство этого факта можно вернуть в форме множеств $Y, Z \subseteq X$ и $S \subseteq V$, для которых $|S| < |Y| = |Z| \leq w$ и не существует пути от $Y - S$ к Z − S в $G - S$.

Доказательство. Нужно решить, содержит ли X разделимые подмножества Y и Z, для которых $|Y| = |Z| \le w$. Сначала можно перебрать все пары достаточно малых подмножеств Y и Z; так как X имеет только 2^k подмножеств, количество таких пар не превышает 4^k.

Затем для каждой пары подмножеств Y, Z необходимо определить, являются ли эти множества разделимыми. Пусть $\ell = |Y| = |Z| \le w$. Но ситуация точно совпадает с теоремой о максимальном потоке и минимальном разрезе для ненаправленного графа с пропускными способностями узлов: Y и Z разделимы в том, и только в том случае, если не существуют ℓ путей, не пересекающихся по узлам, один конец которых находится в Y, а другой в Z. (За версией теоремы о максимальном потоке с пропускными способностями узлов обращайтесь к упражнению 13 главы 7.) Чтобы определить, существуют ли такие пути, можно воспользоваться алгоритмом для потока с (единичными) пропускными способностями узлов; это занимает время $O(\ell m)$. ■

W-связное множество можно представить себе как «тесно сплетенное» — у него нет двух малых частей, которые можно легко отделить друг от друга. В то же время декомпозиция производит разбивку графа с использованием очень малых разделителей; интуитивно понятно, что эти две структуры могут рассматриваться как противоположности.

(10.20) Если граф G содержит $(w + 1)$-связное множество с размером не менее $3w$, то древовидная ширина G не менее w.

Доказательство. Предположим от обратного, что граф G имеет $(w + 1)$-связное множество X с размером не менее $3w$, а также для него существует декомпозиция $(T, \{V_t\})$ с шириной менее w; иначе говоря, размер каждого фрагмента V_t не превышает w. Также можно считать, что декомпозиция $(T, \{V_t\})$ не содержит избыточности.

Идея доказательства заключается в том, чтобы найти фрагмент V_t, расположенный «по центру» в отношении X, чтобы при удалении из G некоторой части V_t одно малое подмножество X отделялось от другого. Так как размер V_t не превышает w, это будет противоречить нашему предположению о том, что X является $(w + 1)$-связным.

Как же найти этот фрагмент V_t? Сначала мы размещаем корень дерева T в узле r; как и прежде, обозначим T_t поддерево с корнем в узле t и запишем G_t для G_{T_t}. Пусть теперь t — узел, находящийся как можно дальше от корня r, с тем ограничением, что G_t содержит более $2w$ узлов X.

Очевидно, t не является листом (в противном случае подграф G_t содержал бы не более w узлов X; обозначим $t_1, ..., t_d$ дочерние узлы t. Обратите внимание: так как каждый узел t_i находится от t на большем расстоянии, чем корень, каждый подграф G_{T_i} содержит не более $2w$ узлов из X. Если существует дочерний узел t_i, для которого G_{t_i} содержит не менее w узлов из X, мы можем определить Y как w узлов X, принадлежащих G_{t_i}, а Z — как w узлов X, принадлежащих $G - G_{t_i}$. Так как декомпозиция $(T, \{V_t\})$ не имеет избыточности, размер $S = V_{t_i} \cap V_t$ не превышает $w - 1$; но согласно (10.14), удаление S приводит к отделению $Y - S$ от $Z - S$. Это противоречит нашему предположению о том, что множество X является $(w + 1)$-связным.

Соответственно мы рассматриваем случай, в котором нет такого дочернего узла t_i, для которого G_{t_i} содержит не менее w узлов из X; структура графа в этом случае выглядит примерно так, как показано на рис. 10.9. Мы начинаем с множества узлов G_{t_1}, объединяем его с G_{t_2}, затем G_{t_3} и т. д., пока впервые не получим множество узлов, содержащее более w членов X. Очевидно, это уже случится к тому моменту, когда мы доберемся до G_{t_d}, потому что G_t содержит более 2^w узлов из X и не более w из них могут принадлежать V_t. Допустим, процесс объединения G_{t_1}, G_{t_2}, ... впервые дает более w членов X при достижении индекса $i \le d$. Обозначим W множество узлов в подграфах G_{t_1}, G_{t_2}, ..., G_{t_i}. Согласно условию завершения $|W \cap X| > w$. Но поскольку G_{t_i} содержит менее w узлов X, также имеем $|W \cap X| < 2w$. Следовательно, мы можем определить Y как $w + 1$ узлов X, принадлежащих W, а Z — как $w + 1$ узлов X, принадлежащих $V-W$. Согласно (10.13) фрагмент V_t теперь является множеством с размером, не превышающим w, удаление которого приводит к отделению $Y - V_t$ от $Z - V_t$. И снова это противоречит предположению о том, что множество X является $(w + 1)$-связным, завершая наше доказательство. ■

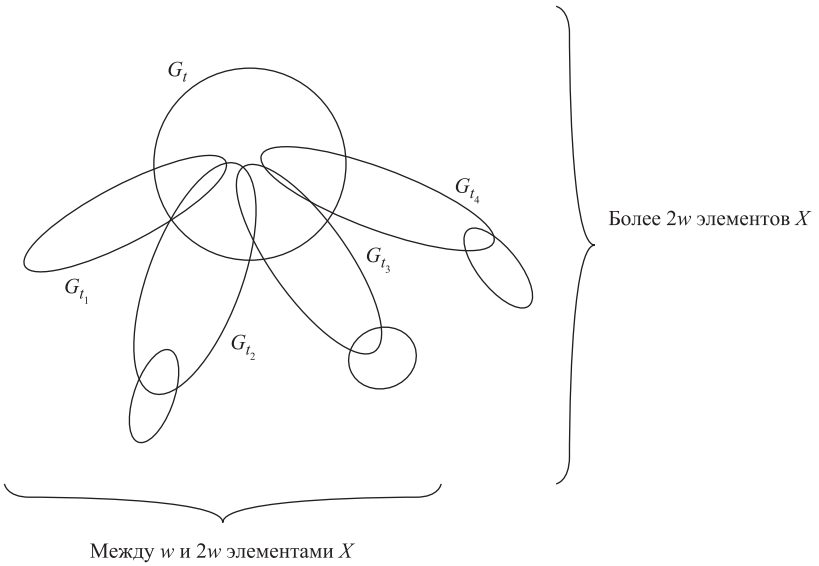


Рис. 10.9. Последний шаг доказательства (10.20)

Алгоритм поиска декомпозиции с малой древовидной шириной

Продолжая развивать эти идеи, мы приведем жадный алгоритм для построения древовидной декомпозиции низкой ширины. Алгоритм не будет точно определять древовидную ширину входного графа $G = (V, E)$; вместо этого он по заданному параметру w либо строит декомпозицию ширины менее $4w$, либо обнаруживает

($w + 1$)-связное множество с размером не менее $3w$. В последнем случае это составляет доказательство того, что древовидная ширина G не менее w согласно (10.20); итак, наш алгоритм, по сути, способен сузить фактическую древовидную ширину G до 4 раз. Как упоминалось ранее, время выполнения определяется в форме $O(f(w) \cdot mn)$, где m и n — количество ребер и узлов G, а $f(\cdot)$ зависит только от w.

С некоторым опытом работы с декомпозициями можно представить себе, что может потребоваться для построения декомпозиции для произвольного входного графа G. На верхнем уровне этот процесс изображен на рис. 10.10. Наша цель — заставить G распасться на древовидные части; декомпозиция начинается с размещения первого фрагмента V_t в любом месте. Если повезет, $G - V_t$ состоит из нескольких изолированных компонент; мы рекурсивно заходим в каждую компоненту и размещаем в ней фрагмент так, чтобы он частично перекрывал уже определенный фрагмент V_t. Мы надеемся, что эти новые фрагменты приведут к дальнейшему разбиению графа; этот процесс продолжается далее. Ключевую роль в работе этого алгоритма играет следующий факт: если в какой-то момент мы окажемся в тупике и наши малые множества не приведут к дальнейшему разбиению графа, можно будет извлечь большое ($w + 1$)-связное множество, которое докажет, что древовидная ширина на самом деле была большой.

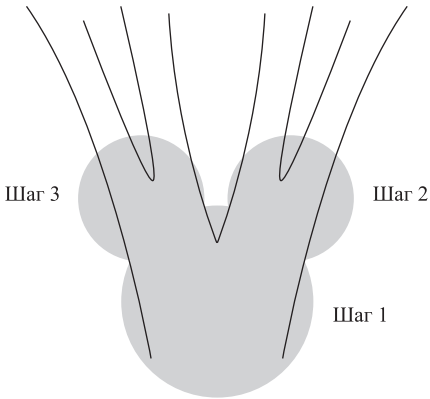


Рис. 10.10. Схематичное представление первых трех шагов в построении декомпозиции графа. Так как каждый шаг порождает новый фрагмент, нашей целью является разбиение остатка графа на несвязанные компоненты для применения дальнейших итераций алгоритма

Все эти объяснения выглядят несколько туманно, но реальный алгоритм следует им точнее, чем можно было ожидать. Мы начинаем с предположения о том, что не существует ($w + 1$)-связного множества с размером не менее $3w$; наш алгоритм строит декомпозицию, если условие выполняется; в противном случае его выполнение можно завершить с доказательством того, что древовидная ширина G не менее w. Дерево декомпозиции T и фрагменты V_t увеличиваются по жадному правилу. На каждой промежуточной стадии алгоритма сохраняется свойство *частичной декомпозиции*: если $U \subseteq V$ — множество узлов G, принадлежащих минимум одному из уже построенных фрагментов, то текущее дерево T и фрагменты V_t должны образовать древовидную декомпозицию подграфа G, порожденного U. Мы

определяем ширину частичной декомпозиции (по аналогии с определением ширины декомпозиции в дерево) на 1 меньше максимального размера фрагмента. Это означает, что для достижения цели — ширины менее $4w$ — достаточно убедиться в том, что размер всех фрагментов не превышает $4w$. Если C — компонента связности $G–U$, то $u \in U$ называется соседом C, если существует некоторый узел $v \in C$, соединенный ребром с u. Для работы алгоритма важно не просто поддерживать частичную декомпозицию с шириной менее $4w$, но и обеспечить постоянное соблюдение следующего инварианта:

(*) На любой стадии выполнения алгоритма каждая компонента C подграфа $G–U$ имеет не более $3w$ соседей, и существует единственный фрагмент V_t, который содержит их все.

Чем этот инвариант так полезен? Тем, что он позволит нам добавить новый узел s в T и нарастить новый фрагмент V_s в компоненте C с уверенностью в том, что s может быть листом, «свисающим» с t в большей частичной декомпозиции.

Более того, (*) требует, чтобы соседей было не более $3w$, при том что мы пытаемся построить декомпозицию с шириной менее $4w$; дополнительное w дает новому фрагменту «свободное место» для расширения при перемещении в C.

А теперь мы опишем, как добавить новый узел и новый фрагмент так, чтобы поддерживалась частичная декомпозиция дерева, не нарушался инвариант (*), а множество U строго увеличилось. При этом алгоритм продвинется по крайней мере на один узел, а следовательно, завершится не более чем за n итераций древовидной декомпозицией всего графа G.

Пусть C — любая компонента $G - U$, X — множество соседей U, а V_t — фрагмент, который, как гарантирует (*), содержит все множество X. Мы знаем (снова из (*)), что X содержит не более $3w$ узлов. Если X содержит строго меньше $3w$ узлов, можно продвинуться вперед немедленно: для любого узла $v \in C$ определяется новый фрагмент $V_s = X \cup \{v\}$, а s становится листом t. Так как концы всех ребер из v в U принадлежат X, легко убедиться в том, что мы имеем частичную декомпозицию, подчиняющуюся (*), а множество U увеличилось.

Предположим, X содержит ровно $3w$ узлов. В этом случае дальнейшее уже не столь очевидно; например, если попытаться создать новый фрагмент произвольным добавлением узла $v \in C$ в X, может получиться компонента из $C - \{v\}$ (в том числе и все множество $C - \{v\}$), множество соседей которой включает все $3w + 1$ узлов $X \cup \{v\}$, а это нарушит (*).

Простого выхода не существует; прежде всего, граф G может и не иметь древовидной декомпозиции с малой шириной. А значит, именно здесь стоит задаться вопросом, создает ли X настоящее препятствие для декомпозиции или нет: мы проверяем, является ли X $(w + 1)$-связным множеством. Согласно (10.19), ответ может быть получен за время $O(f(w) \cdot m)$, так как $|X| = 3w$. Если окажется, что множество X является $(w + 1)$-связным, то дело сделано; выполнение можно прервать с заключением о том, что G имеет древовидную ширину не менее w, что является одним из допустимых результатов алгоритма. С другой стороны, если множество X не является $(w + 1)$–связным, то мы получаем $Y, Z \subseteq X$ и $S \subseteq V$, такие что

$|S| < |Y| = |Z| \le w + 1$, и не существует пути от $Y - S$ к $Z - S$ в $G - S$. Множества Y, Z и S теперь предоставляют средства для расширения частичной декомпозиции.

Пусть множество S' состоит из всех узлов S, лежащих в $Y \cup Z \cup C$. Ситуация выглядит так, как показано на рис. 10.11. Заметим, что множество $S' \cap C$ не пусто: из каждого из множеств Y и Z выходят ребра в C, и если бы множество $S' \cap C$ было пустым, то существовал бы путь от $Y - S$ к $Z - S$ в $G - S$, который начинается в Y, переходит прямо в C, проходит через C и, наконец, снова переходит в Z. Кроме того, $|S'| \le |S| \le w$.

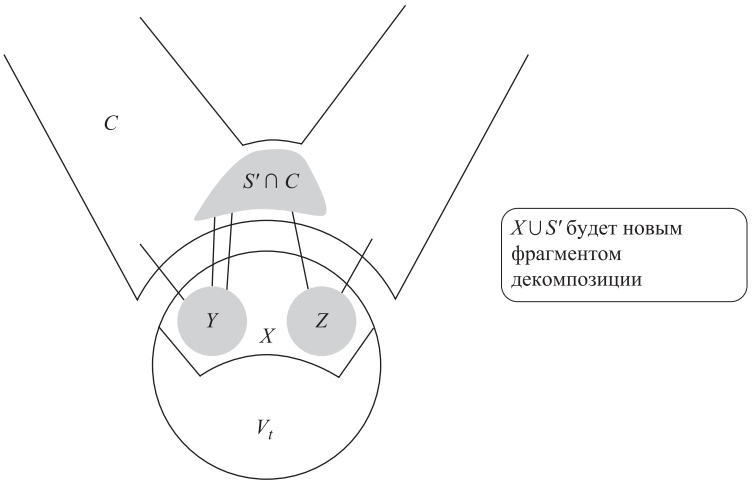


Рис. 10.11. Добавление нового фрагмента в частичную декомпозицию

Мы определяем новый фрагмент $V_s = X \cup S'$, делая s листом t. Концы всех ребер из S' в U принадлежат X и $|X \cup S'| \le 3w + w = 4w$, так что частичная декомпозиция сохраняется. Более того, множество узлов, покрываемое нашей частичной декомпозицией, увеличилось, так как множество $S' \cap C$ не пусто. Следовательно, если нам удастся показать, что инвариант (*) по-прежнему выполняется, работа будет завершена. И здесь мы приходим к интуитивному представлению, которое мы пытались отразить при обсуждении рис. 10.10: при добавлении нового фрагмента $X \cup S'$ мы надеемся, что компонента C хорошо разделится на дальнейшие компоненты.

Конкретнее, наша частичная декомпозиция теперь покрывает $U \cup S'$; и там, где прежде была компонента C из $G - U$, сейчас могут быть несколько компонент $C' \subseteq C$ из $G - (U \cup S')$. У каждой из этих компонент C' все ее соседи принадлежат $X \cup S'$; но мы должны дополнительно убедиться в том, что таких соседей не более $3w$, чтобы не нарушался инвариант (*). Рассмотрим одну из таких компонент C'. Утверждается, что все ее соседи в $X \cup S'$ принадлежат одному из двух подмножеств $(X - Z) \cup S'$ или $(X - Y) \cup S'$ и размер каждого из этих множество не превышает $|X| \le 3w$. Если бы это условие не выполнялось, то у C' был бы сосед, принадлежащий одновременно $Y - S$ и $Z - S$, а следовательно, существовал бы путь через C' от $Y–S$ к $Z - S$ в $G - S$. Но, как было показано ранее, такого пути быть не может. Из этого

следует, что (*) продолжает выполняться после добавления нового фрагмента, что завершает обоснование правильности работы алгоритма.

Наконец, что можно сказать о времени выполнения алгоритма? Во времени добавления нового фрагмента в частичную декомпозицию доминирует время, необходимое для проверки того, является ли множество X $(w+1)$-связным, которое составляет $O(f(w)\cdot m)$. Это происходит не более чем в n итерациях, так как в каждой итерации увеличивается количество задействованных узлов G. Следовательно, общее время выполнения равно $O(f(w)\cdot mn)$.

Ниже перечислены основные свойства нашего алгоритма декомпозиции.

(10.21) Для заданного графа G и параметра w алгоритм древовидной декомпозиции, описанный в этом разделе, делает одно из двух:

- строит декомпозицию с шириной менее $4w$, или
- сообщает (обоснованно), что граф G не имеет древовидной ширины менее w.

Время выполнения алгоритма равно $O(f(w)\cdot mn)$, где функция $f(\cdot)$ зависит только от w.

Упражнения с решениями

Упражнение с решением 1

Как было показано ранее, задача 3-SAT часто используется для моделирования сложных задач планирования и принятия решений в области искусственного интеллекта; переменные представляют бинарные решения, а условия — ограничения, установленные для этих решений. Системам, работающим с экземплярами 3-SAT, часто приходится представлять ситуации, в которых одни решения уже приняты, а другие остаются неопределенными. Для этого полезно ввести понятие *неполного присваивания* логических значений переменным.

А именно, для заданного множества булевых переменных $X = \{x_1, x_2, ..., x_n\}$ неполным присваиванием для X называется присваивание значения 0, 1 или ? каждому x_i; другими словами, это функция $\rho : X \to \{0, 1, ?\}$. Переменная x_i называется *определенной* неполным присваиванием, если она получает значение 0 или 1, и *неопределенной*, если она получает значение ?. Фактически неполное присваивание выбирает значение 0 или 1 для каждой из определенных переменных, а переменные со значением ? остаются неопределенными.

Теперь для заданного набора условий $C_1, ..., C_m$, каждое из которых представляет собой дизъюнкцию трех разных литералов, может возникнуть вопрос, достаточно ли неполного присваивания для «форсированного» выполнения набора условий независимо от состояния неопределенных переменных. Также может представлять интерес другой вопрос: существует ли неполное присваивание с небольшим количеством определенных переменных, которое может обеспечить выполнение набора условий; эти переменные можно считать «влиятельными», потому что их достаточно для того, чтобы обеспечить выполнение условий.

Предположим, заданы условия

$$(x_1 \vee \overline{x}_2 \vee \overline{x}_4), (x_2 \vee \overline{x}_3 \vee x_4), (\overline{x}_2 \vee \overline{x}_3 \vee x_5), (x_1 \vee x_3 \vee \overline{x}_6).$$

Тогда неполное присваивание, которое задает $x_1 = 1$, $x_3 = 0$, а всем остальным переменным значение ?, имеет только две определенные переменные, но обеспечивает выполнение набора условий: как бы ни были заданы остальные четыре переменные, условия будут истинными.

Это определение можно формализовать. Вспомните, что логическим присваиванием для X называется присваивание значения 0 или 1 каждой переменной x_i; другими словами, каждой переменной должно быть присвоено булево значение и ни одна переменная не должна остаться неопределенной. Логическое присваивание ν называется *согласованным* с неполным присваиванием ρ, если каждая логическая переменна, определенная в ρ, имеет одинаковые значения в ρ и ν. (Другими словами, если $\rho(x_i) \neq ?$, то $\rho(x_i) = \nu(x_i)$.) Наконец, мы говорим, что неполное присваивание ρ *форсирует* набор условий $C_1, ..., C_m$, если для каждого логического присваивания ν, согласованного с ρ, ν выполняет $C_1, ..., C_m$. (Также ρ будет называться форсирующим неполным присваиванием.)

А теперь вопрос, следующий из этих определений. Имеется набор булевых переменных $X = \{x_1, x_2, ..., x_n\}$, параметр $b < n$, и набор условий $C_1, ..., C_m$ по переменным, в котором каждое условие представляет собой дизъюнкцию трех разных литералов. Требуется решить, существует ли для X форсирующее неполное присваивание ρ, в котором определено не более b переменных. Предложите алгоритм, который решает задачу с временем вида $O(f(b) \cdot p(n, m))$, где $p(\cdot)$ — полиномиальная функция, а $f(\cdot)$ — произвольная функция, зависящая только от b, но не от n или m.

Решение

Интуитивно понятно, что форсирующее неполное присваивание должно «соприкасаться» с каждым условием хотя бы в одном месте, потому что в противном случае оно не сможет влиять на его значение. Хотя это обстоятельство выглядит естественно, оно не является частью определения (в определении говорится только о логических присваиваниях, согласованных с частичным), поэтому мы начнем с его формализации и доказательства.

(10.22) Частичное присваивание ρ форсирует все условия в том, и только в том случае, если для каждого условия C_i по крайней мере одна из переменных в C_i определяется ρ так, что это выполняет C_i.

Доказательство. Очевидно, если ρ определяет по крайней мере одну переменную в каждом условии C_i так, что это условие выполняется, то как бы ни строилось полное логическое присваивание для остальных переменных, все условия уже выполнены. Таким образом, любое логическое присваивание, согласованное с ρ, выполняет все условия.

И наоборот, предположим, что существует такое условие C_i, что ρ не определяет никакие переменные в C_i способом, выполняющим C_i. Мы хотим показать, что ρ не является форсирующим, а для этого в соответствии с определением требуется представить присваивание, при котором выполняются не все условия. Рассмотрим

следующее логическое присваивание ν: ν соответствует ρ по всем определенным переменным; присваивает произвольное логическое значение каждой неопределенной переменной, не входящей в C_i; и присваивает каждой неопределенной переменной в C_i значение, с которым условие не выполняется. Так как *ν* присваивает каждой переменной в C_i значение, с которым оно не выполняется, следовательно, не является выполняющим присваиванием. Но ν согласуется с ρ, а из этого следует, что оно не является форсирующим неполным присваиванием. ■

Принимая во внимание (10.22), мы получаем задачу, которая очень похожа на задачу поиска малых вершинных покрытий из начала главы. Тогда требовалось найти множество узлов, покрывающее все ребра, и выбор ограничивался *k* узлами. Сейчас ищется множество переменных, покрывающее все условия (с правильными значениями истина/ложь), и выбор ограничивается *b* переменными.

Попробуем действовать аналогично тому, как мы действовали при поиске малого вершинного покрытия. Выберем произвольное условие C_l, содержащее литералы x_i, x_j и x_k (возможно, с инвертированием). Из (10.22) известно, что любое форсирующее присваивание ρ должно присвоить одной из этих трех переменных значение, входящее в C_l, поэтому мы можем опробовать все три возможности. Допустим, мы задали переменную x_i так, как она входит в C_l; тогда из экземпляра можно исключить все условия (включая *C′*), выполняемые этим присваиванием x_i, и попытаться выполнить оставшиеся. Назовем это уменьшенное множество условий экземпляром, *сокращенным по присваиванию* x_i. То же самое можно сделать для x_j и x_k. Так как присваивание ρ должно определять одну из этих переменных в том состоянии, в каком она входит в *C′*, а затем выполнять все остальное, нам удалось обосновать следующую аналогию с (10.3). (Чтобы упростить изложение, мы будем называть *размером* частичного присваивания количество определяемых им переменных.)

(10.23) Форсирующее присваивание с размером не более *b* существует в том, и только в том случае, если существует форсирующее присваивание с размером не более $b - 1$ для минимум одного из экземпляров, сокращенных по присваиванию x_i, x_j или x_k.

Итак, мы получаем следующий алгоритм. Его работа зависит от граничных случаев: отсутствия условий (тогда по определению выполнение считается успешным) и наличия условий с $b = 0$ (тогда следует объявление о неудаче).

```
Чтобы провести поиск форсирующего неполного присваивания с размером не более b:
   Если условий нет, по определению имеем форсирующее присваивание
   Иначе Если b = 0, то согласно (10.22) форсирующего присваивания нет
   Иначе пусть C_l – произвольное присваивание с переменными x_i, x_j и x_k
      Для каждой из переменных x_i, x_j и x_k:
         Задать переменной x* состояние, в котором она присутствует в C_l
         Сократить экземпляр по этому присваиванию
         Рекурсивно проверить наличие форсирующего присваивания
            с размером не более b – 1 для этого сокращенного экземпляра
      Конец цикла
      Если любой из этих рекурсивных вызовов (допустим, для x_i)
         возвращает форсирующее присваивание ρ' с размером не более b – 1
```

```
        Объединение ρ' с присваиванием x_i – искомый ответ
    Иначе (ни один из рекурсивных вызовов не завершился успехом)
        Форсирующего присваивания с размером не более b не существует
    Конец Если
Конец Если
```

Чтобы найти границу времени выполнения, мы рассматриваем дерево возможностей, по которому проводится поиск (как и в случае с алгоритмом нахождения вершинного покрытия). Каждый рекурсивный вызов порождает три дочерних узла в этом дереве, и это происходит на глубину до b. Таким образом, дерево содержит не более $1 + 3 + 3^2 + ... + 3^b \leq 3^{b+1}$ узлов, и в каждом узле для получения сокращенных экземпляров затраты времени не превышают $O(m + n)$. Следовательно, общее время выполнения составляет $O(3^b(m + n))$.

Упражнения

1. В упражнении 5 главы 8 утверждалось, что задача множества представителей является *NP*-полной. Чтобы вспомнить определения, возьмем множество $A = \{a_1, ..., a_n\}$ и набор $B_1, B_2, ..., B_m$ подмножеств A. Множество $H \subseteq A$ называется множеством представителей для набора $B_1, B_2, ..., B_m$, если H содержит по крайней мере один элемент из каждого B_i, то есть если результат $H \cap B_i$ не пуст для каждого i. (Иначе говоря, H содержит «представителя» из каждого множества B_i.)

   Теперь предположим, что для имеющегося экземпляра задачи нужно определить, существует ли множество представителей с размером не более k. Кроме того, предположим, что каждое множество B_i содержит не более c элементов для константы c. Предложите алгоритм, который решает задачу с временем выполнения вида $O(f(c, k) \cdot p(n, m))$, где $p(\cdot)$ — полиномиальная функция, а $f(\cdot)$ — произвольная функция, зависящая только от c и k, но не от n или m.

   Сложность задачи 3-SAT обусловлена тем фактом, что существуют 2^n возможных присваиваний входным переменным $x_1, x_2, ..., x_n$ при отсутствии очевидного способа поиска в этом пространстве за полиномиальное время. Впрочем, это интуитивное представление может создать неверное впечатление, что самые быстрые алгоритмы 3-SAT требуют времени 2^n. Когда вы слышите об этом впервые, это кажется противоестественным, однако для 3-SAT существуют алгоритмы, работающие значительно быстрее 2^n в худшем случае; другими словами, они определяют, существует ли выполняющее присваивание за меньшее время, чем необходимо для перебора всех возможных значений переменных.

   Здесь мы создадим один такой алгоритм, решающий экземпляры 3-SAT за время $O(p(n) \cdot (\sqrt{3})^n)$ для некоторой полиномиальной функции $p(n)$. Обратите внимание: основной составляющей в этом времени выполнения является $(\sqrt{3})^n$, то есть $1{,}74^n$.

(a) Для логического присваивания Φ с переменными $x_1, x_2, ..., x_n$ запись $\Phi(x_i)$ обозначает значение, присвоенное Φ переменной x_i (0 или 1). Для двух логических присваиваний Φ и Φ' *расстояние* между Φ and Φ' определяется как количество переменных x_i, которым присваиваются разные значения; это расстояние обозначается $d(\Phi, \Phi')$. Другими словами, $d(\Phi,\Phi') = |\{i : \Phi(x_i) \neq \Phi'(x_i)\}|$.

Основным структурным элементом этого алгоритма будет способность отвечать на вопросы следующего типа: для заданного логического присваивания Φ и расстояния d требуется определить, существует ли такое выполняющее присваивание Φ', что расстояние от Φ до Φ' не превышает d.

Рассмотрим следующий алгоритм Explore(Φ, d), который пытается ответить на этот вопрос:

```
Explore(Φ, d):
   Если Φ является выполняющим присваиванием, вернуть "да"
   Иначе Если d = 0, вернуть «нет»
   Иначе
      Пусть C_i – условие, не выполняемое Φ
         (то есть все три литерала в C_i ложны).
      Пусть Φ_1 - присваивание, полученное из Φ инвертированием
         переменной, которая встречается в первом литерале условия C_i
      Определить Φ_2 и Φ_3 аналогично для второго и третьего
         литералов условия C_i
      Рекурсивно вызвать:
         Explore(Φ_1, d − 1)
         Explore(Φ_2, d − 1)
         Explore(Φ_3, d − 1)
      Если хотя бы один из трех вызовов вернет “да”
        Вернуть “да”
      Иначе вернуть “нет”
```

Докажите, что Explore(Φ, d) возвращает «да» в том, и только в том случае, если существует выполняющее присваивание Φ', для которого расстояние от Φ до Φ' не превышает d. Также приведите анализ времени выполнения Explore(Φ, d) как функции от n и d.

(b) Очевидно, расстояние между любыми двумя присваиваниями Φ и Φ' не превышает n, так что один из способов решения заданного экземпляра 3-SAT заключается в выборе произвольного начального присваивания Φ и выполнении Explore(Φ, n). Однако это не даст нужного нам времени выполнения.

Вместо этого мы сделаем несколько вызовов Explore из разных начальных точек Φ и каждый раз будем проводить поиск на более ограниченное расстояние. Опишите, как это сделать для того, чтобы экземпляр 3-SAT решался за время $O(p(n) \cdot (\sqrt{3})^n)$.

2. Допустим, имеется направленный граф $G = (V, E)$ с $V = \{v_1, v_2, ..., v_n\}$ и нужно решить, существует ли в G гамильтонов путь от v_1 к v_n. (То есть существует ли в G путь, проходящий от v_1 к v_n через все остальные вершины ровно по одному разу?)

Так как задача о гамильтоновом пути является *NP*-полной, ожидать для нее решения с полиномиальным временем не приходится. Однако это не означает, что все алгоритмы с неполиномиальным временем одинаково «плохи». Например, простейшее решение методом «грубой силы» выглядит так: для каждой перестановки вершин проверить, образует ли она гамильтонов путь от v_1 к v_n. Это займет время, приблизительно пропорциональное $n!$, что составляет около 3×10^{17} для $n = 20$.

Покажите, что задача о гамильтоновом пути может быть решена за время $O(2^n \cdot p(n))$, где $p(n)$ — полиномиальная функция от n. Для умеренных значений n этот алгоритм работает намного эффективнее; например, значение 2^n для $n = 20$ всего лишь около миллиона.

3. Граф $G = (V, E)$ называется *триангулированным циклическим графом*, если он состоит из вершин и ребер триангулированного выпуклого n-угольника на плоскости, другими словами, если он может быть нарисован на плоскости следующим образом.

   Все вершины размещаются на контуре выпуклого множества на плоскости (будем считать, что на окружности), а каждая пара последовательных вершин соединяется ребром. Остальные ребра изображаются отрезками прямой во внутренней части круга, и никакая пара ребер не пересекается. Изображение графа должно обладать следующим свойством: если обозначить S множество всех точек плоскости, находящихся на вершинах или ребрах изображения, то после удаления S каждая ограниченная область плоскости ограничивается ровно тремя ребрами (собственно, поэтому граф и называется «триангулированным»).

   Триангулированный циклический граф изображен на рис. 10.12.

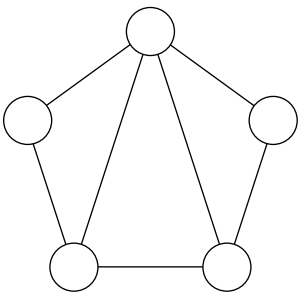


Рис. 10.12. Триангулированный циклический граф: ребра образуют границу выпуклого многоугольника и отрезки, разделяющие внутреннюю область на треугольники

   Докажите, что для каждого триангулированного циклического графа существует древовидная декомпозиция с шириной не более 2. Опишите эффективный алгоритм построения такой декомпозиции.

4. *Задача о доминирующем множестве с минимальной стоимостью* определяется ненаправленным графом $G = (V, E)$ и стоимостями $c(v)$ для узлов $v \in V$. Подмножество $S \subset V$ называется *доминирующим множеством*, если все узлы $u \in V - S$

соединяются ребром (u, v) с узлом v из S. (Обратите внимание на отличие между доминирующими множествами и вершинными покрытиями: в доминирующем множестве у ребра (u, v) узлы u и v могут не входить в множество S при условии, что у u и v имеются соседи в S.)

(a) Предложите алгоритм с полиномиальным временем для задачи о доминирующем множестве в частном случае, когда G является деревом.

(b) Предложите алгоритм с полиномиальным временем для задачи о доминирующем множестве в частном случае, когда G имеет древовидную ширину 2 и также имеется древовидная декомпозиция G с шириной 2.

5. Задача о путях, не пересекающихся по ребрам, определяется ненаправленным графом G и k парами узлов (s_i, t_i) для $i = 1, ..., k$. Требуется решить, существуют ли непересекающиеся по ребрам пути P_i, для которых путь P_i соединяет s_i с t_i. Предложите алгоритм с полиномиальным временем для задачи о путях, не пересекающихся по ребрам, для частного случая: граф G имеет древовидную ширину 2, и также имеется древовидная декомпозиция T графа G с шириной 2.

6. *Хроматическим числом* графа G называется минимальное значение k, при котором граф имеет k-раскраску. Как было показано в главе 8, принятие решения о том, имеет ли заданный входной граф хроматическое число $\leq k$, является *NP*-полной задачей.

(a) Покажите, что для каждого натурального числа $w \geq 1$ существует число $k(w)$, для которого выполняется следующее условие: если G — граф с древовидной шириной, не превышающей w, то G имеет хроматическое число, не превышающее $k(w)$. (Суть в том, что $k(w)$ зависит только от w, а не от количества узлов в G.)

(b) Для заданного ненаправленного n-узлового графа $G = (V, E)$ с древовидной шириной не более w покажите, как вычислить хроматическое число G за время $O(f(w) \cdot p(n))$, где $p(\cdot)$ — полиномиальная функция, но $f(\cdot)$ может быть произвольной функцией.

7. Рассмотрим класс экземпляров 3-SAT, в котором каждая из n переменных встречается (считая как простые, так и инвертированные вхождения) ровно в трех условиях. Покажите, что любой такой экземпляр 3-SAT выполним и что выполняющее присваивание может быть найдено за полиномиальное время.

8. Предложите алгоритм с полиномиальным временем для следующей задачи. Имеется бинарное дерево $T = (V, E)$ с нечетным числом узлов и неотрицательным весом каждого узла. Требуется найти разбиение узлов V на два множества *равного* размера с максимально возможным весом разреза между двумя множествами (то есть общим весом ребер, концы которых принадлежат разным подмножествам). Обратите внимание: ограничение на то, что граф является деревом, в этой задаче критично, а предположение о том, что дерево является бинарным, — нет. Для обобщенных графов задача является *NP*-сложной.

Примечания и дополнительная литература

Самая первая тема этой главы (о том, как избежать времени выполнения $O(kn^{k+1})$ для задачи о вершинном покрытии) является примером общей темы параметризованной сложности; для задач с двумя такими «параметрами размера» n и k время выполнения в форме $O(f(k)\cdot p(n))$, где $p(\cdot)$ — полиномиальная функция, обычно предпочтительнее времени выполнения в форме $O(nk)$. По этой теме была проведена значительная работа, включая разработку методологии выявления *NP*-полных задач, которые, скорее всего, не допускают такого улучшения времени выполнения. Эта область описана в книге Дауни и Феллоуза (Downey, Fellows, 1999).

NP-полнота задачи раскраски множества дуг была доказана Гэри, Джонсоном, Миллером и Пападимитриу (Garey, Johnson, Miller, Papadimitriou, 1980). Они также описали, как алгоритм, представленный в этой главе, напрямую вытекает из конструкции, предложенной Такером (Tucker, 1975). Задачи о раскраске интервалов и дуг принадлежат следующему классу задач: взять набор геометрических объектов (таких, как интервалы или дуги), определить граф соединением пар пересекающихся объектов и проанализировать задачу раскраски полученного графа. В книге о раскраске графов Дженсена и Тофта (Jensen, Toft, 1995) приведены описания многих других задач этого типа.

Важность декомпозиций и древовидной ширины вышла на передний план в основном благодаря работе Робертсона и Сеймура (Robertson, Seymour, 1990). Алгоритм построения декомпозиции, описанный в разделе 10.5, был предложен в работе Дистела и др. (Diestel et. al., 1999). Дальнейшее обсуждение древовидной ширины и ее роли в алгоритмах и теории графов приведено в обзоре Рида (Reed, 1997) и книге Дистела (Diestel, 2000). Древовидная ширина также играет важную роль в алгоритмах вероятностных моделей машинного обучения, основанных на логическом выводе Джордан (Jordan, 1998).

Примечания к упражнениям

Упражнение 2 основано на результатах Уве Шонинга; упражнение 8 основано на задаче, о которой мы узнали от Амита Кумара.

Глава 11

Аппроксимирующие алгоритмы

После обсуждения *NP*-полноты и концепции вычислительной неразрешимости в целом мы начали искать ответ на фундаментальный вопрос: как разрабатывать алгоритмы для задач, у которых полиномиальное время, скорее всего, является недостижимым?

В этой главе мы займемся новой темой, связанной с этим вопросом: *аппроксимирующие алгоритмы* выполняются за полиномиальное время и находят решения, гарантированно близкие к оптимальным. В этом определении следует обратить внимание на ключевые слова: *гарантированно* и *близкие*. Мы не пытаемся искать оптимальное решение, и в результате появляется возможность достичь полиномиального времени выполнения. В то же время нужно будет доказывать, что алгоритмы находят решения, гарантированно близкие к оптимуму. Эта задача весьма непростая по своей природе: чтобы доказать гарантию аппроксимации, необходимо сравнить решение с оптимальным решением, которое очень трудно найти (и, возможно, привести обоснование). Данная проблема неоднократно встречается при анализе алгоритмов в этой главе.

В этой главе рассматриваются четыре общие методологии разработки аппроксимирующих алгоритмов. Мы начнем с *жадных* алгоритмов, аналогичных тем, что разрабатывались в главе 4. Эти алгоритмы быстры и просты, как и в главе 4, поэтому основная проблема связана с поиском жадного правила, приводящего к решению, доказуемо близкого к оптимальному. Второй общий метод — *метод назначения цены* — имеет экономическую подоплеку; мы рассматриваем цену, которую необходимо уплатить за соблюдение каждого ограничения в задаче. Метод назначения цены часто называется *прямо-двойственным методом* (primal-dual) — термин происходит из области линейного программирования, которое тоже может использоваться в этой области. Описание метода назначения цены не требует знакомства с линейным программированием, которое будет представлено в третьей категории — *линейном программировании с округлением*, использующем отношения между вычислительной разрешимостью линейного программирования и выразительной мощью его «более сложного» родственника, *целочисленного программирования*. В завершение будет описан метод, приводящий к исключительно качественным аппроксимациям: применение динамического программирования к округленной версии входных данных.

11.1. Жадные алгоритмы и ограничения оптимума: задача распределения нагрузки

В первом разделе этой главы рассматривается фундаментальная *задача распределения нагрузки*, в которой несколько серверов должны обработать множество заданий или запросов. Мы сконцентрируемся на базовой версии задачи, в которой все серверы идентичны и каждый сервер может использоваться для обслуживания любых запросов. Эта простая задача демонстрирует некоторые базовые проблемы, которые приходится учитывать при разработке и анализе аппроксимирующих алгоритмов, прежде всего задачи сравнения приближенного решения с оптимальным решением, которое мы не можем эффективно вычислить. Как вы увидите, общая задача распределения нагрузки весьма многогранна, и мы исследуем некоторые из ее аспектов в следующих разделах.

Задача

Задача распределения нагрузки формулируется следующим образом: имеется множество из m машин M_1, ..., M_m и множество из n заданий; каждое задание j характеризуется временем обработки t_j. Требуется назначить каждое задание на одну из машин, чтобы нагрузка по всем машинам была по возможности «сбалансированна».

А конкретнее, в любом распределении заданий по машинам можно обозначить $A(i)$ множество заданий, назначенных на машину M_i; в этом распределении общее время работы машины M_i, равное

$$T_i = \sum_{j \in A(i)} t_j ,$$

называется *нагрузкой* на машину M_i. Мы хотим минимизировать величину, называемую *периодом обработки*, — максимальную нагрузку на каждую машину $T = \max_i T_i$. Хотя здесь этот факт не доказывается, задача нахождения распределения с минимальным периодом обработки является *NP*-сложной.

Разработка алгоритма

Начнем с рассмотрения очень простого жадного алгоритма. Алгоритм перебирает задания в любом порядке за один проход; когда очередь доходит до задания j, оно назначается на машину с минимальной на данный момент нагрузкой.

```
Greedy-Balance:
   В начале назначенных заданий нет
   Присвоить T_i = 0 и A(i) = Ø для всех машин M_i
   For j = 1, ..., n
      Пусть Mi - машина с минимальным min_k T_k
      Назначить задание j на машину M_i
```

```
    Присвоить A(i) ← A(i) ∪ {j}
    Присвоить T_i ← T_i + t_j
  Конец For
```

На рис. 11.1 показан результат выполнения этого жадного алгоритма для последовательности из шести заданий с размерами 2, 3, 4, 6, 2, 2; полученный период обработки равен 8 («высота» заданий на первой машине). Обратите внимание: это решение не является оптимальным; если бы задания поступили в другом порядке и были получены алгоритмом в последовательности 6, 4, 3, 2, 2, 2, то было бы получено назначение с периодом обработки 7.

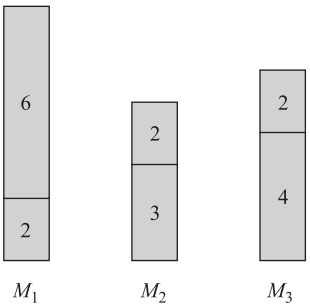


Рис. 11.1. Результат выполнения жадного алгоритма распределения нагрузки на трех машинах с размерами заданий 2, 3, 4, 6, 2, 2

Анализ алгоритма

Обозначим T период обработки полученного присваивания; требуется показать, что T не намного больше минимально возможного периода обработки T^*. Конечно, при этом мы немедленно сталкиваемся с основной проблемой, о которой упоминалось выше: решение должно сравниваться с оптимальным значением T^*, хотя мы не знаем, что это за значение, и не можем рассчитывать на его вычисление. Следовательно, для анализа нам понадобится *нижняя граница* оптимума — величина, которая бы гарантировала, что даже самый лучший оптимум не может быть ниже этой границы.

У оптимума может быть много возможных нижних границ. Одна из идей определения нижней границы основана на анализе общего времени обработки $\sum_j t_j$. Одна из m машин должна выполнить по крайней мере $1/m$ часть общей работы, поэтому имеем следующее:

(11.1) Оптимальный период обработки не менее

$$T^* \geq \frac{1}{m}\sum_j t_j .$$

Впрочем, в одном конкретном случае эта нижняя граница оказывается слишком слабой, чтобы от нее была хоть какая-то практическая польза. Допустим, имеется

одно задание, чрезвычайно длинное по сравнению с суммой всех остальных времен обработки. В достаточно экстремальном варианте оптимальное решение будет заключаться в том, чтобы разместить это задание на отдельной машине, которая последней завершит работу. В таком случае наш жадный алгоритм действительно выдаст оптимальное решение, но нижняя граница в (11.1) для этого недостаточно сильна.

Из этого следует усиленная нижняя граница для T^*.

(11.2) Оптимальный период обработки не менее $T^* \geq \max_j t_j$.

Теперь мы готовы к оценке распределения заданий, построенного нашим жадным алгоритмом.

(11.3) Алгоритм *Greedy-Balance* строит распределение заданий между машинами с периодом обработки $T \leq 2T^*$.

Доказательство. Общий план доказательства: в ходе анализа аппроксимирующего алгоритма полученное решение сравнивается с тем, что нам известно об оптимуме — в данном случае это нижние границы (11.1) и (11.2). Рассмотрим машину M_i, которой досталась максимальная нагрузка T в этом назначении, и спросим себя: какое последнее задание было назначено на машину M_i? Если задание t_j было не слишком большим по сравнению с другими заданиями, то мы находимся не слишком далеко от нижней границы (11.1). А если задание t_j было очень большим, то мы можем использовать (11.2). Логика рассуждений изображена на рис. 11.2.

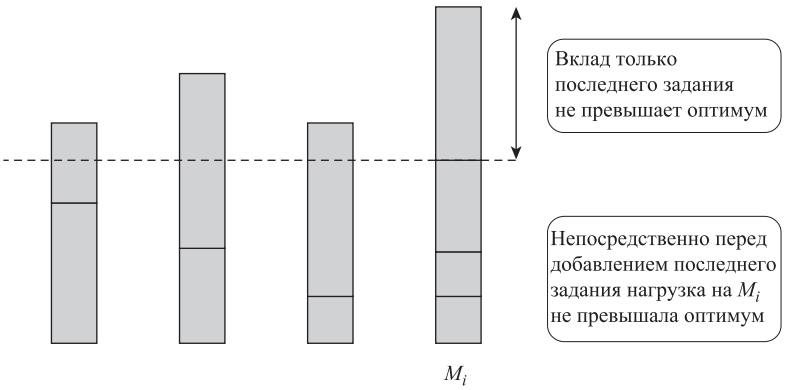


Рис. 11.2. Анализ нагрузки на машине M_i состоит из двух фаз: последнее добавляемое задание и все остальные

А вот как формализовать эти рассуждения: при назначении задания j на машину M_i последняя обладает наименьшей нагрузкой среди всех машин; это ключевое свойство нашего жадного алгоритма. Ее нагрузка непосредственно перед присваиванием составляла $T_i - t_j$, а поскольку она была наименьшей на тот момент, из этого следует, что каждая машина имела нагрузку не менее $T_i - t_j$. Суммируя нагрузки всех машин, имеем $\sum_k T_k \geq m\left(T_i - t_j\right)$, или, эквивалентно,

$$T_i - t_j \leq \frac{1}{m}\sum_k T_k \ .$$

Но значение $\sum_k T_k$ — это всего лишь суммарная нагрузка по всем заданиям $\sum_j T_j$ (так как каждое задание назначено ровно на одну машину), поэтому величина в правой части неравенства в точности совпадает с нижней границей оптимального значения из (11.1). Следовательно,

$$T_i - t_j \le T^*.$$

Затем мы учитываем оставшуюся часть нагрузки M_i, которая в точности определяется последним заданием j. Здесь мы просто используем другую нижнюю границу (11.2), согласно которой $t_j \le T^*$. Суммируя эти два неравенства, мы видим, что

$$T_i = (T_i - t_j) + t_j \le 2T^*.$$

Так как наш период обработки T равен T_i, мы приходим к искомому результату. ■

Нетрудно привести пример, в котором решение отличается от оптимального почти в 2 раза. Допустим, имеются m машин и $n = m(m-1) + 1$ заданий. Каждое из первых $m(m-1) = n-1$ заданий требует времени $t_j = 1$. Последнее задание намного больше; оно требует времени $t_n = m$. Как будет работать этот жадный алгоритм с такой последовательностью заданий? Он равномерно распределит первые $n-1$ заданий, а затем добавит огромное задание n к одному из них; полученный период обработки составит $T = 2m - 1$.

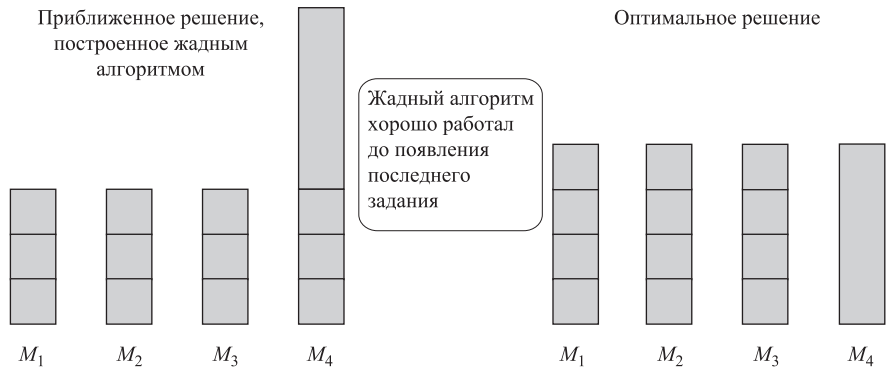


Рис. 11.3. Плохой пример жадного алгоритма распределения нагрузки с $m = 4$

Как должно выглядеть оптимальное решение в этом примере? Оно назначает большое задание на одну из машин (допустим, M_1) и равномерно распределяет остальные задания по остальным $m - 1$ машинам. Так достигается период выполнения m. Следовательно, соотношение между решением жадного алгоритма и оптимальным решением составляет $(2m - 1)/m = 2 - 1/m$, что близко к 2 при больших значениях m.

Схема для $m = 4$ изображена на рис. 11.3; можно только восхищаться изощренностью этой конструкции, которая сбивает с толку жадный алгоритм и заставляет его идеально сбалансировать всю нагрузку только для того, чтобы все разрушить последним большим заданием.

При некотором старании анализ (11.3) можно улучшить и показать, что результат жадного алгоритма с m машинами всегда лежит в пределах множителя $2 - 1/m$ для любого экземпляра; приведенный выше пример плох настолько, насколько это возможно.

Расширения: улучшенный аппроксимирующий алгоритм

Давайте подумаем, как можно создать улучшенный аппроксимирующий алгоритм — другими словами, алгоритм, для которого решение всегда гарантированно превышает оптимум не более чем в 2 раза. Для этого стоит подумать о худших случаях текущего аппроксимирующего алгоритма. Предыдущий плохой пример строился по следующей схеме: задания по возможности равномерно распределяются между машинами в предположении, что ущерб от последующих маленьких заданий будет невелик.

Теперь проанализируем разновидность жадного алгоритма, которая сначала сортирует задания по убыванию времени обработки, а затем действует так же, как прежде. Мы докажем, что полученное распределение имеет период обработки, превосходящий оптимум не более чем в 1,5 раза.

```
Sorted-Balance:
   В начале назначенных заданий нет
   Присвоить T_i = 0 и A(i) = Ø для всех машин M_i
   Отсортировать задания в порядке убывания времени обработки t_j
   Предполагается, что t_1 ≥ t_2 ≥ ... ≥ t_n
   For j = 1, ..., n
      Пусть Mi — машина с минимальным min_k T_k
      Назначить задание j на машину M_i
      Присвоить A(i) ← A(i) ∪ {j}
      Присвоить T_i ← T_i + t_j
   Конец For
```

Улучшение происходит от следующего наблюдения: если заданий меньше m, то жадное решение очевидно будет оптимальным, потому что оно назначает каждое задание на отдельную машину. Если же количество заданий больше m, мы можем использовать следующую нижнюю границу оптимума:

(11.4) Если заданий больше m, то $T^* \geq 2t_{m+1}$.

Доказательство. Рассмотрим только первые $m + 1$ заданий в порядке сортировки. Каждое из них выполняется за время не менее t_{m+1}. Всего существуют $m + 1$ заданий и только m машин, поэтому должна быть машина, которой назначаются два задания. У этой машины время обработки составит не менее $2t_{m+1}$. ■

(11.5) Алгоритм *Sorted-Balance* строит распределение заданий между машинами с периодом обработки $T \leq \frac{3}{2}T^*$.

Доказательство. Оно очень похоже на анализ предыдущего алгоритма. Как и прежде, рассмотрим машину M_i с максимальной нагрузкой. Если на M_i назначено только одно задание, то расписание оптимально. Итак, предположим, что машина M_i содержит минимум два задания, а t_j — последнее задание, назначенное на эту машину. Заметим, что $j \geq m + 1$, потому что алгоритм назначит первые m заданий на m разных машин. Следовательно, $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$, где второе неравенство — (11.4).

Дальнейшее следует по образцу доказательства (11.3), с единственным изменением. Тогда в конце доказательства мы имели неравенства $T_i - t_j \leq T^*$ и $t_j \leq T^*$, которые суммировались для получения множителя 2. Но на этот раз второе неравенство имеет вид $t_j \leq \frac{1}{2}T^*$, поэтому суммирование дает границу $T_i \leq \frac{3}{2}T^*$. ■

11.2. Задача о выборе центров

Задача о выборе центров, как и задача из предыдущего раздела, также относится к общей категории задач по распределению работы между серверами. Центральное место в ней занимает вопрос о том, где лучше разместить серверы; чтобы формулировка оставалась простой и понятной, мы не будем включать в задачу концепцию распределения нагрузки. Задача о выборе центров также дает пример случая, в котором наиболее естественный жадный алгоритм может привести к сколь угодно плохому решению, тогда как слегка измененный жадный метод гарантированно приводит к почти оптимальному решению.

Задача

Рассмотрим следующий сценарий: имеется множество S из n мест — допустим, n маленьких городков в сельской части штата Нью-Йорк. Требуется выбрать k мест для строительства больших торговых комплексов. Предполагается, что население каждого из n городков будет посещать один из торговых комплексов, поэтому места для строительства k торговых комплексов должны находиться в центральных точках.

Начнем с формального определения входных данных нашей задачи. Имеется целое число k, множество S из n мест (соответствующих городкам) и функция расстояния. Для экземпляров задачи, в которых местами являются точки на плоскости, функция расстояния будет стандартным евклидовым расстоянием между точками, а любая точка на плоскости может рассматриваться для размещения центра. Тем не менее разработанный нами алгоритм может применяться и для более общей концепции расстояния. На практике под расстоянием часто понимается расстояние по прямой, но также может пониматься время перемещения от точки s в точку z или расстояние вдоль дороги и даже расходы на перемещение. Допускается любая функция расстояния, обладающая следующими естественными свойствами:

- $dist(s, s) = 0$ для всех $s \in S$;
- симметричность: $dist(s, z) = dist(z, s)$ для всех $s, z \in S$;
- неравенство треугольника: $dist(s, z) + dist(z, h) \geq dist(s, h)$.

Первое и третье свойства присущи практически всем естественным понятиям расстояния. И хотя существуют ситуации с асимметричными расстояниями, большинство случаев, представляющих интерес, также удовлетворяет второму свойству. Наш жадный алгоритм применяется к любой функции расстояния, обладающей этими тремя свойствами, и зависит от всех трех.

Затем необходимо прояснить, что же подразумевается под размещением «в центре». Пусть C — множество центров. Предполагается, что жители каждого городка посещают ближайший торговый комплекс. Это подсказывает, что расстояния от места s до центров должны определяться в виде $dist(s, C) = \min_{c \in C} dist(s,c)$. Множество C образует r-покрытие, если каждое место находится на расстоянии не более r от одного из центров, то есть если $dist(s,C) \leq r$ для всех мест $s \in S$. Минимальное значение r, для которого C образует r-покрытие, называется *радиусом покрытия* C и обозначается $r(C)$. Другими словами, радиус покрытия множества центров C определяется как максимальное расстояние, которое необходимо проехать, чтобы попасть к ближайшему центру. Нашей целью является выбор множества C из k центров, для которых величина $r(C)$ минимальна.

Разработка и анализ алгоритма

Недостатки простого жадного алгоритма

Для начала обсудим жадные алгоритмы для этой задачи. Как и прежде, понятие «жадности» по необходимости получается несколько размытым; фактически мы рассматриваем алгоритмы, которые перебирают места одно за другим «недальновидно», то есть при выборе каждого места не учитывается, куда попадут остальные.

Пожалуй, самый простой жадный алгоритм работает так: первый центр размещается в месте, лучше всего подходящем для одного центра, после чего центры добавляются для сокращения радиуса покрытия — каждый раз настолько, насколько это возможно. Как выясняется, такой подход излишне упрощен, чтобы быть эффективным: в некоторых случаях он приводит к очень плохим решениям.

Чтобы убедиться в этом, рассмотрим пример с двумя местами s и z и $k = 2$. Допустим, s и z располагаются на плоскости — на расстоянии, равном стандартному евклидовому расстоянию на плоскости, а любая точка плоскости подходит для размещения центра. Обозначим d расстояние между s и z. В этом случае лучшее расположение для одного центра c_1 находится в середине пути между s и z, а радиус покрытия этого центра равен $r(\{c_1\}) = d/2$. Жадный алгоритм разместит первый центр в точке c_1. Неважно, где бы ни был размещен второй центр, по крайней мере для одного из мест s и z центр c_1 окажется ближе, поэтому радиус покрытия множества двух центров останется равным $d/2$. При этом в оптимальном решении с $k = 2$

центры размещаются в s и z, что приведет к радиусу покрытия 0. В более сложном примере, демонстрирующем ту же проблему, используются два плотных «скопления» мест возле s и z. Предложенный нами жадный алгоритм начнет с размещения центра в середине между скоплениями, тогда как в оптимальном решении создается отдельный центр для каждого скопления.

Знать оптимальный радиус полезно

Поиски улучшенного алгоритма мы начнем с полезного мысленного эксперимента. Представьте на минуту, что вам кто-то подсказал значение оптимального радиуса r. Пригодится ли эта информация? То есть предположим, что нам известно, что существует множество из k центров C^* с радиусом $r(C^*) \leq r$ и мы хотим найти некоторое множество из k центров из C, радиус покрытия которых ненамного больше r. Как выясняется, найти множество из k центров с радиусом покрытия, не превышающим $2r$, относительно несложно.

Идея заключается в следующем: мы можем использовать факт существования этого решения C^* в своем алгоритме несмотря на то, что мы не знаем, как выглядит C^*. Рассмотрим любое место $s \in S$. Должен существовать центр $c^* \in C^*$, покрывающий место s, и этот центр c^* находится на расстоянии не более r от s. Теперь в качестве центра в нашем решении выбирается это место s вместо c^*, потому что мы понятия не имеем, что собой представляет c^*. Нам хотелось бы сделать так, чтобы из s покрывались все места, покрываемые c^* в неизвестном решении C^*. Задача решается увеличением радиуса с r до $2r$. Все места, находившиеся на расстоянии не более r от центра c^*, находятся на расстоянии не более $2r$ от s (это следует из неравенства треугольника). Простая иллюстрация этого аргумента представлена на рис. 11.4.

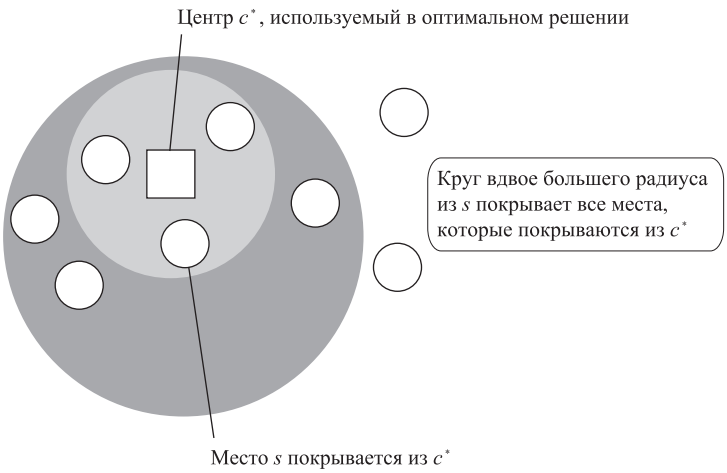


Рис. 11.4. Все места, покрываемые с радиусом r из c^*, также покрываются с радиусом $2r$ из s

```
S′ будет представлять все места, для которых еще нужно обеспечить покрытие
Инициализировать S′ = S
Присвоить C = Ø
Пока S′ = Ø
   Выбрать любое место s ∈ S′ и добавить s в C
   Удалить из S′ все места, находящиеся на расстоянии не более 2r от s
Конец Пока
Если |C| ≤ k
   Вернуть C как выбранное множество мест
Иначе
   Сообщить (обоснованно) о том, что множества из k центров
      с радиусом покрытия не более k не существует
Конец Если
```

Очевидно, если этот алгоритм возвращает множество, содержащее не более k центров, мы получаем желаемый результат.

(11.6) У любого множества центров C, возвращаемого алгоритмом, радиус покрытия $r(C) \leq 2r$.

Затем необходимо обосновать, что если алгоритм не возвращает множество центров, то его заключение об отсутствии множества с радиусом покрытия не более r действительно является верным:

(11.7) Предположим, алгоритм выбирает более k центров. Тогда для любого множества C^* с размером не более k радиус покрытия $r(C^*) > r$.

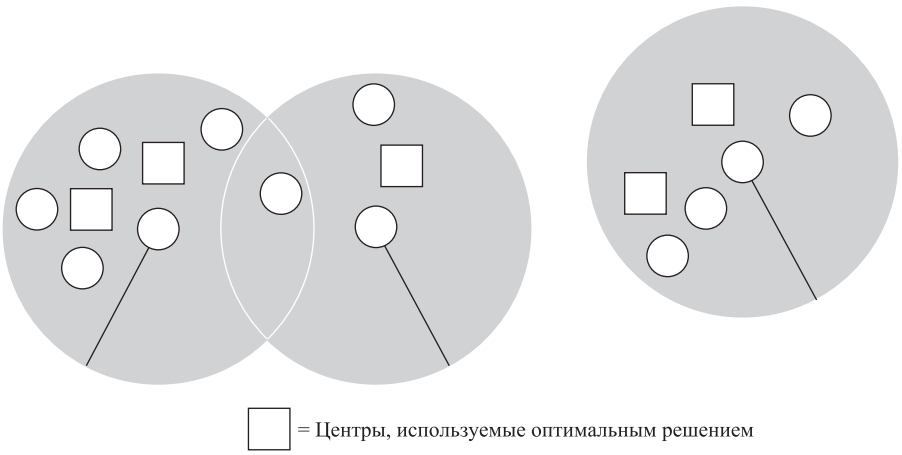


Рис. 11.5. Критический шаг в анализе жадного алгоритма, которому известен оптимальный радиус r. Никакой центр, используемый оптимальным решением, не может принадлежать двум разным кругам, поэтому количество оптимальных центров должно быть по крайней мере не меньше количества центров, выбранных жадным алгоритмом

Доказательство. Предположим обратное: существует множество C^*, содержащее не более k центров, с радиусом покрытия $r(C^*) \leq r$. Каждый центр $c \in C$,

выбираемый жадным алгоритмом, является одним из исходных мест в S, и множество C^* имеет радиус покрытия не более r, поэтому должен быть центр $c^* \in C^*$, находящийся от c на расстоянии не более r, то есть $dist(c, c^*) \leq r$. Мы будем называть такой центр c^* *близким к* c. Нам хотелось бы доказать, что ни один центр c^* в оптимальном решении C^* не может быть близким к двум разным центрам в жадном решении C. Если это удастся, работа закончена: у каждого центра $c \in C$ имеется близкий оптимальный центр $c^* \in C^*$, и все эти близкие оптимальные центры различны. Из этого следует, что $|C^*| \geq |C|$, а поскольку $|C| > k$, это противоречит нашему предположению о том, что C^* содержит не более k центров.

Итак, нужно лишь показать, что ни один оптимальный центр $c^* \in C$ не может быть близким к каждому из двух центров $c, c' \in C$. Причина изображена на рис. 11.5: каждая пара центров $c, c' \in C$ разделяется расстоянием более $2r$, поэтому нахождение c^* на расстоянии не более r от каждого из них приведет к нарушению неравенства треугольника), поскольку $dist(c, c^*) + dist(c^*, c') \geq \geq dist(c, c') > 2r$. ■

Снятие предположения об известном оптимальном радиусе

Вернемся к исходному вопросу: как выбрать хорошее множество из k центров, если оптимальный радиус покрытия неизвестен?

Стоит рассмотреть два разных ответа на этот вопрос. Во-первых, при разработке аппроксимирующих алгоритмов часто бывает концептуально полезно предположить, что значение, обеспечиваемое оптимальным решением, известно. В таких ситуациях часто можно начать с алгоритма, спроектированного с учетом этого предположения, и преобразовать его в алгоритм, обеспечивающий сравнимые гарантии производительности простым перебором по диапазону «предположений» относительно оптимального значения. В ходе выполнения алгоритма эта последовательность предположений становится все более и более точной, пока не будет достигнуто приближенное решение.

Для задачи о выборе центров это может происходить так: мы начинаем с очень слабых исходных предположений относительно радиуса оптимального решения: известно, что оно больше 0 и что оно не превышает максимального расстояния r_{max} между любыми двумя местами. Итак, сначала мы делим разность между ними надвое и применяем разработанный выше жадный алгоритм со значением $r = r_{max}/2$. Затем в соответствии со структурой алгоритма происходит одно из двух: либо мы находим множество из k центров с радиусом покрытия не более $2r$, либо заключаем, что решения с радиусом покрытия не более r не существует. В первом случае можно понизить оценку радиуса оптимального решения; во втором она должна быть повышена. Так по радиусу проводится своего рода «бинарный поиск»: алгоритм итеративно определяет значения $r_0 < r_1$, чтобы мы знали, что оптимальный радиус больше r_0, но решение имеет радиус не более $2r_1$. По этим значениям описанный выше алгоритм выполняется с радиусом $r = (r_0 + r_1)/2$; затем мы либо решаем, что радиус оптимального решения больше $r > r_0$, либо получаем решение с радиусом

не более $2r = (r_0 + r_1) < 2r_1$. В любом случае оценка уточняется с одной или с другой стороны так же, как это происходит при бинарном поиске. Алгоритм прекращает работу, когда оценки r_0 и r_1 оказываются достаточно близкими друг к другу; в этой точке наше решение с радиусом $2r_1$ близко к 2-аппроксимации оптимального радиуса, так как мы знаем, что оптимальное решение больше r_0 (а следовательно, близко к r_1).

Работающий жадный алгоритм

В конкретном случае задачи о выборе центров существует неожиданный способ обойти предположение об известном радиусе, не прибегая к общему методу, описанному выше. Оказывается, жадный алгоритм в практически неизменном виде можно выполнить, ничего не зная о значении r.

Предыдущий жадный алгоритм, располагая информацией о r, многократно выбирает одно из исходных мест s как следующий центр; при этом он убеждается в том, что это место находится на расстоянии не менее $2r$ от всех ранее выбранных мест. Чтобы добиться практически того же эффекта, не зная r, можно просто выбрать место s, самое дальнее от всех ранее выбранных центров: если существуют места, находящиеся на расстоянии не менее $2r$ от всех ранее выбранных мест, то это самое дальнее место s должно быть одним из них. Ниже приведено описание алгоритма.

```
Предположить, что k ≤ |S| (иначе определить C = S)
Выбрать любое место s и присвоить C = {s}
Пока |C| < k
   Выбрать место s ∈ S, максимизирующее dist(s, C)
   Добавить место s в C
Конец Пока
Вернуть C как выбранное множество мест
```

(11.8) Этот жадный алгоритм возвращает множество C из k точек — таких, что $r(C) \le 2r(C^*)$, где C^* — оптимальное множество из k точек.

Доказательство. Обозначим $r = r(C^*)$ минимально возможный радиус множества из k центров. Предположим, что можно получить множество из k центров C с $r(C) > 2r$, и придем к противоречию.

Пусть s — место, находящееся на расстоянии более $2r$ от каждого центра в C. Рассмотрим промежуточную итерацию в ходе выполнения цикла, к моменту которой было выбрано множество центров C'. Предположим, в этой итерации добавляется центр c'. Утверждается, что c' находится на расстоянии не менее $2r$ от всех мест в C'. Это следует из того, что место s находится на расстоянии более $2r$ от всех мест в большем множестве C, а мы выбираем место c, самое дальнее от всех ранее выбранных центров. Формально имеется следующая цепочка неравенств:

$$dist(c', C') \ge dist(s, C') \ge dist(s, C) > 2r.$$

Следовательно, наш жадный алгоритм правильно реализует первые k итераций цикла *Пока* предыдущего алгоритма, в котором был известен оптимальный радиус r: при каждой итерации добавляется центр, находящийся на расстоянии

более $2r$ от всех ранее выбранных центров. Но у предыдущего алгоритма после выбора k центров $S' \neq \varnothing$, так как $s \in S'$, поэтому он продолжит работу, выберет более k центров и в конечном итоге придет к заключению, что k центров не могут иметь радиус покрытия не более r. Это противоречит нашему выбору r, и такое противоречие доказывает, что $r(C) \leq 2r$. ■

Обратите внимание на удивительный факт: наш итоговый жадный 2-аппроксимирующий алгоритм представляет собой очень простую модификацию первого (неработающего) жадного алгоритма. Вероятно, самое важное изменение заключается в том, что наш алгоритм всегда выбирает в качестве центров готовые места (то есть торговый комплекс будет построен в одном из городков, а не на середине пути между ними).

11.3. Покрытие множества: обобщенная жадная эвристика

В этом разделе рассматривается общая задача, которая уже встречалась в главе 8, — задача покрытия множества. Некоторые важные алгоритмические задачи формулируются как особые случаи задачи покрытия множества, поэтому аппроксимирующий алгоритм для этой задачи найдет широкое применение. Мы покажем, что возможно разработать жадный алгоритм, который создает решения с гарантированным коэффициентом аппроксимации относительно оптимума, хотя этот коэффициент будет слабее тех, что мы видели в задачах из разделов 11.1 и 11.2.

Хотя жадный алгоритм, который мы построим для задачи покрытия множества, будет очень простым, его анализ сложнее приводившегося в двух предыдущих разделах. Тогда мы могли обойтись очень простыми границами для (неизвестного) оптимального решения, тогда как здесь задача сравнения с оптимумом более сложна, и нам потребуются более сложные границы. Этот аспект метода можно рассматривать как первый пример метода назначения цены, который будет более полно рассмотрен в двух следующих разделах.

Задача

Как вы помните из главы, посвященной *NP*-полноте, задача покрытия множества базируется на множестве U из n элементов и списке $S_1, ..., S_m$ подмножеств U; *покрытием множества* называется совокупность этих множеств, объединение которых равно всему множеству U.

В версии задачи, которая рассматривается здесь, с каждым множеством S_i вес $w_i \geq 0$. Требуется найти покрытие множества C с минимальным общим весом

$$\sum_{S_i \in C} w_i.$$

Стоит заметить, что по сложности эта задача по крайней мере не уступает версии задачи покрытия множества с принятием решения, которая встречалась

ранее; если назначить все $w_i = 1$, то минимальный вес покрытия множества не превосходит k в том, и только в том случае, если существует совокупность не более чем k множеств, покрывающая U.

Разработка алгоритма

Мы разработаем и проанализируем жадный алгоритм для этой задачи. Алгоритм должен обладать тем свойством, что он строит покрытие по одному множеству; чтобы выбрать следующее множество, он ищет то, которое, насколько можно судить, обеспечивает наибольший прогресс по направлению к цели. Как естественно определить «прогресс» в этой конфигурации? Предпочтительные множества обладают двумя свойствами: они имеют малый вес w_i и покрывают много элементов. Тем не менее ни одного из этих свойств в отдельности не достаточно для построения хорошего алгоритма аппроксимации. Вместо этого будет естественно объединить эти два критерия в одну метрику $w_i/|S_i|$: то есть, выбирая S_i, мы покрываем $|S_i|$ элементов за стоимость w_i; соответственно это отношение определяет условную «цену покрытия элемента» — весьма разумная метрика для выбора.

Конечно, после того, как некоторые множества уже были выбраны, нас интересует лишь то, что происходит с еще не покрытыми элементами. Соответственно мы будем вести множество R оставшихся непокрытых элементов и выберем множество S_i, минимизирующее $w_i/|S_i \cap R|$.

```
Greedy-Set-Cover:
Изначально R = U, и выбранных множество нет
Пока R ≠ Ø
   Выбрать множество S_i, минимизирующее w_i / |S_i ∩ R|
   Удалить множество S_i из R
Конец Пока
Вернуть выбранные множества
```

Пример поведения этого алгоритма представлен на рис. 11.6. Сначала алгоритм выбирает множество из четырех нижних узлов (так как оно имеет лучшее отношение вес/покрытие 1/4). Затем выбирается множество, состоящее из двух узлов во второй строке, и, наконец, множество из двух отдельных узлов наверху. Таким образом, будет выбрана совокупность множество с общим весом 4. Так как алгоритм каждый раз «близоруко» выбирает лучший вариант, он упускает тот факт, что все узлы можно покрыть с весом всего $2 + 2\varepsilon$ — для этого достаточно выбрать два множества, каждое из которых покрывает целый столбец.

Анализ алгоритма

Множества, выбираемые алгоритмом, очевидно образуют покрытие. Нас интересует вопрос: насколько вес этого покрытия превышает вес w^* оптимального покрытия?

Как и в разделах 11.1 и 11.2, для нашего анализа потребуется хорошая нижняя граница оптимума. В задаче о распределении нагрузки мы использовали нижние

границы, которые естественно вытекали из постановки задачи: среднюю нагрузку и максимальный размер задания. Задача о покрытии множества оказывается более сложной; «простые» нижние границы не особенно полезны, вместо них мы используем нижнюю границу, которая неявно строится жадным алгоритмом и является своего рода «побочным результатом».

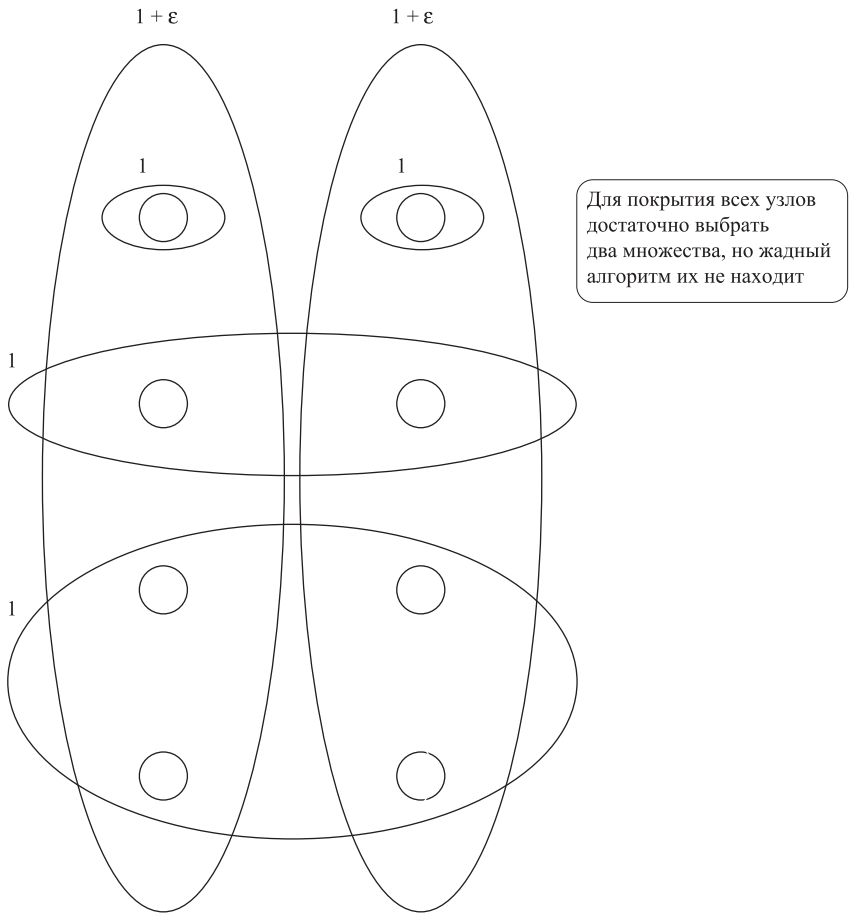


Рис. 11.6. Экземпляр задачи покрытия множества, в котором веса множеств равны либо 1, либо 1 + ε для некоторого малого ε > 0. Жадный алгоритм выбирает множества с общим весом 4 вместо оптимального решения с весом 2 + 2ε

Вспомните интуитивный смысл отношения $w_i / |S_i \cap R|$, используемого алгоритмом, — это «стоимость», выплачиваемая за покрытие каждого нового элемента. Сохраним эту стоимость, выплаченную за элемент s, в величине c_s. Добавим следующую строку в алгоритм сразу же после выбора множества S_i:

Определить $c_s = w_i / |S_i \cap R|$ для всех $s \in S_i \cap R$

Значения c_s никак не влияют на поведение алгоритма; это своего рода промежуточный результат, который упрощает сравнение с оптимальным w^*. При выборе

каждого множества S_i его вес распределяется по стоимостям c_s последних покрытых элементов. Таким образом, эти стоимости отражают общий вес покрытия множества, и мы имеем

(11.9) Если C — покрытие множества, построенное процедурой `Greedy-Set-Cover`, то $\sum_{S_i \in C} w_i = \sum_{s \in U} c_s$.

Ключевую роль в анализе играет следующий вопрос: за какую общую стоимость может «отвечать» любое отдельное множество S_k, другими словами, предоставление границы для $\sum_{s \in S_k} c_s$ относительно веса w_k множества (даже для множеств, не выбранных жадным алгоритмом). Определение для отношения

$$\frac{\sum_{s \in S_k} c_s}{w_k}$$

верхней границы, выполняемой для каждого множества, фактически означает: «Чтобы покрыть большую стоимость, необходимо использовать большой вес». Мы знаем, что оптимальное решение должно покрывать полную стоимость $\sum_{s \in U} c_s$ через выбранные им множества; следовательно, граница такого типа устанавливает, что оно должно использовать по крайней мере некоторую величину веса. Это нижняя граница оптимума, что и требуется для нашего анализа.

В нашем анализе будет использоваться *гармоническая функция*

$$H(n) = \sum_{i=1}^{n} \frac{1}{i}.$$

Чтобы понять ее асимптотический размер как функцию n, можно интерпретировать ее как сумму, аппроксимирующую область под кривой $y = 1/x$. На рис. 11.7 показано, как она естественно ограничивается сверху $1 + \int_1^n \frac{1}{x} dx = 1 + \ln n$ и снизу $\int_1^{n+1} \frac{1}{x} dx = \ln (n + 1)$. Следовательно, мы видим, что $H(n) = \Theta(\ln n)$.

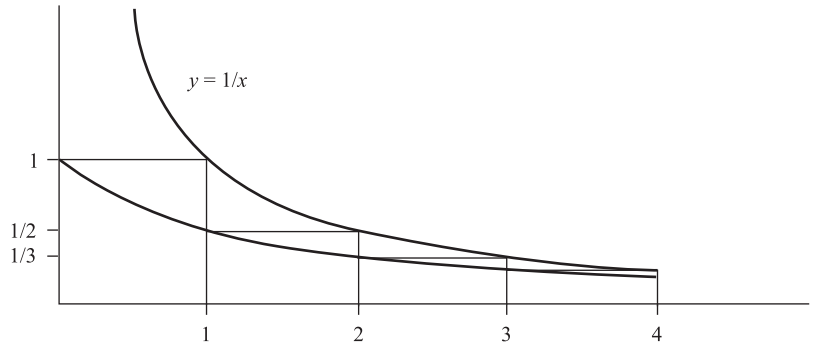


Рис. 11.7. Верхняя и нижняя границы для гармонической функции $H(n)$

Следующее утверждение является ключевым для установления границы для производительности алгоритма.

(11.10) Для каждого множества S_k сумма $\sum_{s \in S_k} c_s$ не превышает $H(|S_k|) \cdot w_k$.

Доказательство. Для упрощения записи будем считать, что элементами S_k являются первые $d = |S_k|$ элементов множества U, то есть $Sk = \{s_1, ..., s_d\}$. Кроме того, предположим, что эти элементы следуют в порядке назначения им стоимости c_{s_j} жадным алгоритмом (с произвольной разбивкой «ничьих»). Такие допущения не приводят к потере общности, так как речь идет о простом переименовании элементов U.

Теперь рассмотрим итерацию, в которой элемент s_j покрывается жадным алгоритмом для некоторого $j \le d$. В начале этой итерации $s_j, s_{j+1}, ..., s_d \in R$ в соответствии с нашей пометкой элементов. Из этого следует, что $|S_k \cap R|$ не меньше $d - j + 1$, а средняя стоимость множества S_k не превышает

$$\frac{w_k}{|S_k \cap R|} \le \frac{w_k}{d - j + 1}.$$

Обратите внимание: это не обязательно равенство, так как элемент s_j может быть покрыт в той же итерации, что и некоторые другие элементы s_j' для $j' < j$. В этой итерации жадный алгоритм выбрал множество S_i с минимальной средней стоимостью; следовательно, средняя стоимость этого множества S_i не превышает среднюю стоимость S_k. Средняя стоимость S_i присваивается s_j, поэтому мы имеем

$$c_{s_j} = \frac{w_i}{|S_i \cap R|} \le \frac{w_k}{|S_k \cap R|} \le \frac{w_k}{d - j + 1}.$$

Теперь просто просуммируем все эти неравенства для всех элементов $s \in S_k$:

$$\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \le \sum_{j=1}^{d} \frac{w_k}{d - j + 1} = \frac{w_k}{d} + \frac{w_k}{d-1} + ... + \frac{w_k}{1} = H(d) \cdot w_k. \blacksquare$$

А теперь завершим план по использованию границы из (11.10) для сравнения покрытия множества жадного алгоритма с оптимальным. Обозначив $d^* = \max_i |S_i|$ максимальный размер любого множества, мы приходим к следующему результату аппроксимации:

(11.11) Вес покрытия C, выбранного процедурой `Greedy-Set-Cover`, превышает оптимальный вес w^* не более чем в $H(d^*)$ раз.

Доказательство. Пусть C^* — оптимальное покрытие множества, такое что $w^* = \sum_{S_i \in C^*} w_i$. Для каждого из множеств в C^* из (11.10) следует

$$w_i \ge \frac{1}{H(d^*)} \sum_{s \in S_i} c_s.$$

Так как эти множества образуют покрытие множества, имеем

$$\sum_{S_i \in C^*} \sum_{s \in S_i} c_s \ge \sum_{s \in U} c_s.$$

Объединяя с (11.9), получаем искомую границу:

$$w^* = \sum_{S_i \in C^*} w_i \geq \sum_{S_i \in C^*} \frac{1}{H(d^*)} \sum_{s \in S_i} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in C} w_i.$$

Тогда асимптотически граница из (11.11) говорит, что жадный алгоритм находит решение с коэффициентом $O(\log d^*)$ от оптимального. Так как максимальный размер множества d^* может выражаться как постоянная доля от общего количества элементов n, верхняя граница для худшего случая равна $O(\log n)$. Однако выражение границы в контексте d^* показывает, что если самое большое множество мало, ситуация значительно улучшается.

Интересно заметить, что эта граница фактически является лучшей из возможных, так как существуют экземпляры, в которых жадный алгоритм работает плохо. Чтобы понять, откуда берутся такие экземпляры, снова рассмотрим пример на рис. 11.6. Предположим, он обобщен так, что базовое множество элементов U состоит из двух высоких столбцов, каждый из которых содержит $n/2$ элементов. Два множества, каждое из которых имеет вес $1 + \varepsilon$ (для некоторого малого $\varepsilon > 0$), покрывают каждый из столбцов по отдельности. Также создадим $\Theta(\log n)$ множеств, которые обобщают структуру других множеств на схеме: одно множество покрывает нижние $n/2$ узлов, другое покрывает следующие $n/4$, третье покрывает следующие $n/8$ и т. д. Каждое из этих множеств имеет вес 1.

Теперь жадный алгоритм в процессе поиска решения с весом $\Theta(\log n)$ будет выбирать множества размера $n/2$, $n/4$, $n/8$, ... С другой стороны, выбор двух множеств, покрывающих столбцы, дает оптимальное решение с весом $2 + 2\varepsilon$. При помощи более сложных конструкций можно усилить пример до получения экземпляров, в которых жадный алгоритм дает вес, очень близкий к произведению оптимального веса и $H(n)$. И еще более сложными средствами было показано, что ни один аппроксимирующий алгоритм с полиномиальным временем не может достичь границы аппроксимации, намного лучшей оптимума, умноженного на $H(n)$, если только не окажется, что $P = NP$.

11.4. Метод назначения цены: вершинное покрытие

Обратимся ко второму общему методу разработки аппроксимирующих алгоритмов: *методу назначения цены*. Для знакомства с этим методом будет рассмотрена версия задачи о вершинном покрытии. Как было показано в главе 8, задача о вершинном покрытии является частным случаем задачи покрытия множества, поэтому сначала мы рассмотрим возможность применения сведения при разработке аппроксимирующих алгоритмов. После этого мы разработаем алгоритм с улучшенными гарантиями аппроксимации по сравнению с обобщенной границей, полученной для задачи покрытия множества в предыдущем разделе.

Задача

Напомним, что вершинным покрытием в графе $G = (V, E)$ называется такое множество $S \subseteq V$, что по крайней мере один конец каждого ребра принадлежит S. В версии задачи, которая рассматривается здесь, каждой вершине $i \in V$ назначается вес $w_i \geq 0$, а вес множества вершин S обозначается $w(S) = \sum_{i \in S} w_i$. Требуется найти вершинное покрытие S, для которого значение $w(S)$ минимально. В стандартной версии задачи о вершинном покрытии все веса равны 1, и требуется принять решение о том, существует ли вершинное покрытие с весом не более k.

Аппроксимация посредством сведения

Прежде чем переходить к разработке алгоритма, мы обсудим один интересный вопрос: задача о вершинном покрытии легко сводится к задаче покрытия множества, для которой мы только что рассмотрели аппроксимирующий алгоритм. Какие выводы можно сделать относительно аппроксимируемости вершинного покрытия? При рассмотрении этого вопроса проявляются некоторые неочевидные аспекты взаимодействия результатов аппроксимации со сведениями с полиномиальным временем.

Начнем с частного случая, в котором все веса равны 1, — иначе говоря, ищется вершинное покрытие минимального размера. Назовем этот случай *невзвешенным*. Вспомните, что для доказательства *NP*-полноты задачи покрытия множества использовалось сведение от невзвешенной версии задачи о вершинном покрытии с принятием решения, то есть

$$\textit{Вершинное покрытие} \leq_P \textit{Покрытие множества}$$

Это сведение означает: «Если бы у нас был алгоритм с полиномиальным временем, решающий задачу покрытия множества, то мы могли бы использовать этот алгоритм для решения задачи о вершинном покрытии за полиномиальное время». Теперь у нас есть алгоритм с полиномиальным временем, возвращающий приближенное решение для задачи покрытия множества. Означает ли это, что алгоритм может использоваться для формулировки аппроксимирующего алгоритма для задачи о вершинном покрытии?

(11.12) Аппроксимирующий алгоритм задачи покрытия множества может использоваться для получения $H(d)$-аппроксимирующего алгоритма для взвешенной задачи о вершинном покрытии, где d — максимальная степень графа.

Доказательство. Оно базируется на сведении, которое демонстрировало, что *Вершинное покрытие* $\leq_P$ *Покрытие множества*; оно также распространяется на взвешенную версию. Рассмотрим экземпляр взвешенной задачи о вершинном покрытии, заданный графом $G = (V, E)$. Экземпляр задачи покрытия множества определяется следующим образом: используемое множество U равно E. Для каждого узла i определяется множество S_i, состоящее из всех ребер, инцидентных узлу i, и этому множеству назначается вес w_i. Совокупности множеств, покрывающие U, точно соответствуют вершинным покрытиям. При этом максимальный размер любого множества S_i точно совпадает с максимальной степенью d.

Следовательно, аппроксимирующий алгоритм для задачи покрытия множества может использоваться для нахождения вершинного покрытия, вес которого лежит в пределах множителя $H(d)$ от минимума. ■

Эта $H(d)$-аппроксимация достаточно хороша при малых d; но с ростом d она ухудшается, приближаясь к границе, логарифмической по количеству вершин. Позднее мы разработаем усиленный аппроксимирующий алгоритм, отличающийся от оптимума не более чем в 2 раза.

Но прежде чем заниматься 2-аппроксимирующим алгоритмом, мы должны отметить один факт: при использовании сведений для разработки аппроксимирующих алгоритмов необходимо действовать очень осторожно. В (11.12) такой подход сработал, но мы специально обосновали, *почему* он сработал; это вовсе не значит, что любое сведение с полиномиальным временем приводит к аналогичному выводу для аппроксимирующего алгоритма.

Рассмотрим поучительный пример: мы использовали задачу о независимом множестве, чтобы доказать *NP*-полноту задачи о вершинном покрытии. А именно, мы доказали

$$\textit{Независимое множество} \leq_P \textit{Вершинное покрытие}$$

Это означает: «Если бы у нас был алгоритм с полиномиальным временем, решающий задачу о вершинном покрытии, то мы могли бы использовать этот алгоритм для решения задачи о независимом множестве за полиномиальное время». Сможем ли мы использовать аппроксимирующий алгоритм для вершинного покрытия с минимальным размером для разработки соизмеримо хорошего аппроксимирующего алгоритма для независимого множества с максимальным размером?

Ответ — нет. Вспомните, что множество I вершин называется независимым в том, и только в том случае, если его дополнение $S = V - I$ является вершинным покрытием. Для заданного вершинного покрытия S^* с минимальным размером мы получим независимое множество с максимальным размером, вычисляя дополнение $I^* = V - S$. Теперь предположим, что мы используем аппроксимирующий алгоритм задачи о вершинном покрытии для получения приближенного минимального вершинного покрытия S. Дополнение $I = V - S$ действительно является независимым множеством — здесь проблем нет. Проблемы появляются тогда, когда мы пытаемся определить коэффициент аппроксимации для задачи о независимом множестве; I может быть очень далеко от оптимума. Предположим, например, что оптимальное вершинное покрытие S^* и оптимальное независимое множество I^* оба имеют размер $|V|/2$. Если выполнить 2-аппроксимирующий алгоритм для задачи о вершинном покрытии, вполне возможно, что мы вернемся к множеству $S = V$. Но в этом случае «приближенное максимальное независимое множество» $I = V - S$ не содержит элементов.

Разработка алгоритма: метод назначения цены

Хотя (11.12) предоставляет аппроксимирующий алгоритм с доказуемыми гарантиями, это не лучший вариант. Следующее описание хорошо демонстрирует применение метода назначения цены при разработке аппроксимирующих алгоритмов.

Применение метода назначения цены для минимизации стоимости

Метод назначения цены (также называемый *прямо-двойственным методом*) основан на экономических представлениях. Для задачи о вершинном покрытии веса мы будем рассматривать веса узлов как *стоимости*, а каждое ребро — как необходимость оплаты его «доли» в стоимости найденного вершинного покрытия. На самом деле мы уже видели пример подобного анализа в жадном алгоритме для покрытия множества из раздела 11.3; он тоже может рассматриваться как алгоритм назначения цены. Жадный алгоритм покрытия множества определял значения c_s — стоимости, оплачиваемые алгоритмом за покрытие элемента s. Значение c_s может рассматриваться как «доля» элемента s в стоимости. Утверждение (11.9) показывает, что модель долевого участия c_s очень естественна, так как сумма долей $\sum_{s \in U} c_s$ равна стоимости покрытия множества C, возвращаемой алгоритмом, $\sum_{S_i \in C} w_i$.

Ключом к доказательству того, что алгоритм является $H(d^*)$-аппроксимирующим, было некоторое приближенное свойство «справедливости» долей стоимости: (11.10) показывает, что оплата за элементы множества S_k превосходит стоимость покрытия их множеством S_k не более чем в $H(|S_k|)$ раз.

В этом разделе мы воспользуемся методом назначения цены на другом примере — на примере задачи о вершинном покрытии. И снова вес w_i вершины i будет рассматриваться как стоимость использования i в покрытии. Каждое ребро e будет рассматриваться как отдельный «агент», желающий «заплатить» что-то покрывающему его узлу. Алгоритм будет не только находить вершинное покрытие S, но и определять цены $p_e \geq 0$ для каждого ребра $e \in E$, так что если каждое ребро $e \in E$ платит цену p_e, в сумме это приблизительно покроет стоимость S. Цены p_e являются аналогами c_s из алгоритма о покрытии множества.

Интерпретация ребер как «агентов» предполагает некоторые естественные правила «справедливости» цен, аналогичные свойству, доказанному (11.10). Прежде всего, выбор вершины i покрывает все ребра, инцидентные i, поэтому будет «несправедливо» взимать за эти инцидентные ребра оплату, в сумме превосходящую стоимость вершины i. Цены p_e будут называться *справедливыми*, если для каждой вершины i за ребра, смежные с i, не приходится платить больше стоимости вершины: $\sum_{e=(i,j)} p_e \leq w_i$. Обратите внимание: свойство, доказанное (11.10) для задачи покрытия множества, является приближенным условием справедливости, тогда как в алгоритме вершинного покрытия будет использоваться точная концепция справедливости, определенная здесь. Полезная особенность справедливых цен заключается в том, что они всегда предоставляют нижнюю границу стоимости любого решения.

(11.13) Для каждого вершинного покрытия S^* и любых неотрицательных и справедливых цен p_e выполняется $\sum_{e \in E} p_e \leq w(S^*)$.

Доказательство. Рассмотрим вершинное покрытие S^*. По определению справедливости имеем $\sum_{e=(i,j)} p_e \leq w_i$ для всех узлов $i \in S^*$. Суммируя эти неравенства по всем узлам S^*, получаем

$$\sum_{i \in S^*} \sum_{e=(i,j)} p_e \leq \sum_{e \in S^*} w_i = w(S^*).$$

Теперь выражение в левой части представляет собой сумму слагаемых, каждое из которых представляет собой некоторую цену ребра p_e. Поскольку S^* является вершинным покрытием, каждое ребро e вносит как минимум одно слагаемое p_e в левую часть. Оно может внести в сумму более одного экземпляра p_e, потому что может покрываться S^* с обоих концов; но цены неотрицательные, поэтому сумма в левой части по крайней мере не меньше суммы всех цен p_e. А значит,

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,j)} p_e.$$

Объединяя с предыдущим неравенством, получаем:

$$\sum_{e \in E} p_e \leq w(S^*),$$

как и требовалось. ■

Алгоритм

Целью аппроксимирующего алгоритма является поиск вершинного покрытия одновременно с назначением цен. Алгоритм можно рассматривать как жадный в отношении того, как он назначает цены. Затем цены используются для управления выбором узлов вершинного покрытия.

Узел i называется *насыщенным* (или «оплаченным»), если $\sum_{e=(i,j)} p_e = w_i$.

```
Vertex-Cover-Approx(G,w):
  Присвоить p_e = 0 для всех e ∈ E
  Пока существует ребро e = (i,j), для которого ни i, ни j ненасыщенны
     Выбрать такое ребро e
     Увеличить p_e без нарушения справедливости
  Конец Пока
  Присвоить S множество всех насыщенных узлов
  Вернуть S
```

Для примера рассмотрим выполнение алгоритма для экземпляра на рис. 11.8. Изначально плотных узлов нет; алгоритм решает выбрать ребро (*a, b*). Он может поднять цену, оплачиваемую (*a, b*), до 3; в этот момент узел *b* становится плотным и повышение останавливается. Затем алгоритм выбирает ребро (*a, d*). Оно может поднять цену только на 1, так как в этот момент узел становится насыщенным (так как вес узла *a* равен 4 и он уже инцидентен ребру с оплатой 3). Наконец, алгоритм выбирает ребро (*c, d*). Он может поднять цену, оплачиваемую (*c, d*), на 2, когда узел *d* становится насыщенным. В возникшей ситуации у каждого ребра имеется хотя бы один насыщенный конец, поэтому алгоритм завершается. Насыщенными являются узлы *a*, *b* и *d*; это и есть полученное вершинное покрытие. (Обратите внимание: это вершинное покрытие не минимально; для получения минимального покрытия следовало бы выбрать *a* и *c*.)

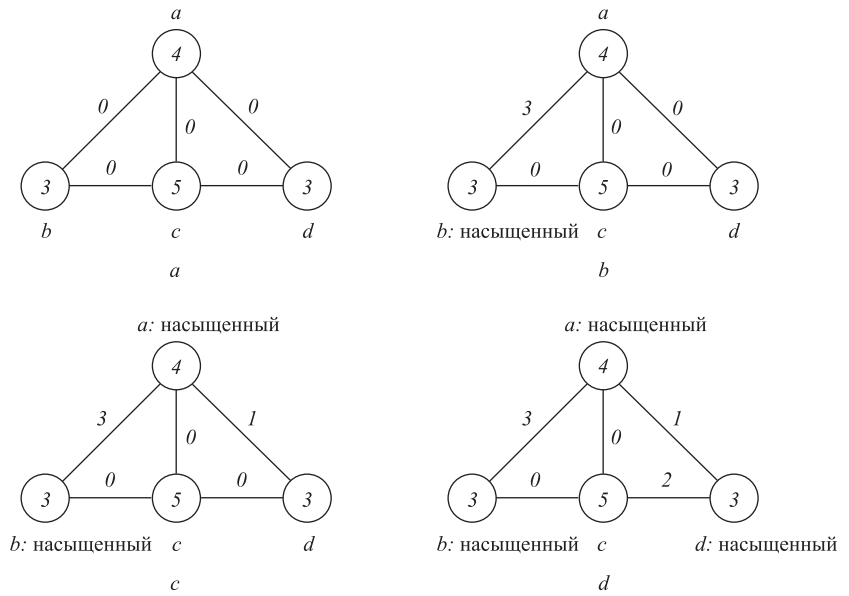


Рис. 11.8. Части (*a*)-(*d*) представляют шаги выполнения алгоритма назначения цены в экземпляре взвешенной задачи о вершинном покрытии. Числа в узлах обозначают веса; числа у ребер обозначают цены, выплачиваемые в ходе выполнения алгоритма

Анализ алгоритма

На первый взгляд может показаться, что вершинное покрытие S полностью оплачивается ценами: все узлы в S насыщенны, а следовательно, ребра, прилегающие к узлу i в S, могут оплачивать стоимость i. Но дело в том, что ребро e может прилегать к более чем одному узлу вершинного покрытия (если оба конца e принадлежат покрытию), и возможно, e придется «платить» более чем за один узел. Например, так обстоит дело с ребрами (a, b) и (a, d) в конце выполнения алгоритма на рис. 11.8.

Однако заметьте, что если взять ребра, для которых оба конца принадлежат вершинному покрытию и мы взимаем их цену дважды, то мы в точности оплачиваем вершинное покрытие. (В примере стоимость вершинного покрытия складывается из стоимости узлов a, b и d, которая равна 10. Эта стоимость складывается из двукратной оплаты (a, b) и (a, d) и однократной оплаты (c, d).) Действительно, для некоторых ребер это несправедливо, но величина несправедливости может быть ограничена: каждое ребро оплачивается не более двух раз (по одному для каждого конца).

Формализуем этот аргумент:

(11.14) Для множества S и цен p, возвращаемых алгоритмом, выполняется неравенство $w(S) \le 2 \sum_{e \in E} p_e$.

Доказательство. Все узлы в S являются насыщенными, поэтому $\sum_{e=(i,j)} p_e = w_i$ для всех $i \in S$. Суммируя по всем узлам в S, получаем

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e.$$

Ребро $e = (i, j)$ может входить в сумму в правой части не более двух раз (если i и j входят в S), поэтому мы получаем

$$w(S) = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq 2 \sum_{e \in E} p_e,$$

как и было заявлено. ■

Наконец, этот множитель 2 переходит в утверждение, дающее гарантии аппроксимации.

(11.15) Множество S, возвращаемое алгоритмом, является вершинным покрытием, а его стоимость не более чем вдвое превышает стоимость минимального вершинного покрытия.

Доказательство. Сначала докажем, что S действительно является вершинным покрытием. Предположим от обратного, что S не покрывает ребро $e = (i,j)$. Из этого следует, что ни i, ни j ненасыщенны, а это противоречит тому факту, что цикл Пока в алгоритме завершился.

Чтобы получить заявленную границу аппроксимации, достаточно объединить (11.14) с (11.13). Пусть p — цены, заданные алгоритмом, а S^* — оптимальное вершинное покрытие. Согласно (11.14), имеем $2\sum_{e \in E} p_e \geq w(S)$, а по (11.13) — $\sum_{e \in E} p_e \leq w(S^*)$.

Другими словами, сумма цен ребер является нижней границей для веса любого вершинного покрытия, а удвоенная сумма цен ребер является верхней границей веса нашего вершинного покрытия:

$$w(S) \leq 2 \sum_{e \in E} p_e \leq 2w(S^*). \blacksquare$$

11.5. Максимизация методом назначения цены: задача о непересекающихся путях

В продолжение темы алгоритмов назначения цены рассмотрим фундаментальную задачу из области сетевой маршрутизации: задачу о непересекающихся путях. Начнем с разработки жадного алгоритма для этой задачи, а затем рассмотрим улучшенный алгоритм, основанный на назначении цены.

Задача

Чтобы сформулировать задачу, полезно вспомнить одно из первых практических применений задачи о максимальном потоке: нахождение непересекающихся путей

в графах (обсуждалось в главе 7). Тогда мы искали пути, не пересекающиеся по ребрам, которые начинались в узле s и заканчивались в узле t. Насколько критично для разрешимости задачи требование о том, чтобы все пути начинались и заканчивались в одном узле? Используя методы из раздела 7.7, алгоритм можно расширить для поиска непересекающихся путей для расширенной конфигурации: имеется множество начальных узлов S и множество конечных узлов T, а задача заключается в нахождении путей, не пересекающихся по ребрам, причем эти пути могут начинаться с любого узла в S и заканчиваться на любом узле из T.

Однако мы рассмотрим случай, при котором каждый путь имеет собственную пару из начального и конечного узлов. А именно, будет рассмотрена следующая задача о нахождении максимальных непересекающихся путей: имеется направленный граф G с k парами узлов (s_1, t_1), (s_2, t_2), ..., (s_k, t_k) и целочисленная пропускная способность c. Каждая пара (s_i, t_i) рассматривается как *запрос на маршрутизацию*, то есть запрос на получение пути от s_i к t_i. Решение экземпляра состоит из подмножества удовлетворяемых запросов, $I \subseteq \{1, ..., k\}$, вместе с путями, которые удовлетворяют их без перегрузки одного ребра; путь P_i для $i \in I$, так что P_i проходит из s_i в t_i, и каждое ребро используется не более чем c путями. Задача заключается в том, чтобы найти решение с наибольшим возможным $|I|$, то есть удовлетворить как можно больше запросов. Обратите внимание: пропускная способность c управляет допустимой степенью «повторного использования» ребер; при $c = 1$ пути должны быть полностью непересекающимися по ребрам, тогда как большие значения c допускают некоторое перекрытие путей.

Мы уже видели в упражнении 39 главы 8, что задача определения возможности удовлетворения всех k заявок маршрутизации для путей, не пересекающихся по узлам, является *NP*-полной. Нетрудно показать, что версия задачи, не пересекающаяся по ребрам (соответствующая случаю $c = 1$), также является *NP*-полной.

Для применения эффективных алгоритмов сетевого потока критично, чтобы конечные точки путей не образовывали явно заданные пары, как в задаче о максимальных непересекающихся путях. Чтобы этот момент стал более понятным, предположим, что мы пытаемся свести задачу максимальных непересекающихся путей к задаче о потоке в сети: определим множество источников $S = \{s_1, s_2, ..., s_k\}$, множество стоков $T = \{t_1, t_2, ..., t_k\}$, пропускную способность каждого ребра c и попробуем найти максимально возможное число непересекающихся путей, которые начинаются в S и заканчиваются в T. Почему из этого ничего не выйдет? Дело в том, что алгоритму потока невозможно сообщить, что путь, начинающийся в $s_i \in S$, должен заканчиваться в $t_i \in T$; алгоритм гарантирует лишь то, что путь закончится в некотором узле из множества T. В результате пути, сформированные на выходе алгоритма потока, могут не быть решением экземпляра задачи о максимальных непересекающихся путях, потому что они могут не связывать источник s_i с соответствующей конечной точкой t_i.

Задачи о непересекающихся путях, в которых требуется найти пути, соединяющие заданные пары терминальных узлов, очень часто встречаются в сетевых приложениях. Только представьте себе пути в Интернете, по которым передаются мультимедийные потоки или веб-данные, или пути в телефонной сети, по которым

передается голосовой трафик[1]. Пути с общими ребрами могут мешать работе друг друга, и слишком большое количество путей, проходящих через одно ребро, создаст проблемы. Максимально допустимый уровень совместного использования ребер зависит от конкретного применения. Полный запрет на пересечение путей — самое сильное ограничение, предотвращающее любые помехи между путями. Но, как вы вскоре убедитесь, в тех случаях, в которых допустимо некоторое совместное использование (даже если через ребро проходят всего два пути), возможны более эффективные аппроксимирующие алгоритмы.

Разработка и анализ жадного алгоритма

Сначала мы рассмотрим очень простой алгоритм для пропускной способности $c = 1$, то есть когда пути не должны пересекаться по ребрам. По сути, это жадный алгоритм, не считая того, что он предпочитает короткие пути. Мы покажем, что этот простой алгоритм является $O(\sqrt{m})$-аппроксимирующим алгоритмом, где $m = |E|$ — количество ребер в G. Может показаться, что это довольно большой коэффициент аппроксимации; это действительно так, однако есть довольно веская причина, по которой это фактически лучшее, на что можно рассчитывать. Задача о максимальных непересекающихся путях не только является *NP*-полной, но и плохо аппроксимируется: доказано (если только не окажется, что $P = NP$), что алгоритм с полиномиальным временем не способен достичь границы апрроксимации, существенно лучшей $O(\sqrt{m})$ для произвольных направленных графов.

После разработки жадного алгоритма будет рассмотрен чуть более сложный алгоритм назначения цены для версии с пропускными способностями. Интересно, что алгоритм назначения цены работает намного эффективнее простого жадного алгоритма, даже если пропускная способность c ненамного выше 1.

```
Greedy-Disjoint-Paths:
Присвоить I = Ø
Пока удается найти новые пути
   Пусть Pi — кратчайший путь, не пересекающийся по ребрам
   с ранее выбранными путями и соединяющий пару (s_i,t_i),
   которая еще не была соединена
   Добавить i в I и выбрать путь P_i для соединения s_i с t_i
Конец Пока
```

[1] Один ученый, работающий в области телекоммуникаций, однажды так объяснил различия между задачами максимальных непересекающихся путей и потока в сети и суть нарушения сведения из предыдущего абзаца. В День матери, когда количество звонков традиционно достигает максимума, телефонной компании приходится решать грандиозную задачу непересекающихся путей: следить за тем, чтобы каждый источник s_i был связан путем по голосовой сети со своей матерью t_i. С другой стороны, алгоритмы сетевого потока, ищущие непересекающиеся пути между множеством S и множеством T, обещают лишь то, что каждый человек дозвонится до чьей-то матери.

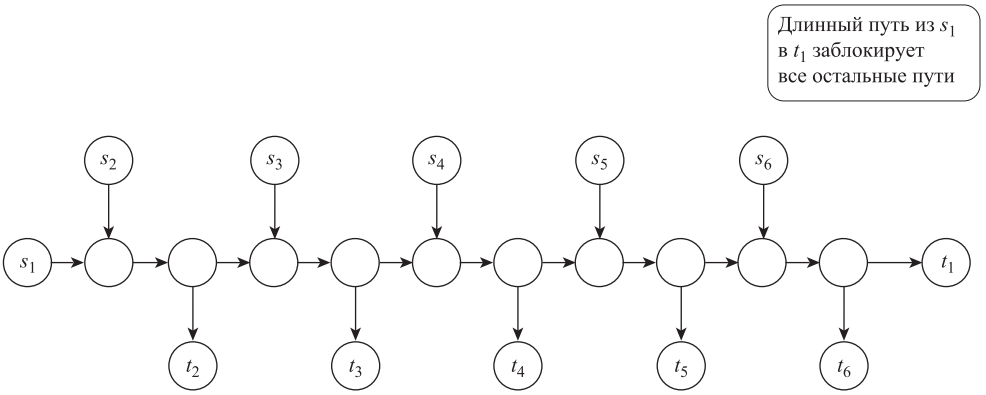


Рис. 11.9. В этом случае критично, чтобы жадный алгоритм выбора непересекающихся путей отдавал предпочтение коротким путям перед длинными

Анализ алгоритма

Понятно, что алгоритм выбирает пути, не пересекающиеся по ребрам. Если предположить, что граф G является связным, он должен выбрать хотя бы один путь. Но как количество выбранных путей сравнивается с максимально возможным? Ситуация, в которой могут возникнуть проблемы, изображена на рис. 11.9: один из путей (из s_1 в t_1) очень длинный, и если выбрать его первым, мы потеряем до $\Omega(m)$ других путей.

Теперь покажем, что предпочтительность коротких путей в жадном алгоритме не только избегает проблему, представленную в этом примере, но и ограничивает количество других путей, с которыми может взаимодействовать выбранный путь.

(11.16) Алгоритм *Greedy-Disjoint-Paths* является $(2\sqrt{m}+1)$-аппроксимирующим алгоритмом для задачи о максимальных непересекающихся путях.

Доказательство. Рассмотрим оптимальное решение: пусть I^* — множество пар, для которых в этом оптимальном решении был выбран путь, а P^*_i для $i \in I^*$ — выбранные пути. Обозначим I множество пар, возвращаемых алгоритмом, а P_i для $i \in I$ — соответствующие пути. Требуется ограничить $|I^*|$ в отношении $|I|$. Ключевую роль в этом анализе играет возможность различать короткие и длинные пути, которые рассматриваются по отдельности. Путь будет считаться *длинным*, если он содержит не менее $\sqrt{m}$ ребер; в противном случае путь считается *коротким*. Пусть I^*_s обозначает множество индексов в I^*, для которого соответствующий путь P^*_i является коротким, а I_s — множество индексов I, для которого соответствующий путь P_i является коротким.

Граф G содержит m ребер, и каждый длинный путь использует не менее $\sqrt{m}$ ребер, так что в I^* может быть не более $\sqrt{m}$ длинных путей.

Теперь рассмотрим короткие пути в I^*. Чтобы множество I^* было намного больше I, должно быть большое количество пар, соединенных в I^*, но не в I. Рассмотрим пары, соединенные оптимумом с использованием короткого пути, но

не соединенные жадным алгоритмом. Так как путь P^*_i, соединяющий s_i и t_i в оптимальном решении I^*, является коротким, жадный алгоритм выбрал бы этот путь, если бы он был доступен, прежде чем выбирать какие-либо длинные пути. Но жадный алгоритм не соединил s_i и t_i, а следовательно, одно из ребер e на пути P^*_i должно входить в путь P_j, выбранный ранее жадным алгоритмом. Мы будем говорить, что ребро *e блокирует* путь P^*_i.

Длины путей, выбранных жадным алгоритмом, монотонно возрастают, так как у каждой итерации остается меньше вариантов выбора путей. Путь P_j был выбран до рассмотрения P^*_i, а следовательно, он должен быть короче: $|P_j| \le |P^*_i| \le \sqrt{m}$. Таким образом, путь P_j является коротким. Так как пути, используемые оптимумом, не пересекаются по ребрам, каждое ребро в пути P_j может блокировать не более одного пути P^*_i. Отсюда следует, что каждый короткий путь P_j блокирует не более $\sqrt{m}$ путей в оптимальном решении, и мы получаем границу

$$\left|I^*_s - I\right| \le \sum_{j \in I_s} \left|P_j\right| \le |I_s|\sqrt{m}.$$

Эта граница будет использована для получения границы общего размера оптимального решения. Для этого множество I^* рассматривается как состоящее из трех типов путей в соответствии с предшествующим анализом:

- длинные пути, которых может быть не более $\sqrt{m}$;
- пути, также входящие в I;
- короткие пути, не входящие в I, которые только что были ограничены $|I_s|\sqrt{m}$.

Объединяя все сказанное, мы используем тот факт, что $|I| \ge 1$ всюду, где может быть соединен по крайней мере один набор терминальных пар, и получаем заявленную границу:

$$\left|I^*\right| \le \sqrt{m} + |I| + \left|I^*_s - I\right| \le \sqrt{m} + |I| + \sqrt{m}|I_s| \le \left(2\sqrt{m}+1\right)|I|. \blacksquare$$

Итак, мы получили аппроксимирующий алгоритм для случая, в котором выбранные пути должны быть непересекающимися. Как упоминалось ранее, граница аппроксимации $O(\sqrt{m})$ достаточно слаба, но если не окажется, что $P = NP$, по сути, это лучший возможный результат для непересекающихся путей в произвольных направленных графах.

Разработка и анализ алгоритма назначения цены

Запрет на использование одного ребра двумя путями — крайний случай; в большинстве практических применений ребра могут использоваться несколькими путями. Сейчас мы разработаем аналогичный алгоритм, базирующийся на методе назначения цены, для случая, когда любое ребро может использоваться в $c > 1$ путей. В только что рассмотренном случае без пересечений все ребра рассматривались как равные, а предпочтение отдавалось коротким путям. Эту схему можно представить как простую разновидность алгоритма назначения цены: пути должны «платить» за использование ребер, и у каждого ребра имеется стоимость. Сейчас будет рас-

смотрена схема назначения цены, в которой ребра считаются более дорогими, если они использовались ранее, а следовательно, у них осталось меньше свободной пропускной способности. При таком подходе алгоритм будет распределять свои пути вместо того, чтобы «сваливать» их на одно ребро. Мы будем называть стоимость ребра e его длиной ℓ_e, а длина пути определяется как сумма длин содержащихся в нем ребер: $\ell(P) = \sum_{e \in P} l_e$. Параметр-множитель β используется для увеличения длины ребра при каждом его использовании очередным путем.

Greedy-Paths-with-Capacity:
Присвоить $I = \emptyset$
Присвоить длину ребра $e = 1$ для всех $e \in E$
Пока удается найти новые пути
  Пусть Pi – кратчайший путь, для которого при добавлении P_i
  в множество путей ни одно ребро не используется более c раз,
  и соединяющий пару (s_i, t_i), которая еще не была соединена
  Добавить i в I и выбрать путь P_i для соединения s_i с t_i
  Умножить длину всех ребер пути P_i на β
Конец Пока

Анализ алгоритма

В ходе анализа мы сосредоточимся на простейшем случае, в котором одно ребро может использоваться не более чем двумя путями, то есть $c = 2$. Будет показано, что в этом случае значение $\beta = m^{1/3}$ обеспечивает лучший результат аппроксимации для алгоритма. В отличие от случая с непересекающимися путями (при $c = 1$) неизвестно, будут ли границы аппроксимации, полученные здесь для $c > 1$, близки к лучшим возможным для алгоритмов с полиномиальным временем вообще (при условии, что $P \neq NP$).

Анализ случая без пересечения базировался на возможности различать «короткие» и «длинные» пути. Для случая $c = 2$ путь P_i, выбранный алгоритмом, будет считаться *коротким*, если его длина менее β^2. Пусть I_s — множество коротких путей, выбранных алгоритмом.

Затем мы хотим сравнить количество выбранных и максимально возможных путей. Пусть I^* — оптимальное решение, а P^*_i — множество путей, использованных в решении. Как и прежде, ключевая идея анализа — рассмотрение ребер, блокирующих выбор путей в I^*. Длинные пути могут заблокировать многие другие пути, поэтому пока мы сосредоточимся на коротких путях в I_s. Однако пытаясь действовать по той же схеме, что и в непересекающемся случае, мы немедленно сталкиваемся с трудностями. В том случае длина пути в I^* определялась количеством содержащихся в нем ребер, а на этот раз длины изменяются в ходе выполнения алгоритма, и теперь неясно, как определить длину пути в I^* в целях анализа. Другими словами, когда измерять эту длину в контексте проводимого анализа? (В начале выполнения? В конце?)

Как выясняется, критическим моментом алгоритма для проводимого анализа является первая точка, в которой не остается ни одного короткого пути для выбора. Пусть $\overline{l}$ — функция длины для текущей точки в выполнении алгоритма; мы

используем $\bar{\ell}$ для измерения длины путей в I^*. Длина пути P, $\sum_{e \in P} \bar{\ell}_e$, будет обозначаться $\bar{\ell}(P)$. Путь P_i^* в оптимальном решении I^* будет считаться *коротким*, если $\bar{\ell}(P_i^*) < \beta^2$, и *длинным* в противном случае. Пусть I_s^* обозначает множество коротких путей в I^*. Прежде всего нужно показать, что не существует коротких путей, соединяющих пары, которые не были соединены аппроксимирующим алгоритмом.

(11.17) Возьмем пару «источник-сток» $i \in I^*$, не соединенную аппроксимирующим алгоритмом; то есть $i \neq I$. Для нее выполняется $\bar{\ell}(P_i^*) \geq \beta^2$.

Доказательство. Пока выбираются короткие пути, нам не придется специально следить за тем, чтобы каждое ребро использовалось не более чем в $c = 2$ путям; любое ребро e, рассматриваемое для включения в третий путь, уже имеет длину $\ell_e = \beta^2$, а следовательно, является длинным.

Рассмотрим состояние алгоритма при длине $\bar{\ell}$. Как следует из предыдущего абзаца, можно считать, что алгоритм отработал до этой точки, не беспокоясь об ограничении c; он просто выбирал короткий путь тогда, когда мог его найти. Так как точки s_i, t_i из P_i^* не соединяются жадным алгоритмом и так как при достижении функцией длины $\bar{\ell}$ коротких путей не остается, из этого следует, что путь P_i^* имеет длину по крайней мере β^2 согласно $\bar{\ell}$. ■

При анализе случая без пересечений тот факт, что количество ребер не превышает m, использовался для ограничения количества длинных путей. На этот раз в качестве величины, потребляемой путями, будет использоваться длина $\bar{\ell}$ вместо количества ребер. Следовательно, для дальнейших рассуждений нам понадобится граница общей длины в графе $\sum_e \bar{\ell}_e$. Сумма длин по всем ребрам $\sum_e \ell_e$ начинается с m (длина 1 для каждого ребра). Добавление короткого пути в решение I_s может увеличить длину не более чем на β^3, так как длина выбранного пути не превышает β^2, а длины ребер увеличиваются с множителем β вдоль пути. Это дает нам полезное сравнение между количеством выбранных коротких путей и общей длиной.

(11.18) Множество I_s коротких путей, выбранных аппроксимирующим алгоритмом, и длины $\bar{\ell}$ находятся в отношении $\sum_e \bar{\ell}_e \leq \beta^3 |I_s| + m$.

Итак, граница аппроксимации для этого алгоритма доказана. Оказывается, даже при простом увеличении количества путей, разрешенных для каждого ребра, с 1 до 2 гарантия аппроксимации уменьшается на значительную величину, которая, по сути, встраивает изменение в экспоненту: с $O(m^{1/2})$ до $O(m^{1/3})$.

(11.19) Алгоритм *Greedy-Paths-with-Capacity* с $\beta = m^{1/3}$ является $(4m^{1/3} + 1)$-аппроксимирующим алгоритмом в случае пропускной способности $c = 2$.

Доказательство. Начнем с ограничения $|I^* - I|$. Согласно (11.17), имеем $\bar{\ell}(P_i^*) \geq \beta^2$ для всех $i \in I^* - I$. Суммируя по всем путям в $I^* - I$, получаем

$$\sum_{i \in I^* - I} \bar{\ell}(P_i^*) \geq \beta^2 |I^* - I|.$$

С другой стороны, каждое ребро в решении I^* используется не более чем двумя путями, поэтому

$$\sum_{j \in I^* - I} \bar{l}\left(P_i^*\right) \leq \sum_{e \in E} 2\bar{l}_e.$$

Объединяя эти границы с (11.18), получаем

$$\beta^2 \left|I^*\right| \leq \beta^2 \left|I^* - I\right| + \beta^2 |I| \leq \sum_{i \in I^* - I} \bar{l}\left(P_i^*\right) + \beta^2 |I| \leq$$
$$\leq \sum_{e \in E} 2\bar{l}_e + \beta^2 |I| \leq 2\left(\beta^3 |I| + m\right) + \beta^2 |I|.$$

Остается разделить на β^2, использовать $|I| \geq 1$ и присвоить $\beta = m^{1/3}$, и мы получаем $|I^*| \leq (4m^{1/3} + 1)|I|$. ■

Этот алгоритм также применим к задаче о непересекающихся путях с произвольной пропускной способностью $c > 0$. Если выбрать $\beta = m^1/(c + 1)$, то алгоритм является $(2cm^{1/(c+1)} + 1)$-аппроксимирующим. При расширении анализа пути считаются короткими, если их длина не превышает β^c.

(11.20) Алгоритм `Greedy-Paths-with-Capacity` с $\beta = m^{1/c+1}$ является $(2cm^{1/(c+1)} + 1)$-аппроксимирующим алгоритмом, если пропускные способности ребер равны c.

11.6. Линейное программирование и округление: применение к задаче о вершинном покрытии

Для начала познакомимся с эффективным методом из области исследования операций: *линейным программированием*. Дисциплина линейного программирования становится темой целых учебных курсов, и мы даже не пытаемся привести здесь сколько-нибудь исчерпывающий обзор. В этом разделе будут представлены некоторые базовые идеи, лежащие в основе линейного программирования, а также продемонстрированы возможности их применения для аппроксимации *NP*-сложных оптимизационных задач.

Напомним, что в разделе 11.4 мы разработали 2-аппроксимирующий алгоритм для взвешенной задачи о вершинном покрытии. В первом примере использования метода линейного программирования мы рассмотрим другой 2-аппроксимирующий алгоритм, который концептуально намного проще первого (хотя и работает медленнее).

Линейное программирование как обобщенный метод

Наш 2-аппроксимирующий алгоритм взвешенной версии задачи о вершинном покрытии будет основан на линейном программировании. Линейное программиро-

вание описывается здесь не для того, чтобы дать общий алгоритм аппроксимации, а чтобы продемонстрировать его мощь и универсальность.

Что же такое линейное программирование? Чтобы ответить на этот вопрос, будет полезно для начала вспомнить из линейной алгебры задачу системы линейных уравнений. В матрично-векторной записи имеется вектор x неизвестных вещественных чисел, заданная матрица A и заданный вектор b; требуется решить уравнение $Ax = b$. Многим хорошо известен *метод исключения Гаусса*, позволяющий эффективно решать такие задачи.

Базовая задача линейного программирования может рассматриваться как более сложная версия этой задачи, в которой равенства заменены неравенствами. Говоря конкретнее, рассмотрим задачу определения вектора x, удовлетворяющего неравенству $Ax \geq b$. Под этой записью мы имеем в виду, что каждая координата вектора Ax должна быть больше либо равна соответствующей координаты вектора b. Такие системы неравенств определяют области в пространстве. Например, предположим, $x = (x_1, x_2)$ — двумерный вектор, и имеются четыре неравенства:

$$x_1 \geq 0,\ x_2 \geq 0$$

$$x_1 + 2x_2 \geq 6$$

$$2x_1 + x_2 \geq 6$$

В этом случае множество решений определяет область на плоскости, изображенную на рис. 11.10. Для заданной области, определяемой уравнением $Ax \geq b$, линейное программирование стремится минимизировать линейную комбинацию координат x по всем x, принадлежащим области.

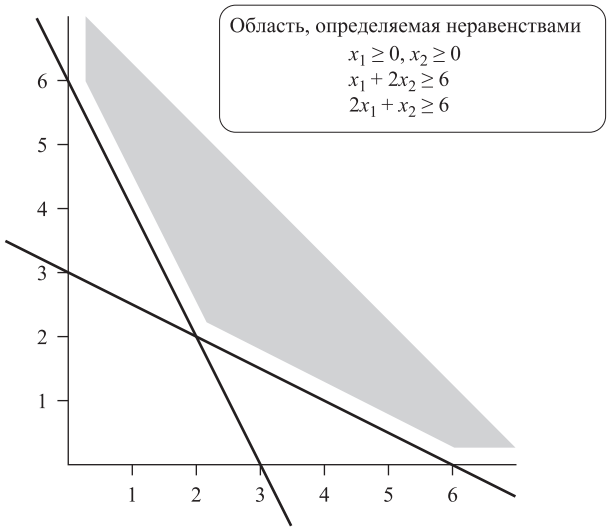


Рис. 11.10. Область допустимых решений простой задачи линейного программирования

Такая линейная комбинация может быть записана в виде $c^t x$, где c — вектор коэффициентов, а $c^t x$ обозначает скалярное произведение двух векторов. Таким образом, стандартная форма линейного программирования как задачи оптимизации выглядит следующим образом.

Для заданной матрицы A с размерами $m \times n$, векторов $b \in R^m$ и $c \in R^n$ найти вектор $x \in R^n$ для решения следующей задачи оптимизации:

$$\min(c^t x \text{ для } x \geq 0; Ax \geq b).$$

$c^t x$ часто называется *целевой функцией* линейной программы, а $Ax \geq b$ называется набором ограничений. Например, в примере на рис. 11.10, допустим, определяется вектор c (1,5, 1); другими словами, мы стремимся минимизировать величину $1{,}5x_1 + x_2$ по области, определяемой неравенствами.

Задача решается выбором точки $x = (2{,}2)$, в которой пересекаются две наклонные линии; в ней $c^t x = 5$, и вы можете убедиться, что получить меньшее значение невозможно.

Задачу линейного программирования можно сформулировать как задачу принятия решения следующим образом:

Для заданной матрицы A, векторов b и c, и границы γ существует ли такое значение x, для которого $x \geq 0$, $Ax \geq b$, и $c^t x \leq \gamma$?

Чтобы избежать проблем, связанных с представлением вещественных чисел, мы будем считать, что координаты в векторах и матрице являются целыми числами.

Вычислительная сложность линейного программирования

Версия задачи линейного программирования с принятием решения принадлежит *NP*. Интуитивно это вполне понятно — нужно лишь представить вектор x, обладающий нужными свойствами. Единственная проблема в том, что даже если все входные числа являются целыми, координаты такого вектора x могут быть нецелыми, и для их представления может потребоваться очень большая точность: откуда известно, что мы сможем читать и обрабатывать эти данные за полиномиальное время? Но на самом деле можно показать, что если решение существует, то существует и рациональное решение, для записи которого достаточно полиномиального количества битов, так что это не проблема.

Кроме того, уже давно известно, что задачи линейного программирования относятся к классу *co-NP*, хотя это и не очевидно. Студенты, прошедшие курс линейного программирования, могут заметить, что этот факт следует из *двойственности* линейного программирования[1].

В течение долгого времени линейное программирование, пожалуй, было самым известным примером задачи, принадлежащей *NP* и *co-NP*, для которой не известно

[1] Читатели, знакомые со свойством двойственности, могут также заметить, что метод назначения цены из предыдущего раздела также работает на основе двойственности линейного программирования: цены точно соответствуют переменным в двойственной задаче линейного программирования (и это объясняет, почему алгоритмы назначения цены часто называются прямо-двойственными алгоритмами).

решение с полиномиальным временем. Затем в 1981 году Леонид Хачиян, который в то время был молодым ученым из СССР, предложил алгоритм с полиномиальным временем для этой задачи. Американская массовая пресса забеспокоилась, не сыграет ли это открытие ту же роль, что и запуск спутника в период холодной войны, но этого не произошло, и вскоре ученые разобрались в том, что же сделал Хачиян. Его исходный алгоритм работал с полиномиальным временем, но был довольно медленным и непригодным для практического применения; но с тех пор после выхода работы Нарендры Кармаркара в 1984 году были разработаны практичные алгоритмы с полиномиальным временем — так называемые *методы внутренней точки*.

Пример линейного программирования также интересен и по другой причине. Самым распространенным алгоритмом для решения этой задачи является *симплексный метод*. Он очень хорошо работает на практике и в реальных задачах может успешно конкурировать с внутренними методами, работающими с полиномиальным временем. Тем не менее известно, что в худшем случае он выполняется за полиномиальное время; просто экспоненциальное поведение очень редко встречается на практике. По этой причине линейное программирование стало очень полезным и важным примером для анализа ограничений полиномиального времени как формального определения эффективности.

Впрочем, для наших целей важно то, что задачи линейного программирования могут решаться за полиномиальное время, и на практике существуют очень эффективные алгоритмы. На курсах линейного программирования вы сможете получить гораздо больше информации по этой теме, а нас сейчас интересует следующий вопрос: как линейное программирование помогает при решении таких комбинаторных задач, как задача о вершинном покрытии?

Задача о вершинном покрытии как целочисленная программа

Вспомните, что вершинным покрытием в графе $G = (V, E)$ называется такое множество $S \subseteq V$, что у каждого ребра в графе хотя бы один конец принадлежит S. Во взвешенной задаче о вершинном покрытии каждая вершина $i \in V$ имеет *вес* $w_i \geq 0$, а вес множества S обозначается $w(S) = \sum_{i \in S} w_i$. Требуется найти вершинное покрытие S, для которого значение $w(S)$ минимально.

Попробуем сформулировать линейную программу, которая находится в близком соответствии с задачей о вершинном покрытии. Следовательно, мы рассматриваем граф $G = (V, E)$ с весом $w_i \geq 0$ для каждого узла i. Линейное программирование основано на использовании векторов переменных. В нашем случае с каждым узлом $i \in V$ связывается переменная x_i, которая моделирует выбор о том, нужно ли включать узел i в вершинное покрытие; $x_i = 0$ означает, что узел i не входит в вершинное покрытие, а с $x_i = 1$ узел включается в него. Мы создадим один n-мерный вектор x, в котором i-я координата соответствует i-й переменной решения x_i.

Требование о том, что выбранные узлы образуют вершинное покрытие, будет закодировано с использованием линейных неравенств, а цель минимизации общего веса кодируется при помощи целевой функции. У каждого ребра $(i, j) \in E$ один

конец должен находиться в вершинном покрытии, и мы запишем этот факт в виде неравенства $x_i + x_j \geq 1$. Наконец, для выражения задачи минимизации множество весов записывается в виде n-мерного вектора w, в котором i-я координата соответствует w_i; мы стремимся минимизировать $w^t x$.

В итоге задача о вершинном покрытии формулируется следующим образом.

$$\text{(VC.IP) Min } \sum_{i \in V} w_i x_i$$

$$\text{s.t. } x_i + x_j \geq 1 \ (i,j) \in E$$

$$x_i \in \{0,1\} \ i \in V.$$

Утверждается, что вершинные покрытия G однозначно соответствуют решениям x этой системы линейных неравенств, в которой все координаты равны 0 или 1.

(11.21) S является вершинным покрытием G в том, и только в том случае, если вектор x, в котором $x_i = 1$ для $i \in S$, и $x_i = 0$ для $i \notin S$, удовлетворяет ограничениям в (VC.IP). Кроме того, $w(S) = w^t x$.

Эта система может быть переведена в матричную форму, используемую для линейного программирования. Мы определяем матрицу A, столбцы которой соответствуют узлам в V, а строки — ребрам в E; элемент $A[e,i] = 1$, если узел i является концом ребра e, и 0 в противном случае. (Каждая строка содержит ровно два ненулевых элемента.) Если использовать 1 для обозначения вектора, все координаты которого равны 1, а 0 — для обозначения вектора, все координаты которого равны 0, то система неравенств может быть записана в следующем виде:

$$Ax \geq \vec{1}$$

$$\vec{1} \geq x \geq \vec{0}.$$

Однако следует помнить, что это не просто экземпляр задачи линейного программирования: мы выдвинули критичное требование о том, что все координаты в решении равны 0 или 1. Итак, из нашей формулировки следует, что нужно решить задачу $\min(w^t x$ subject to $\vec{1} \geq x \geq \vec{0}$, $Ax \geq \vec{1}$, x имеет целые координаты.)

В этом экземпляре задачи о линейном программировании координаты x принимают целые значения; без дополнительного ограничения координаты x могут быть произвольными вещественными числами. Назовем ее *задачей целочисленного программирования*, так как мы ищем целочисленные решения для задачи линейного программирования.

Целочисленное программирование существенно сложнее линейного; однако в нашем обсуждении в действительности происходит сведение задачи о вершинном покрытии к версии задачи целочисленного программирования с принятием решения. иначе говоря, мы доказали, что

(11.22) *Вершинное покрытие* $\leq_P$ *Целочисленное программирование*

Чтобы продемонстрировать *NP*-полноту целочисленного программирования, необходимо еще установить, что версия с принятием решения принадлежит *NP*. И здесь возникают сложности, как и в случае с линейным программированием,

поскольку мы должны доказать, что всегда существует решение x, которое может быть записано с полиномиальным количеством битов. Тем не менее это возможно доказать. В нашем случае задача целочисленного программирования явно ограничивается решениями, в которых каждая координата равна либо 0, либо 1; очевидно, такая задача принадлежит *NP*, а сведение от вершинного покрытия устанавливает, что даже этот особый случай является *NP*-полным.

Использование линейного программирования для задачи вершинного покрытия

Пока еще не совсем понятно, принесет ли реальную пользу наш экскурс в линейное и целочисленное программирование или же мы окажемся в тупике. Очевидно, попытки оптимального решения задачи целочисленного программирования (VC.IP) бесперспективны, так как эта задача является *NP*-сложной.

Чтобы добиться результатов, мы воспользуемся тем фактом, что линейное программирование обладает меньшей сложностью, чем целочисленное программирование. Допустим, мы возьмем задачу (VC.IP) и изменим ее, сняв требование $x_i \in \{0,1\}$, и вернемся к ограничению, в соответствии с которым x_i является произвольным вещественным числом от 0 до 1. При этом будет получен экземпляр задачи линейного программирования (назовем его (VC.LP)), который решается за полиномиальное время: мы можем найти множество значений $\{x^*_i\}$ в диапазоне от 0 до 1, такие что $x^*_i + x^*_j \geq 1$ для каждого ребра (i, j) и значение $\sum_i w_i x^*_i$ минимизировано. Обозначим этот вектор x^*, а значение целевой функции $w_{LP} = w^t x^*$.

Выделим следующий базовый факт:

(11.23) Пусть S^* — вершинное покрытие с минимальным весом. Тогда $w_{LP} \leq w(S^*)$.

Доказательство. Вершинные покрытия G соответствуют целочисленным решениям (VC.IP), так что минимум $\min(w^t x: \vec{1} \geq x \geq 0, Ax \geq 1)$ по всем целочисленным векторам x точно совпадает с вершинным покрытием минимального веса. Чтобы получить минимум для линейной программы (VC.LP), мы разрешим x принимать произвольные вещественные значения, то есть минимизация производится по много большему количеству вариантов x, а следовательно, минимум (VC.LP) не больше минимума (VC.IP). ■

Утверждение (11.23) — один из ключевых ингредиентов, необходимых для алгоритма аппроксимации: хорошая нижняя граница для оптимума в форме эффективно вычисляемой величины w_{LP}.

Однако w_{LP} определенно может быть меньше $w(S^*)$. Например, если граф G представляет собой треугольник, а все веса равны 1, то минимальное вершинное покрытие имеет вес 2. Но в решении задачи линейного программирования мы можем присвоить $x_i = 1/2$ всем трем вершинам и получить решение с весом только 3/2. Если нужен более общий пример, возьмем граф с n узлами, в котором каждая пара узлов соединяется ребром. Все веса снова равны 1. Минимальное вершинное покрытие имеет вес $n - 1$, но мы можем найти решение линейного программирования со значением $n/2$, присвоив $x_i = 1/2$ для всех вершин i.

Возникает вопрос: как решение этой задачи линейного программирования поможет найти вершинное покрытие, близкое к оптимальному? Идея заключается в том, чтобы работать со значениями x^*_i и по ним вычислить вершинное покрытие S. Вполне естественно, что если $x^*_i = 1$ для некоторого узла i, то этот узел включается в вершинное покрытие S, а если $x^*_i = 0$, то узел остается за пределами S. Но что делать с дробными промежуточными значениями? Что делать, если $x^*_i = 0{,}4$ или $x^*_i = 0{,}5$? В таких ситуациях естественно воспользоваться округлением.

Для заданного дробного решения $\{x^*_i\}$ мы определяем $S = I \in V: x^*_i \geq \frac{1}{2}\}$, то есть значения от 1/2 округляются в бóльшую сторону, а значения, меньшие 1/2, округляются в меньшую сторону.

(11.24) Множество S, определяемое таким образом, является вершинным покрытием, и $w(S) \leq w_{LP}$.

Доказательство. Сначала докажем, что S является вершинным покрытием. Рассмотрим ребро $e = (i,j)$. Утверждается, что по крайней мере одна из вершин i и j должна входить в S. Вспомните, что одно из неравенств $x_i + x_j \geq 1$. Следовательно, в любом решении x^*, удовлетворяющем этому неравенству, либо $x^*_i \geq \frac{1}{2}$, либо $x^*_j \geq \frac{1}{2}$. По крайней мере одна из этих величин будет округлена в бóльшую сторону, и i или j включается в S.

Теперь рассмотрим вес $w(S)$ этого вершинного покрытия. Множество S включает вершины только с $x^*_i \geq \frac{1}{2}$; следовательно, экземпляр линейного программирования «заплатил» по крайней мере $\frac{1}{2} w_i$ за узел i, а мы платим w_i; — максимум вдвое больше. В более формальном виде используется следующая цепочка неравенств:

$$w_{LP} w^t x^* = \sum_i w_i x^*_i \geq \sum_{i \in S} w_i x^*_i \geq \frac{1}{2} \sum_{i \in S} w_i = \frac{1}{2} w(S).$$

Таким образом, мы создали вершинное покрытие S с весом не более $2w_{LP}$. Нижняя граница в (11.23) показывает, что оптимальное вершинное покрытие имеет вес не менее w_{LP}, поэтому имеем следующий результат:

(11.25) Алгоритм строит вершинное покрытие S, вес которого не более чем вдвое превышает минимально возможный.

11.7.* Снова о распределении нагрузки: более сложное применение LP

В этом разделе рассматривается более общая задача распределения нагрузки. Мы разработаем аппроксимирующий алгоритм по той же общей схеме, как и в случае

с 2-аппроксимацией, только что разработанной для вершинного покрытия: мы решаем соответствующую задачу линейного программирования, а затем округляем решение. Впрочем, алгоритм и его анализ будут гораздо более сложными, чем требовалось для вершинного покрытия. Как выясняется, экземпляр задачи линейного программирования, который придется решать, по сути, является задачей потока в сети. Используя этот факт, мы сможем гораздо лучше понять, как выглядят нецелые решения задач линейного программирования, и применить это понимание для правильного округления. Для этой задачи единственный аппроксимирующий алгоритм с постоянным множителем основан на округлении решения задачи линейного программирования.

Задача

В этом разделе рассматривается важное естественное обобщение задачи распределения нагрузки, с которой началось наше знакомство с аппроксимирующими алгоритмами. Как тогда, так и сейчас имеется множество J из n заданий, множество M из m машин, и требуется назначить каждое задание некоторой машине, чтобы максимальная нагрузка по любой машине была как можно меньше. В простой задаче распределения нагрузки, рассматривавшейся ранее, каждое задание j могло быть назначено любой машине i. В новом варианте множество машин, рассматриваемых для каждого задания, может быть ограничено; иначе говоря, для каждого задания существует подмножество машин, на которых оно может быть размещено. Это ограничение возникает естественным образом во многих ситуациях: например, при стремлении к распределению нагрузки так, чтобы каждое задание должно назначаться на физически близкую машину или на машину, обладающую необходимыми полномочиями для обработки задания.

В формальном выражении у каждого задания j имеется фиксированный заданный размер $t_j \geq 0$ и множество машин $M_j \subseteq M$, на которые оно может быть назначено. Множества M_j определяются полностью произвольно.

Распределение заданий между машинами называется *действительным*, если каждое задание j назначается на машину $i \in M_j$. Цель, как и прежде, заключается в минимизации максимальной нагрузки на любой машине: используя запись $J_i \subseteq J$ для обозначения заданий, назначенных на машину $i \in M$ в действительном распределении, и запись $L_i = \sum_{j \in J_i} t_j$ для обозначения итоговой нагрузки, мы стремимся минимизировать $\max_i l_i$. Полученная формулировка является определением *обобщенной задачи распределения нагрузки*.

Кроме включения исходной задачи распределения нагрузки как частного случая (в котором $M_j = M$ для всех заданий j), обобщенная задача распределения нагрузки также включает задачу о двудольном идеальном паросочетании как другой частный случай. В самом деле, для двудольного графа с одинаковым количеством узлов на каждой стороне узлы в левой части могут рассматриваться как задания, а узлы в правой части — как машины; мы определяем $t_j = 1$ для всех заданий j и определяем M_j как множество узлов i, для которых существует ребро $(i,j) \in E$. Распределение

с максимальной нагрузкой 1 существует в том, и только в том случае, если в двудольном графе существует идеальное паросочетание. (А следовательно, методы нахождения потока в сети могут использоваться для нахождения оптимальной нагрузки в этом конкретном случае.) Тот факт, что обобщенная задача распределения нагрузки включает обе эти задачи как частные случаи, дает некоторое представление о трудности разработки алгоритма для нее.

Разработка и анализ алгоритма

Разработка аппроксимирующего алгоритма основана на применении линейного программирования для обобщенной задачи распределения нагрузки. Базовый план остается тем же, что и в предыдущем разделе: сначала мы сформулируем задачу как эквивалентную задачу линейного программирования, в которой переменные принимают конкретные дискретные значения; затем мы ослабим ее до задачи линейного программирования, сняв это требование к значениям переменных; и наконец, полученное дробное назначение будет использовано для получения распределения, близкого к оптимальному. При округлении решения для получения распределения нам придется действовать осторожнее, чем в случае с задачей о вершинном покрытии.

Формулировки целочисленного и линейного программирования

Сначала мы сформулируем обобщенную задачу распределения нагрузки как задачу линейного программирования с ограничениями на значения переменных. В формулировке будут использоваться переменные x_{ij}, соответствующие каждой паре (i, j) из машины $i \in M$ и задания $j \in J$. Присваивание $x_{ij} = 0$ означает, что задание j не назначено на машину i, а присваивание x_{ij} сообщает о том, что вся нагрузка t_j задания $j = t_j$ распределена на машину i. При этом x может рассматриваться как один вектор с mn координатами.

Для кодирования требования о том, что каждое задание должно быть назначено на машину, мы воспользуемся линейными неравенствами: для каждого задания j должно выполняться условие $\sum_i x_{ij} = t_j$. Тогда нагрузка на машину i выражается в виде $L_i = \sum_j x_{ij}$. Требуется, чтобы $x_{ij} = 0$ при всех $i \notin M_j$. Мы воспользуемся целевой функцией для кодирования цели — нахождения распределения, минимизирующего максимальную нагрузку. При этом нам понадобится еще одна переменная L, соответствующая нагрузке. Для всех машин i будут использоваться $\sum_j x_{ij} < L$. В итоге мы приходим к следующей формулировке задачи:

$$\text{(GL.IP) min } L$$

$$\sum_i x_{ij} = t_j \quad \text{for all } j \in J$$

$$\sum_j x_{ij} \le L \quad \text{for all } i \in M$$

$$x_{ij} \in \{0, t_j\} \text{ for all } j \in J, i \in M_i$$
$$x_{ij} = 0 \text{ for all } j \in J, i \notin M_j.$$

Сначала мы докажем утверждение о том, что действительные распределения однозначно соответствуют решениям x, удовлетворяющим приведенным выше ограничениям, и в оптимальном решении (GL.IP) L определяет нагрузку соответствующего присваивания.

(11.26) Распределение заданий между машинами имеет нагрузку не более L в том, и только в том случае, если вектор x, элементы которого равны $x_{ij} = t_j$ для задания j, назначенного на машину i, и $x_{ij} = 0$ в противном случае, удовлетворяет ограничениям (GL.IP), где L назначает максимальную нагрузку распределения.

Рассмотрим соответствующую задачу линейного программирования, полученную заменой требования $x_{ij} \in \{0, t_j\}$ более слабым требованием $x_{ij} \geq 0$ для всех $j \in J$ и $i \in M_j$. Обозначим полученную задачу линейного программирования (GL.LP). Также будет естественно добавить $x_{ij} \leq t_j$. Мы не добавляем эти неравенства явно, так как они следуют из неотрицательности и уравнения $\sum_j x_{ij} = t_j$, которое должно выполняться для каждого задания j.

Сразу же видно, что если существует распределение с нагрузкой не более L, то у (GL.IP) должно существовать решение со значением не более L. И наоборот:

(11.27) Если оптимальное значение (GL.LP) равно L, то оптимальная нагрузка не менее $L^* \geq L$.

Мы можем использовать линейное программирование для получения такого решения (x, L) за полиномиальное время. Нашей целью будет использование x для создания распределения. Вспомните, что обобщенная задача распределения нагрузки является *NP*-сложной, а следовательно, мы не можем надеяться найти ее решение в точности за полиномиальное время. Вместо этого мы найдем распределение с нагрузкой, превышающей минимум не более чем вдвое. Для этого нам также понадобится простая нижняя граница (11.2), которая уже использовалась в исходной задаче распределения нагрузки.

(11.28) Оптимальная нагрузка не меньше $L^* \geq \max_j t_j$.

Округление решения при отсутствии циклов

Основная идея заключается в округлении значений x_{ij} до 0 или t_j. Однако мы не можем просто округлять бóльшие значения в сторону увеличения, а меньшие — в сторону уменьшения. Проблема в том, что решение задачи линейного программирования может назначить малые доли задания j на каждую из m машин, а следовательно, для некоторых заданий больших значений x_{ij} может не быть.

Наш алгоритм будет производить «слабое» округление x: каждое задание j будет назначаться на машину i с $x_{ij} > 0$, но возможно, некоторые малые значения придется округлять в бóльшую сторону. Слабое округление уже гарантирует, что распределе-

ние является действительным в том смысле, что никакое задание j не назначается на машину i, не входящую в M_j (потому что если $i \notin M_j$, то $x_{ij} = 0$).

Важно понять структуру нецелочисленного решения и показать, что хотя некоторые задания могут быть распределены по нескольким машинам, таких заданий не может быть слишком много. Для этого мы рассмотрим следующий двудольный граф $G(x) = (V(x), E(x))$: множество узлов $V(x) = M \cup J$, множество заданий и множество машин и ребро $(i, j) \in E(x)$ существуют в том, и только в том случае, если $x_{ij} > 0$.

Мы покажем, что при наличии решения для (GL.LP) можно получить новое решение x с такой же нагрузкой L, в котором $G(x)$ не содержит циклов. Этот шаг очень важен, так как вы увидите, что решение x без циклов может использоваться для получения распределения с нагрузкой не более $L + L^*$.

(11.29) Для решения (x,L) задачи (GL.LP), в котором граф $G(x)$ не имеет циклов, решение x может использоваться для получения действительного распределения заданий между машинами с нагрузкой не более $L + L^*$ за время $O(mn)$.

Доказательство. Так как граф $G(x)$ не содержит циклов, каждая из его компонент связности является деревом. Распределение можно получить, рассматривая каждую компоненту по отдельности. Возьмем одну из компонент, которая представляет собой дерево; ее узлы соответствуют заданиям и машинам, как показано на рис. 11.11.

Разместим корень дерева в произвольном узле и рассмотрим задание j. Если узел, соответствующий заданию j, является узлом дерева, пусть узел машины i является его родителем. Так как степень j в дереве $G(x)$ равна 1, машина i — единственная машина, которой была назначена часть задания j, а следовательно, должно выполняться $x_{ij} = t_j$. В нашем распределении такое задание j будет назначено только на его соседнюю машину i. Задание j, узел которого не является листовым в $G(x)$, назначается произвольному дочернему узлу соответствующего дочернего узла в корневом дереве.

Очевидно, что этот метод может быть реализован за время $O(mn)$ (включая время создания графа $G(x)$). Он определяет действительное распределение, так как задача линейного программирования (GL.LP) требует $x_{ij} = 0$ для $i \notin M_j$. Чтобы завершить доказательство, необходимо показать, что нагрузка не превышает $L + L^*$. Пусть i —любая машина, а J_i — множество заданий, назначенных на машину i. Задания, назначенные на машину i, образуют подмножество соседей i в $G(x)$: множество J_i содержит дочерние узлы узла i, которые являются листьями, а также, возможно, родителя $p(i)$ узла i. Чтобы ограничить нагрузку, мы рассмотрим родителя $p(i)$ отдельно. Для всех остальных заданий $j = p(i)$, назначенных i, $x_{ij} = t_j$, а следовательно для ограничения нагрузки можно воспользоваться решением x:

$$\sum_{j \in J_i,\, j \neq p(i)} t_j \leq \sum_{j \in J} x_{ij} \leq L$$

с использованием неравенства, ограничивающего нагрузку в (GL.LP). Для родителя $j = p(i)$ узла i используется $t_j \leq L^*$ согласно (11.28). Суммируя два неравенства, приходим к $\sum_{j \in J_i} p_{ij} \leq L + L^*$, как и утверждалось. ■

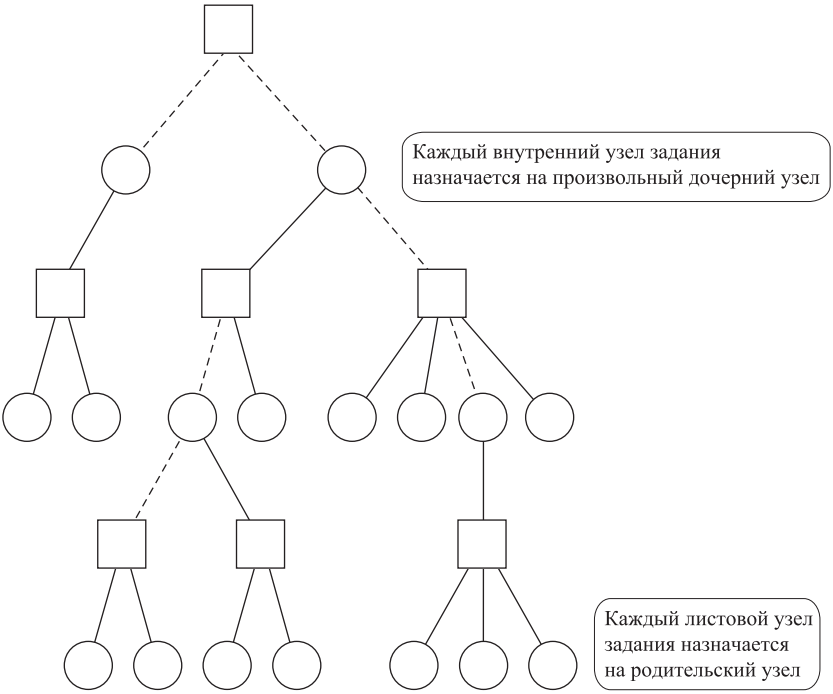


Рис. 11.11. Пример графа $G(x)$, не содержащего циклов; квадраты изображают машины, а круги — задания. Сплошными линиями обозначено итоговое распределение заданий между машинами

Теперь из (11.27) известно, что $L \leq L^*$, так что решение, нагрузка которого ограничивается $L + L^*$, также ограничивается $2L^*$ — то есть удвоенный оптимум. Итак, приходим к следующему следствию из (11.29).

(11.30) Для решения (x, L) задачи (GL.LP), в котором граф $G(x)$ не имеет циклов, решение x может использоваться для получения действительного распределения заданий между машинами с нагрузкой, не превышающей удвоенного оптимума, за время $O(mn)$.

Исключение циклов из решения задачи линейного программирования

Чтобы завершить аппроксимирующий алгоритм, остается лишь показать, как преобразовать произвольное решение (GL.LP) в решение x, не содержащее циклов в $G(x)$. Попутно мы также покажем, как получить решение задачи линейного программирования (GL.LP) с использованием потоковых вычислений. А точнее, для заданного значения нагрузки L мы покажем, как при помощи потоковых вычислений решить, имеет ли (GL.LP) решение со значением не более L. Для этого рассмотрим направленный граф $G = (V, E)$, изображенный на рис. 11.12. Множество вершин потокового графа G $V = M \cup J \cup \{v\}$, где v — новый узел. Узлы $j \in J$ будут

источниками с поставкой t_j, а единственным потребляющим узлом будет новый сток v с уровнем потребления $\sum_j t_j$. Поток в этой сети будет рассматриваться как «перетекание» нагрузки из заданий в сток v через машины. Ребро (j,i) с бесконечной пропускной способностью от задания j к машине i добавляется в том, и только в том случае, если $i \in M_j$. Наконец, мы добавляем ребро (i,v) для каждого узла машины i с пропускной способностью L.

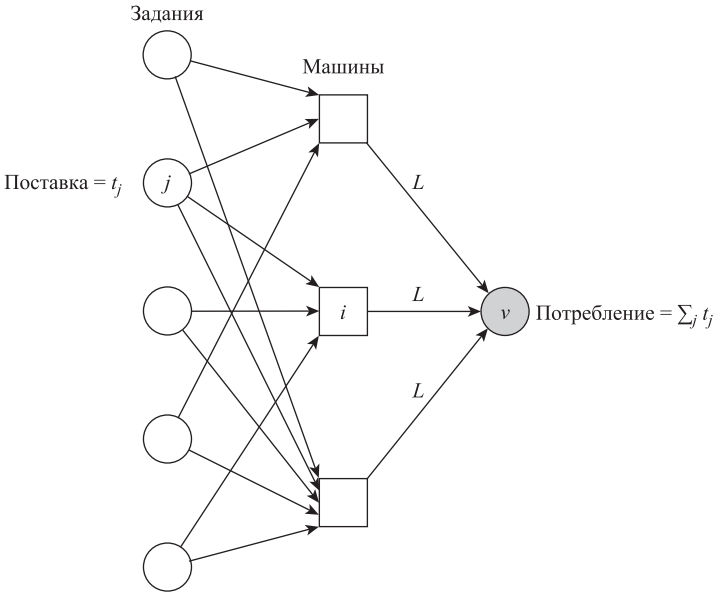


Рис. 11.12. Схема вычисления потока в сети, используемая для поиска решения (GL.LP). Ребра между заданиями и машинами обладают бесконечной пропускной способностью

(11.31) Решения задачи нахождения потока с пропускной способностью L однозначно соответствуют решениям (GL.LP) со значением L, где x_{ij} — величина потока по ребру (i, j), а величина потока по ребру (i, t) равна нагрузке $\sum_j t_j$ на машине i.

Это утверждение позволяет решить (GL.LP) с использованием потоковых вычислений и бинарного поиска для оптимального значения L: мы последовательно опробуем значения L, пока не найдем минимальное, для которого существует действительный поток.

Воспользуемся информацией, полученной о (GL.LP) из эквивалентной формулировки задачи нахождения потока, и изменим решение x для устранения из $G(x)$ всех циклов. В контексте только что определенного потока $G(x)$ представляет собой ненаправленный граф, для получения которого в G игнорируются направления ребер, удаляется сток v и все прилегающие ребра, а также удаляются все ребра из J в M, по которым не передается поток. Удаление всех циклов из $G(x)$ будет выполнено не более чем за mn шагов, а целью каждого шага является исключение из $G(x)$ по крайней мере одного ребра без увеличения нагрузки L или добавления новых ребер.

(11.32) Пусть (x, L) — произвольное решение (GL.LP), а C — цикл в $G(x)$. За время, линейное по отношению к длине цикла, можно изменить решение x для исключения из $G(x)$ по крайней мере одного ребра без увеличения нагрузки или добавления новых ребер.

Доказательство. Рассмотрим цикл C в $G(x)$. Вспомните, что $G(x)$ соответствует множеству ребер, передающих поток в решении x. Мы изменим решение, увеличивая поток по циклу C; для этого будет использоваться процедура `augment` из раздела 7.1. Увеличение потока в цикле не изменит баланс между входным и выходным потоком в любом узле; вместо этого оно приведет к исключению одного обратного ребра из остаточного графа, а следовательно, ребра из $G(x)$. Предположим, узлы в цикле обозначаются $i_1, j_1, i_2, j_2, ..., i_k, j_k$, где i_ℓ — узел машины, а j_ℓ — узел задания. Мы изменим решение, сокращая поток по всем ребрам $(i_\ell j_\ell)$ и увеличивая поток по ребрам $(j_\ell, i_{\ell+1})$ для всех $\ell = 1, ..., k$ (где $k + 1$ обозначает 1) на одинаковую величину δ. Изменение не повлияет на ограничения сохранения потока. Задавая $\delta = \min_{\ell=1}^{k} x_{i_\ell j_\ell}$, мы гарантируем, что поток остается действительным, а ребро, в котором достигается минимум, удаляется из $G(x)$. ■

Алгоритм, содержащийся в доказательстве (11.32), можно применить многократно для удаления всех циклов из $G(x)$. В исходном состоянии $G(x)$ может содержать mn ребер, так что после максимум $O(mn)$ итераций полученное решение (x, L) не содержит циклов в $G(x)$. На этой стадии можно применить (11.30) для получения действительного распределения с нагрузкой, превышающей оптимум не более чем вдвое.

(11.33) Для заданного экземпляра обобщенной задачи распределения нагрузки может быть найдено за полиномиальное время действительное распределение с нагрузкой, превышающей возможный минимум не более чем вдвое.

11.8. Аппроксимации с произвольной точностью: задача о рюкзаке

Когда к вам обращаются за помощью люди, столкнувшиеся с *NP*-сложной задачей оптимизации, они часто надеются получить алгоритм, способный выдать решение в пределах, скажем, 1 % от оптимума — или по крайней мере в относительно небольшом диапазоне от оптимума. С этой точки зрения аппроксимирующие алгоритмы, рассматривавшиеся ранее, были довольно слабыми: решения с множителем 2 для минимума задач выбора центров и вершинного покрытия (то есть превышение оптимума может достигать 100 %). С алгоритмом покрытия множества из раздела 10.3 дело обстоит еще хуже: его стоимость даже не лежит в пределах фиксированного постоянного множителя от возможного минимума!

В основе этого состояния дел лежит важный факт: все *NP*-полные задачи, как вам известно, эквивалентны в отношении разрешимости с полиномиальным временем; но в предположении $P \neq NP$ они существенно различаются по возможности эффективной аппроксимации их решений. В некоторых случаях ограничения

аппроксимируемости могут быть доказаны. Например, если $P \neq NP$, то гарантия, предоставляемая алгоритмом выбора центров, является лучшей из возможных среди всех алгоритмов с полиномиальным временем. Аналогичным образом гарантия, предоставляемая алгоритмом покрытия множества, как бы плохо она ни выглядела, очень близка к лучшей из возможных (если только не окажется, что $P = NP$). Для других задач — например, задачи о вершинном покрытии — приведенный нами аппроксимирующий алгоритм является лучшим из известных, но вопрос о том, не существуют ли алгоритмы с полиномиальным временем, обладающий лучшими гарантиями, остается открытым. В этой книге тема нижних границ аппроксимируемости не рассматривается; хотя некоторые нижние границы такого типа доказываются не так уж сложно (например, как в задаче о выборе центров), доказательства во многих случаях перегружаются огромным количеством технических подробностей.

Задача

В этом разделе рассматривается *NP*-полная задача, для которой возможно построение алгоритма с полиномиальным временем, обеспечивающим очень сильную аппроксимацию. Мы рассмотрим слегка обобщенную версию задачи о рюкзаке (или задачи о сумме подмножеств). Допустим, имеется n предметов, которые нужно уложить в рюкзак. Каждый предмет $i = 1, ..., n$ обладает двумя целочисленными параметрами: весом w_i и значением v_i. Для заданной емкости рюкзака W требуется найти подмножество S предметов, обладающее максимальным значением, при условии, что общий вес подмножества не превышает W. Другими словами, требуется максимизировать $\sum_{i \in S} v_i$ при условии $\sum_{i \in S} w_i \leq W$.

На какую точность аппроксимации можно рассчитывать? Наш алгоритм получает на входе веса и значения, определяющие задачу, а также дополнительный параметр ϵ (требуемая точность). Он находит подмножество S, суммарный вес которого не превышает W, с суммарным значением $\sum_{i \in S} v_i$, уступающим максимально возможному не более чем на $(1 + \epsilon)$. Алгоритм выполняется за полиномиальное время для любого *фиксированного* выбора $\epsilon > 0$; при этом зависимость от ϵ не является полиномиальной. Назовем этот алгоритм *схемой аппроксимации с полиномиальным временем*.

Возникает вопрос: возможно ли, чтобы такой сильный аппроксимирующий алгоритм выполнялся за полиномиальное время, когда задача о рюкзаке является *NP*-сложной? Ведь с целочисленными значениями при достаточном приближении к оптимальному значению мы должны достичь самого оптимума! Загвоздка кроется в неполиномиальной зависимости от желаемой точности: для любого фиксированного выбора ϵ — например, $\epsilon = 0{,}5$, $\epsilon = 0{,}2$ или даже $\epsilon = 0{,}01$ — алгоритм выполняется за полиномиальное время, но при замене ϵ все меньшими и меньшими значениями время выполнения возрастает. К тому времени, когда ϵ станет достаточно малым для гарантированного достижения оптимума, алгоритм уже не будет алгоритмом с полиномиальным временем.

Разработка алгоритма

В разделе 6.4 рассматривались алгоритмы для задачи о сумме подмножеств — частного случая задачи о рюкзаке, в котором $v_i = w_i$ для всех элементов i. Для этого частного случая был предложен алгоритм динамического программирования, выполнявшийся за время $O(nW)$ при условии, что веса являются целыми числами. Алгоритм естественным образом расширяется на более общую задачу о рюкзаке (см. конец раздела 6.4). Алгоритм, приведенный в разделе 6.4, хорошо работает для малых весов (даже если значения велики). Также возможно определить алгоритм динамического программирования для случая с малыми значениями (даже при больших весах). В конце раздела приводится алгоритм динамического программирования для этого случая, выполняемый за время $O(n^2v^*)$, где $v^* = \max_i v_i$. Учтите, что этот алгоритм не выполняется за полиномиальное время: он только псевдополиномиален из-за зависимости от размера значений v_i. Так как *NP*-полнота этой задачи была доказана в главе 8, не стоит рассчитывать, что нам удастся найти алгоритм с полиномиальным временем.

Алгоритмы с псевдополиномиальной зависимостью от значений часто могут использоваться для разработки схем аппроксимации с полиномиальным временем, и разработанный нами алгоритм очень наглядно демонстрирует эту базовую стратегию. В частности, мы используем алгоритм динамического программирования с временем выполнения $O(n^2v^*)$ для разработки схемы аппроксимации с полиномиальным временем: если значения являются малыми целыми числами, то значение v^* мало, а задача уже может быть решена за полиномиальное время. С другой стороны, если значения велики, то нам не обязательно работать с ними в точном виде, так как нас интересует только аппроксимированно-оптимальное решение. Мы используем параметр округления b (значение которого будет задано позднее) и будем рассматривать значения, округленные до целого, кратного b. Алгоритм динамического программирования будет использоваться для решения задачи с округленными значениями. А точнее, для каждого элемента i за его округленное значение принимается $\tilde{v}_i = \lceil v_i / b \rceil b$. Обратите внимание: округленное и исходное значения достаточно близки друг к другу.

(11.34) Для каждого элемента i выполняется $v_i \le \tilde{v}_i \le v_i + b$.

Что нам дает округление? Если значения были изначально большими, то меньше они не станут. Однако все округленные значения представляют собой целые числа, кратные общему параметру b, поэтому вместо решения задачи с округленными значениями $\tilde{v}_i$ можно сменить «единицы измерения»; мы разделим все значения на b и получим эквивалентную задачу. Пусть $\hat{v}_i = \tilde{v}_i / b = \lceil v_i / b \rceil$ для $i = 1, ..., n$.

(11.35) Задача о рюкзаке со значениями $\tilde{v}_i$ и масштабированная версия задачи со значениями $\hat{v}_i$ имеют одинаковое множество оптимальных решений, оптимальные решения отличаются точно множителем b, а масштабированные значения являются целочисленными.

Теперь все готово к представлению аппроксимирующего алгоритма. Будем считать, что все элементы имеют вес, не превышающий W (так как элементы с весом $w_i > W$ заведомо не входят в решение и поэтому могут быть удалены). Также для простоты будем считать, что ϵ^{-1} является целым числом.

```
Knapsack-Approx(ε):
Пусть b = (ε/(2n)) max_i v_i
Решить задачу о рюкзаке со значениями v̂_i (эквивалентно ṽ_i).
Вернуть множество S найденных элементов
```

Анализ алгоритма

Для начала отметим, что найденное решение по меньшей мере является действительным; то есть $\sum_{i \in S} w_i \leq W$. Это следует из того, что мы округляли только значения, но не веса. Именно по этой причине нам понадобится новый алгоритм динамического программирования, описанный в конце этого раздела.

(11.36) Общий вес множества элементов S, возвращаемого алгоритмом, не превышает W, то есть $\sum_{i \in S} w_i \leq W$.

Далее мы докажем, что этот алгоритм выполняется за полиномиальное время.

(11.37) Алгоритм `Knapsack-Approx` выполняется за полиномиальное время для любого фиксированного $\epsilon > 0$.

Доказательство. Присваивание b и округление очевидно могут быть выполнены за полиномиальное время. Основные затраты времени в этом алгоритме связаны с решением округленной задачи средствами динамического программирования. Вспомните, что для задач с целыми значениями используемый нами алгоритм динамического программирования выполняется за время $O(n^2 v^*)$, где $v^* = \max_i v_i$.

Теперь мы применим этот алгоритм к экземпляру, в котором каждый элемент i имеет вес w_i и значение $\hat{v}_i$. Чтобы определить время выполнения, необходимо определить $\max_i \hat{v}_i$. Элемент j с максимальным значением $v_j = \max_i v_i$ также имеет максимальное значение в округленной задаче, так что $\max_i \hat{v}_i = \hat{v}_j = \lceil v_i / b \rceil = n\epsilon^{-1}$. Следовательно, общее время выполнения алгоритма составляет $O(n^3\epsilon^{-1})$. Обратите внимание: для любого фиксированного $\epsilon > 0$ это полиномиальное время, как и было заявлено; но зависимость от требуемой точности ϵ не полиномиальна, так как время выполнения включает ϵ^{-1} вместо $\log \epsilon^{-1}$. ■

Остается решить ключевой вопрос: насколько хорошее решение дает этот алгоритм? Утверждение (11.34) показывает, что используемые значения $\tilde{v}_i$ близки к реальным значениям v_i, а это подсказывает, что полученное решение не может быть далеко от оптимального.

(11.38) Если S — решение, найденное алгоритмом `Knapsack-Approx`, и S^* — любое другое решение, удовлетворяющее $\sum_{i \in S^*} w_i \leq W$, то $(1 + \epsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$.

Доказательство. Пусть S^* — любое множество, удовлетворяющее $\sum_{i \in S^*} w_i \leq W$. Наш алгоритм находит оптимальное решение со значениями $\tilde{v}_i$, так что мы знаем, что

$$\sum_{i \in S} \tilde{v}_i \geq \sum_{i \in S^*} \tilde{v}_i.$$

Округленные значения $\tilde{v}_i$ и реальные значения v_i достаточно близки согласно (11.34), поэтому мы получаем цепочку неравенств

$$\sum_{i\in S^*} v_i \le \sum_{i\in S^*} \tilde{v}_i \le \sum_{i\in S} \tilde{v}_i \le \sum_{i\in S} (v_i + b) \le nb + \sum_{i\in S} v_i.$$

Она показывает, что значение $\sum_{i\in S} v_i$ решения, полученного нами, максимум на nb меньше максимально возможного значения. Мы хотели получить относительную погрешность, показывающую, что полученное значение $\sum_{i\in S} v_i$ максимум в $(1 + \epsilon)$ раз меньше возможного максимума, поэтому сейчас нужно сравнить nb со значением $\sum_{i\in S} v_i$.

Пусть j — элемент с наибольшим значением; в соответствии с нашим выбором b имеем $v_j = 2\epsilon^{-1}nb$ и $v_j = \tilde{v}_j$. Из нашего предположения о том, что каждый элемент сам по себе помещается в рюкзак ($w_i \le W$ для всех i), следует $\sum_{i\in S} \tilde{v}_i \ge \tilde{v}_j = 2\epsilon^{-1}nb$. Наконец, приведенная выше цепочка неравенств означает, что $\sum_{i\in S} v_i \ge \sum_{i\in S} \tilde{v}_i - nb$, а следовательно, $\sum_{i\in S} v_i \ge (2\epsilon^{-1} - 1)nb$. Отсюда следует $nb \le \epsilon \sum_{i\in S} v_i$ для $\epsilon \le 1$, и в конечном итоге

$$\sum_{i\in S} v_i \le \sum_{i\in S} v_i + nb \le (1+\epsilon)\sum_{i\in S} v_i.$$

Новый алгоритм динамического программирования для задачи о рюкзаке

Чтобы решить задачу методом динамического программирования, необходимо определить полиномиальное множество задач. Алгоритм динамического программирования, определенный ранее в ходе анализа задачи о рюкзаке, использует подзадачи в форме `OPT`(i,w): подзадача нахождения максимального значения любого решения с использованием подмножества элементов 1, ..., i и рюкзака с весом w. При больших весах это множество задач будет большим. Нам нужно множество подзадач, которые бы хорошо работали при достаточно малых значениях; это наводит на мысль, что следует использовать подзадачи, связанные со значениями, а не с весами. Эти подзадачи будут определяться следующим образом: подзадача определяется i и целевым значением V, а $\overline{OPT}$ (i, V) — наименьший вес W, при котором можно получить решение с использованием подмножества элементов $\{1, ..., i\}$ со значением не менее V. Подзадачи будут создаваться для всех $i = 0, ..., n$ и значений $V = 0, ..., \sum_{j=1}^{i} v_j$. Если v^* обозначает $\max_i v_i$, то мы видим, что наибольшее V, которое может быть получено в подзадаче, равно $\sum_{j=1}^{n} v_j \le nv^*$. Следовательно, в предположении о том, что все значения целочисленны, всего будет не более $O(n^2v^*)$ подзадач. Ни одна из этих подзадач не совпадает в точности с исходным

экземпляром задачи о рюкзаке, но если мы располагаем значениями всех подзадач $\overline{OPT}(n,V)$ для $V = 0, ..., \sum_i v_i$, то значение исходной задачи можно легко получить: это наибольшее значение V, для которого $\overline{OPT}(n,V) \le W$.

Получить рекуррентное отношение для решения этих подзадач нетрудно. По аналогии с алгоритмом динамического программирования для задачи о суммировании подмножеств мы рассматриваем случаи в зависимости от того, включается последний элемент n в оптимальное решение O или нет:

- Если $n \notin O$, то $\overline{OPT}(n,V) = \overline{OPT}\ (n-1,V)$.
- Если $n \in O$ — единственный элемент в O, то $\overline{OPT}(n,V) = w_n$.
- Если $n \in O$ — не единственный элемент в O, то $\overline{OPT}(n,V) = w_n + \overline{OPT}\ (n-1, V-v_n)$.

Два последних варианта можно более компактно записать в следующем виде:

- Если $n \in O$, то $\overline{OPT}(n,V) = w_n + \overline{OPT}\ (n-1,\max(0,V-v_n))$.

Из этого следует аналогия с рекуррентным отношением (6.8) из главы 6:

(11.39) Если $V > \sum_{i=1}^{n-1} v_i$, то $\overline{OPT}(n,V) = w_n + \overline{OPT}\ (n-1,V-v_n)$. В противном случае $\overline{OPT}(n,V) = \min(\overline{OPT}\ (n-1,V), w_n + \overline{OPT}\ (n-1,\max(0,V-v_n)))$.

Тогда по аналогии можно записать следующий алгоритм динамического программирования.

```
Knapsack(n):
  Массив M[0...n, 0...V]
  For i = 0, ..., n
     M[i,0] = 0
  End For
  For i = 1, 2, ..., n
     For v = 0, ..., ∑_{j=1}^{i} v_j
        Если v > ∑_{j=1}^{i-1} v_j
           M[i,V] = w_i + M[i − 1,V]
        Иначе
           M[i,V] = min(M[i − 1,V], w_i + M[i − 1, max(0,V − v_i)])
        Конец Если
     Конец For
  Конец For
  Вернуть максимальное значение V, для которого M[n,V] ≤ W
```

(11.40) Алгоритм *Knapsack*(n) выполняется за время $O(n^2v^*)$ и правильно вычисляет оптимальные значения подзадач.

Как и прежде, оптимальное решение находится обратным отслеживанием по таблице M, содержащей оптимальные значения подзадач.

Упражнения с решениями

Упражнение с решением 1

Вспомните жадный алгоритм для задачи интервального планирования: для заданного множества интервалов многократно выбирается кратчайший интервал I, удаляются все остальные интервалы I', пересекающиеся с I, после чего все повторяется.

В главе 4 было показано, что алгоритм не всегда строит множество максимального размера с неперекрывающимися интервалами. Однако, как выясняется, он предоставляет интересную гарантию аппроксимации. Если s^* — максимальный размер множества неперекрывающихся интервалов, а s — размер множества, произведенного этим алгоритмом, то $s \geq \frac{1}{2}s^*$ (то есть алгоритм выбора кратчайшего интервала является 2-аппроксимацией).

Докажите этот факт.

Решение

Вспомним пример на рис. 4.1 из главы 4: он показывал, что алгоритм выбора кратчайшего интервала не обязательно находит оптимальное множество интервалов. Проблема очевидна: выбор короткого интервала j может привести к исключению двух более длинных прилегающих интервалов i и i', так что алгоритм наполовину уступает оптимуму.

Вопрос в том, как показать, что алгоритм выбора кратчайшего интервала никогда не работает хуже этого. Возникающие проблемы напоминают те, которые встречались в ходе анализа жадного алгоритма для задачи максимальных непересекающихся путей из раздела 11.5: каждый выбираемый интервал может «блокировать» некоторые интервалы в оптимальном решении, и мы хотим обосновать, что при постоянном выборе кратчайшего возможного интервала последствия от блокировки будут умеренными. В случае непересекающихся путей перекрытия между путями приходилось анализировать ребро за ребром, так как используемый граф мог иметь произвольную структуру. На этот раз можно воспользоваться сильно ограниченной структурой интервалов на линии для получения усиленной границы.

Чтобы алгоритм выбора кратчайшего интервала работал хуже чем наполовину от оптимума, должно существовать большое оптимальное решение, которое перекрывается с много меньшим решением, выбираемым алгоритмом. Интуитивно кажется, что единственная ситуация, в которой это может произойти, — если один из интервалов i в оптимальном решении полностью укладывается в один из интервалов j, выбираемых алгоритмом выбора кратчайшего пути. Однако это противоречит поведению алгоритма: почему он не выбрал более короткий интервал i, вложенный в j?

Посмотрим, удастся ли точно сформулировать этот аргумент. Пусть A — множество интервалов, выбираемых алгоритмом, а O — оптимальное множество интервалов. Для каждого интервала $j \in A$ рассмотрим множество интервалов в O, с которыми он конфликтует.

(11.41) Каждый интервал $j \in A$ конфликтует не более чем с двумя интервалами в O.

Доказательство. Действуя от обратного, предположим, что в $j \in A$ существует интервал, конфликтующий не менее чем с тремя интервалами из $i_1, i_2, i_3 \in O$. Эти три интервала не конфликтуют друг с другом, так как они являются частью одного решения O, поэтому они располагаются последовательно во времени. Предположим, сначала идет i_1, затем i_2, а потом i_3. Так как интервал j конфликтует и с i_1 и с i_3, расположенный между ними интервал i_2 должен быть короче j и полностью помещаться в нем. Кроме того, так как интервал i_2 не был выбран алгоритмом, он должен был быть доступен как вариант на тот момент, когда алгоритм выбрал интервал j. Возникает противоречие, так как i_2 короче j. ■

Алгоритм выбора кратчайшего пути завершается только тогда, когда каждый невыбранный интервал конфликтует с одним из выбранных интервалов. Следовательно, каждый интервал в O либо включается в A, либо конфликтует с интервалом в A.

Теперь мы воспользуемся следующей схемой для ограничения количества интервалов в O. Для каждого $i \in O$ некоторый интервал $j \in A$ «платит» за i следующим образом: если i также принадлежит A, то i платит за себя. В противном случае мы произвольно выбираем интервал $j \in A$, конфликтующий с i. Как было показано выше, каждый интервал в O конфликтует с некоторым интервалом в A, так что все интервалы в O будут оплачиваться по этой схеме. Однако согласно (11.41), каждый интервал $j \in A$ конфликтует не более чем с двумя интервалами в O, поэтому он заплатит не более чем за два интервала. Следовательно, все интервалы в O оплачены интервалами в A, и в этом процессе каждый интервал в A платит максимум два раза. Из этого следует, что множество A должно содержать не менее половины от количества интервалов O.

Упражнения

1. Предположим, вы консультируете управление порта в маленькой стране на побережье Тихого океана. Ежегодный объем сделок исчисляется миллиардами, и доход ограничивается исключительно скоростью разгрузки кораблей, прибывающих в порт.

   Типичная задача, возникающая в работе порта, выглядит так: прибывает корабль с n контейнерами веса $w_1, w_2, ..., w_n$. В доке стоят грузовики, каждый из которых способен перевозить до K единиц веса. (Считайте, что K и все w_i являются целыми числами.) В один грузовик можно погрузить несколько контейнеров, на которые распространяется ограничение по весу K; целью является минимизация количества грузовиков, необходимых для перевозки всех контейнеров. Задача является *NP*-полной (это доказывать не надо).

   Жадный алгоритм для решения этой задачи мог бы выглядеть так: начать с пустого грузовика и загружать в него контейнеры 1, 2, 3 ... до тех пор, пока вы не доберетесь до контейнера, который приведет к нарушению ограничения.

Этот грузовик объявляется «загруженным» и отправляется, после чего процесс повторяется со следующим грузовиком. Алгоритм, рассматривающий грузовики по отдельности, может не обеспечить самый эффективный способ упаковки всего множества контейнеров в доступное множество грузовиков.

(a) Приведите пример множества весов и значения K, при которых алгоритм не использует минимально возможное количество грузовиков.

(b) Покажите, что количество грузовиков, используемых этим алгоритмом, не более чем в 2 раза превышает минимально возможное для любого множества весов и любого значения K.

2. На лекции по вычислительной биологии, которую один из авторов посетил несколько лет назад, известный химик рассказывал об идее построения «представительного множества» для обширного множества белковых молекул, свойства которых нам непонятны. Идея заключается в том, чтобы интенсивно изучать белки из представительного множества и тем самым получить информацию (косвенно) обо всех белках полного множества.

   Чтобы представительное множество приносило пользу, оно должно обладать двумя свойствами:

   - оно должно быть относительно небольшим, чтобы его изучение не обходилось слишком дорого;
   - каждый белок в полном множестве должен быть «похож» на некоторый белок из представительного множества. (То есть множество действительно предоставляет некоторую информацию обо всех белках.)

   В более конкретной формулировке существует большое множество белков P. Сходство между белками определяется функцией расстояния d: для двух белков p и q эта функция возвращает число $d(p, q) \geq 0$. Функция $d(\cdot, \cdot)$ обычно используется в качестве метрики выравнивания последовательностей (эта задача рассматривалась при изучении динамического программирования в главе 6); будем считать, что она используется и в этом случае. Существует заранее определенный *порог расстояния* Δ, который задается как часть входных данных задачи; два белка p и q считаются «похожими» в том, и только в том случае, если $d(p, q) \leq \Delta$.

   Подмножество P называется *представительным множеством*, если для каждого белка p существует похожий на него белок q, принадлежащий этому подмножеству, то есть белок, для которого $d(p,q) \leq \Delta$. Требуется найти представительное множество минимального размера.

   (a) Предложите алгоритм с полиномиальным временем, аппроксимирующий минимальное представительное множество до множителя $O(\log n)$. В более конкретной формулировке алгоритм должен обладать следующим свойством: если минимальный возможный размер представительного множества равен s^*, то алгоритм должен возвращать представительное множество с размером не более $O(s^* \log n)$.

   (b) Обратите внимание на сходство этой задачи с задачей о выборе центров; аппроксимирующие алгоритмы для последней рассматривались в разделе 11.2. Почему описанный там алгоритм не решает текущую задачу?

3. Допустим, имеется множество положительных целых чисел $A = \{a_1, a_2, ..., a_n\}$ и положительное целое число B. Подмножество $S \subseteq A$ будет называться *действительным*, если сумма чисел в S не превышает B:

$$\sum_{a_i \in S} a_i \leq B.$$

Сумма чисел в S называется *полной суммой* S.

Требуется выбрать действительное подмножество S множества A с максимально возможной полной суммой.

Пример. Если $A = \{8,2,4\}$ и $B = 11$, то оптимальным решением будет подмножество $S = \{8,2\}$.

(a) Рассмотрим следующий алгоритм для решения этой задачи.

```
Инициализировать S = φ
Определить T = 0
For i = 1,2, ..., n
   Если T + a_i ≤ B
      S ← S ∪ {a_i}
      T ← T + a_i
   Конец Если
Конец For
```

Приведите пример, в котором полная сумма множества S, возвращаемого этим алгоритмом, меньше половины полной суммы некоторого действительного подмножества A.

(b) Предложите аппроксимирующий алгоритм с полиномиальным временем, предоставляющий следующую гарантию: полная сумма возвращаемого действительного множества $S \subseteq A$ не меньше половины максимальной полной суммы любого действительного множества $S' \subseteq A$. Время выполнения алгоритма не должно превышать $O(n \log n)$.

4. Рассмотрим оптимизированную версию задачи о множестве представителей, которая определяется следующим образом. Имеется множество $A = \{a_1, ..., a_n\}$ и набор $B_1, B_2, ..., B_m$ подмножеств A. Кроме того, каждый элемент $a_i \in A$ имеет вес $w_i \geq 0$. Требуется найти множество представителей $H \subseteq A$ с минимально возможным общим весом элементов H — то есть $\sum_{a_i \in H} w_i$. (Как и в упражнении 5 главы 8, множеством представителей называется такое множество H, что $H \cap B_i$ не пусто для всех i.) Обозначим $b = \max_i |B_i|$ максимальный размер множеств $B_1, B_2, ..., B_m$. Предложите аппроксимирующий алгоритм с полиномиальным временем для нахождения множества представителей, общий вес которого не более чем в b раз превышает минимально возможный.

5. Вы работаете на фирму, клиенты которой ежедневно предоставляют задания для обработки. Каждое задание характеризуется временем обработки t_i, известным при поступлении задания. Компания располагает парком из 10 машин, и каждое задание может быть выполнено на любой машине.

В настоящее время фирма использует простой жадный алгоритм распределения нагрузки, рассмотренный в разделе 11.1. Директор фирмы знает, что этот аппроксимирующий алгоритм может быть не лучшим, и интересуется, нужно ли опасаться неэффективности. Впрочем, изменять механизм он не хочет, потому что ему нравится простота текущего алгоритма: задания можно назначать машинам сразу же после поступления, не откладывая принятие решения до прихода новых заданий.

В частности, он слышал, что этот алгоритм может строить решения с периодом обработки вдвое больше минимума. Однако в целом опыт использования алгоритма был вполне неплох: он ежедневно выполнялся в течение последнего месяца, и период обработки ни разу не превысил среднюю нагрузку $\frac{1}{10}\sum_i t_i$ более чем на 20 %.

Чтобы понять, почему «удвоенное» поведение не проявляется в работе, вы собираете информацию о типичных заданиях и нагрузках. Как выясняется, размеры заданий лежат в диапазоне от 1 до 50, то есть $1 \le t_i \le 50$ для всех заданий i; а ежедневная общая нагрузка $\sum_i t_i$ достаточно высока: она не бывает ниже 3000.

Докажите, что для таких входных данных жадный алгоритм распределения нагрузки всегда находит решение, продолжительность обработки которого превышает среднюю не более чем на 20 %.

6. Вспомните, что в базовой задаче распределения нагрузки из раздела 11.1 нас интересовало распределение заданий между машинами, минимизирующее *период обработки* — максимальную нагрузку на одной машине. Достаточно часто приходится рассматривать случаи, в которых доступны машины с разной вычислительной мощью, так что конкретное задание может завершиться на одной из машин быстрее, чем на другой. Вопрос: как распределять задания между машинами в разнородной системе?

   Все эти проблемы продемонстрированы в следующей базовой модели. Допустим, имеется система, состоящая из m медленных и k быстрых машин. Быстрые машины могут выполнять за единицу времени вдвое больше работы, чем медленные. Имеется множество из n заданий; выполнение задания i занимает время t_i на медленной машине и время $\frac{1}{2}t_i$ на быстрой машине.

   Требуется назначить каждое задание на машину; как и прежде, целью является минимизация периода обработки, то есть максимума (по всем машинам) общего времени обработки заданий, назначенных на каждую машину.

7. Предложите алгоритм с полиномиальным временем, который обеспечивает распределение заданий между машинами с периодом обработки, превышающим оптимум не более чем в три раза.

   Вы консультируете коммерческий сайт, который ежедневно получает большое количество посещений. Каждому посетителю i, где $i \in \{1,2, ..., n\}$, сайт назначает значение v_i, представляющее ожидаемый доход, который может быть получен о данного клиента.

Каждому посетителю i при входе на сайт показывается одна из m возможных реклам $A_1, A_2, ..., A_m$. Нужно выбрать по одной рекламе для каждого посетителя, чтобы *каждая реклама* была показана в сумме множеству клиентов с достаточно большим общим весом. Таким образом, для заданного выбора одной рекламы на посетителя *охват* этой выборки определяется как минимум по 1, 2, ..., m общего веса всех клиентов, которым была показана реклама A_j.

Пример. Допустим, имеются шесть посетителей со значениями 3, 4, 12, 2, 4, 6 и $m = 3$ реклам. Тогда в этом экземпляре задачи можно обеспечить охват 9, показав рекламу A_1 посетителям 1, 2, 4, рекламу A_2 — посетителю 3 и рекламу A_3 — посетителям 5 и 6.

Конечной целью является подбор рекламы для каждого посетителя, обеспечивающей максимальный охват. К сожалению, задача оптимизации является *NP*-сложной (вам это доказывать не нужно). Итак, вместо поиска прямого решения мы попробуем аппроксимировать его.

(a) Предоставьте алгоритм с полиномиальным временем, аппроксимирующий максимальный охват в пределах множителя 2. Иначе говоря, если максимальный охват равен s, то ваш алгоритм должен выдавать подборку рекламы с охватом не менее $s/2$. При разработке алгоритма можно считать, что в ней нет посетителей, значение которых значительно превышает среднее; а именно: если обозначить $\overline{v} = \sum_{i=1}^{n} v_i$ общее значение по всем посетителям, можно считать, что нет посетителя со значением, превышающим $\overline{v}/(2m)$.

(b) Приведите пример экземпляра, для которого алгоритм, разработанный в части (a), не находит оптимального решения (то есть решения с максимальным охватом). Укажите в своем примере оптимальное решение и решение, которое будет найдено вашим алгоритмом.

8. Ваши друзья работают над системой, выполняющей планирование заданий на нескольких серверах в реальном времени. Они обратились к вам за помощью — им нужно как-то справиться с неэффективным кодом, который не может быть изменен.

   Ситуация выглядит так: при поступлении пакета заданий система распределяет их по серверам, используя простой жадный алгоритм распределения из раздела 11.1, обеспечивающий аппроксимацию в пределах множителя 2. Однако эта часть системы была написана уже 15 лет назад, а производительность оборудования возросла до такого уровня, что в экземплярах задачи, с которыми имеет дело система, обычно бывает возможно вычислить оптимальное решение.

   Проблема в том, что люди, отвечающие за администрирование системы, не позволяют изменить часть кода, реализующую жадный алгоритм распределения нагрузки, и заменить его кодом поиска оптимального решения. (Эта часть кода взаимодействует с множеством других модулей системы, поэтому риск возникновения проблем в случае замены слишком велик.)

   Немного поворчав, ваши друзья выдвигают альтернативную идею. Допустим, они могут написать маленький фрагмент кода, который получает описание

заданий, вычисляет оптимальное решение (так как они могут сделать это с экземплярами, встречающимися на практике), а потом передает задания жадному алгоритму в таком порядке, который заставит его расположить их оптимальным образом. Иначе говоря, они надеются переупорядочить входные данные так, чтобы при получении данных в новом порядке жадный алгоритм выдавал оптимальное решение.

Ваших друзей интересует: всегда ли это возможно? Они предлагают следующую гипотезу.

Для каждого экземпляра задачи распределения нагрузки из раздела 11.1 существует такой порядок заданий, что при обработке заданий в этом порядке жадный алгоритм распределения нагрузки строит распределение с минимально возможным периодом обработки.

Решите, истинна ли эта гипотеза или ложна; приведите доказательство или контрпример.

9. Рассмотрим следующую максимизирующую версию задачи о трехмерном сочетании. Для заданных непересекающихся множеств X, Y и Z, а также множества $T \subseteq X \times Y \times Z$ упорядоченных триплетов подмножество $M \subseteq T$ называется *трехмерным сочетанием*, если каждый элемент $X \cup Y \cup Z$ содержится не более чем в одном из этих триплетов. В *максимальной задаче о трехмерном сочетании* требуется найти трехмерное сочетание M с максимальным размером. (Размер сочетания, как обычно, определяется количеством содержащихся в нем триплетов. Если хотите, считайте, что $|X| = |Y| = |Z|$.)

   Предложите алгоритм с полиномиальным временем, который находит трехмерное сочетание с размером не менее $\frac{1}{3}$ от максимально возможного.

10. Предположим, имеется решетчатый граф G с размерами $n \times n$ (рис. 11.13).

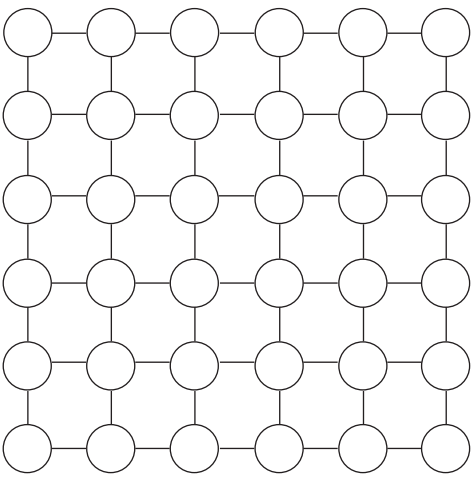


Рис. 11.13. Решетчатый граф

С каждым узлом v связывается вес $w(v)$ — неотрицательное целое число. Считайте, что веса всех узлов различны. Требуется найти такое независимое множество S узлов решетки, чтобы сумма весов узлов S была как можно выше. (Далее сумма весов узлов множества S будет называться его *общим весом*.)

Рассмотрим следующий жадный алгоритм для решения этой задачи.

```
Жадный алгоритм «выбора узла с наибольшим весом»:
   Начать с пустого множества S
   Пока в G остаются узлы
      Выберите узел v_i с максимальным весом
      Добавить v_i в S
      Удалить v_i и его соседей из G
   Конец Пока
   Вернуть S
```

(a) Пусть S — независимое множество, возвращаемое жадным алгоритмом выбора узла с наибольшим весом, а T — любое независимое множество в G. Покажите, что для каждого узла $v \in T$ либо $v \in S$, либо существует такой узел $v' \in S$, что $w(v) \leq w(v)$ и (v, v') является ребром G.

(b) Покажите, что жадный алгоритм выбора узла с наибольшим весом возвращает независимое множество с общим весом не менее $\frac{1}{4}$ от максимального общего веса любого независимого множества в решетчатом графе G.

11. Вспомните, что в задаче о рюкзаке имеются n предметов, каждый из которых обладает весом w_i и значением v_i. Также определяется предельный вес W; задача состоит в нахождении множества предметов S с наибольшим возможным значением при условии, что общий вес не превышает W, то есть $\sum_{i \in S} w_i \leq W$. Один из подходов к рассмотрению аппроксимирующих алгоритмов может быть таким: если существует подмножество O, общий вес которого равен $\sum_{i \in O} w_i \leq W$, а общее значение $\sum_{i \in O} v_i = V$ для некоторого V, то наш аппроксимирующий алгоритм найдет множество A с общим весом $\sum_{i \in A} w_i \leq W$ и общим значением не менее $\sum_{i \in A} v_i \geq V/(1+\varepsilon)$. Таким образом, алгоритм аппроксимирует лучшее значение, удерживая веса строго ниже W. (Конечно, возвращение множества O всегда является действительным решением, но так как задача является *NP*-полной, не стоит рассчитывать, что нам всегда удастся найти само множество O; граница аппроксимации $1 + \epsilon$ означает, что другие множества A, обладающие чуть меньшим значением, также могут быть действительными ответами.)

Но всем известно, что в чемодан всегда можно добавить еще немного вещей, если посидеть на нем, — другими словами, если слегка превысить допустимый предельный вес. Отсюда тоже вытекает способ формализации вопроса об аппроксимации задачи о рюкзаке, но это уже другая формализация.

Допустим, как и прежде, даны n предметов с весами и значениями, а также параметрами W и V; вам сообщают, что существует подмножество O, общий вес которого равен $\sum_{i \in O} w_i \leq W$, а общее значение равно $\sum_{i \in O} v_i = V$ для некоторого V. Для заданного фиксированного $\epsilon > 0$ разработайте алгоритм с полиномиальным временем, который находит подмножество таких элементов A, что $\sum_{i \in A} w_i \leq (1+\varepsilon)$ и $\sum_{i \in A} v_i \geq V$.

Другими словами, подмножество A должно достигать общего значения, не меньшего заданной границы V, но допускается превышение границы веса W на коэффициент $1 + \epsilon$.

Пример. Допустим, имеются четыре предмета со следующими весами и значениями:

$$(w_1, v_1) = (5, 3), (w_2, v_2) = (4, 6)$$

$$(w_3, v_3) = (1, 4), (w_4, v_4) = (6, 11)$$

Также заданы границы $W = 10$ и $V = 13$ (так как подмножество, состоящее из первых трех элементов, имеет общий вес 10 и значение 13). Наконец, $\epsilon = 0{,}1$. Это означает, что нужно найти (при помощи аппроксимирующего алгоритма) подмножество с весом не более $(1 + 0{,}1) * 10 = 11$ и значением не менее 13.

Одним из действительных решений станет подмножество, состоящее из первого и четвертого предметов, со значением $14 \geq 13$. (Обратите внимание: в этом случае достигается значение строго большее V, так как допускается легкое переполнение рюкзака.)

12. Рассмотрим следующую задачу. Имеется множество U из n узлов, которые могут рассматриваться как пользователи (например, пользователи, обращающиеся к веб-серверу или иному виду сервиса). Серверы должны размещаться в разных местах. Предположим, имеется множество из S возможных мест, подходящих для размещения серверов. Для каждого места $s \in S$ определяется плата за размещение установки сервера в этом месте $f_s \geq 0$. Ваша цель — обеспечить аппроксимированную минимизацию стоимости при предоставлении сервиса каждому пользователю. Пока что все очень похоже на задачу о покрытии множества: места s соответствуют множествам, вес множества s равен f_s, требуется выбрать совокупность множеств, покрывающую всех пользователей. Тем не менее существует один нюанс: пользователи $u \in U$ могут обслуживаться из разных мест, но за обслуживание пользователя u из места s взимается стоимость d_{us}. При очень высоком значении d_{us} обслуживание пользователя u с сервера в месте s нежелательно; в общем случае затраты d_{us} должны поощрять обслуживание пользователей с «ближних» серверов, насколько это возможно.

Итак, вопрос, который мы назовем «задачей о размещении серверов»: для заданных множеств U и S, а также цен f и d требуется выбрать подмножество мест $A \subseteq S$ для размещения серверов (со стоимостью $\sum_{s \in A} f_s$) и закрепить за каждым пользователем активный сервер, на котором его обслуживание обой-

дется дешевле всего: $\min_{s \in A} d_{us}$. Целью является минимизация общих затрат $\sum_{s \in A} f_s + \sum_{u \in U} \min_{s \in A} d_{us}$. Предложите $H(n)$-аппроксимацию для этой задачи. (Если все значения d_{us} равны 0 или бесконечны, то задача в точности соответствует задаче покрытия множеств: f_s — стоимость множества s, а $d_{us} = 0$, если узел u принадлежит множеству s, и *бесконечность в противном случае*).

Примечания и дополнительная литература

В области разработки аппроксимирующих алгоритмов для *NP*-сложных задач ведутся активные исследования; в частности, этой теме посвящен сборник статей под редакцией Хохбаума (Hochbaum, 1996) и работа Вазирани (Vazirani, 2001).

Автором жадного алгоритма распределения нагрузки и его анализа стал Грэм (Graham, 1966, 1969); в частности, он доказал, что когда задания предварительно сортируются в порядке убывания размера, жадный алгоритм обеспечивает распределение в границах коэффициента $\frac{4}{3}$ от оптимума. (В тексте приводится более простое доказательство ослабленной границы $\frac{3}{2}$).

При использовании более сложных алгоритмов для этой задачи доказываются даже более сильные гарантии аппроксимации (Hochbaum, Shmoys, 1987; Hall, 1996). Методы, используемые усиленными аппроксимирующими алгоритмами распределения нагрузки, также тесно связаны с методом, описанным в тексте при разработке аппроксимаций произвольной точности для задачи о рюкзаке.

Аппроксимирующий алгоритм для задачи о выборе центров следует методологии Хохбаума и Шмойса (Hochbaum, Shmoys, 1985), а также Дайера и Фризе (Dyer, Frieze, 1985). Другие геометрические задачи выбора мест этого типа обсуждаются Берном и Эппстейном (Bern, Eppstein, 1996), а также в сборнике под редакцией Дрезнера (Drezner, 1995).

Жадный алгоритм для задачи покрытия множества и ее анализ были выполнены независимо Джонсоном (Johnson, 1974), Ловашем (Lovász, 1975) и Хваталом (Chvatal, 1979). Дальнейшие результаты для задачи покрытия множества обсуждаются в сборнике Хохбаума (Hochbaum, 1996).

Как упоминалось в тексте, метод назначения цены при разработке аппроксимирующих алгоритмов также называется *прямо-двойственным методом* и может быть обоснован средствами линейного программирования. Последний метод является темой работы Геманса и Уильямсона (Goemans, Williamson, 1996). Алгоритм назначения цены для аппроксимации взвешенной задачи о вершинном покрытии предложили Бар-Иегуда и Ивен (Bar-Yehuda, Even, 1981). Жадный алгоритм для задачи о непересекающихся путях разработан Клейнбергом и Тардос (Kleinberg, Tardos, 1995); аппроксимирующий алгоритм на базе цен для случая с повторным использованием ребер несколькими путями предложили Авербух, Азар и Плоткин (Awerbuch, Azar, Plotkin, 1993). Алгоритмы были разработаны и для многих других разновидностей задачи о непересекающихся путях; в сборнике под редакцией

Корте и др. (Korte, 1990) обсуждаются случаи, имеющие оптимальное решение за полиномиальное время, а у Плоткина (Plotkin, 1995) и Клейнберга (Kleinberg, 1996) приводится анализ текущих работ по аппроксимации.

Округляющий алгоритм линейного программирования для взвешенной задачи о вершинной покрытии предложен Хохбаумом (Hochbaum, 1982). Округляющий алгоритм для обобщенной задачи распределения нагрузки разработали Ленстра, Шмойс и Тардос (Lenstra, Shmoys, Tardos, 1990); другие результаты в этом направлении представлены в обзоре Холла (Hall, 1996). Как упоминалось в тексте, эти два результата демонстрируют распространенный метод разработки аппроксимирующих алгоритмов: формулировка задачи в контексте целочисленного программирования, преобразование ее во взаимосвязанную (но не эквивалентную) задачу линейного программирования и последующее округление полученного решения. Многие другие применения этого метода обсуждаются у Вазирани (Vazirani, 2001).

Два других эффективных метода разработки аппроксимирующих алгоритмов — локальный поиск и рандомизация; эти связи обсуждаются в следующих двух главах.

В книге не затронута тема *неаппроксимируемости*. По аналогии с тем, как можно доказать, что заданная *NP*-сложная задача может быть аппроксимирована до некоторого множителя за полиномиальное время, также иногда удается установить нижние границы, которые показывают, что если задача может быть аппроксимирована лучше, чем до некоторого множителя c за полиномиальное время, то она может быть решена оптимально; тем самым будет доказано, что $P = NP$. В этой области проводится растущее число исследований, устанавливающих пределы аппроксимируемости для многих *NP*-сложных задач. В некоторых случаях эти положительные и отрицательные результаты идеально сочетаются и образуют порог аппроксимации, устанавливая для некоторых задач наличие аппроксимирующего алгоритма до множителя c с полиномиальным временем и невозможность более эффективного решения при условии $P \neq NP$. Некоторые ранние результаты в области неаппроксимируемости доказывались без особых трудностей, но в последних работах появились мощные и весьма нетривиальные методы. Эта тема рассматривается в работе Ароры и Лунда (Arora, Lund, 1996).

Примечания к упражнениям

Упражнения 4 и 12 основаны на результатах Дорита Хохбаума. Упражнение 8 основано на результатах Сартаджа Сани, Оскара Ибарры и Чул Кима, а также Дорита Хохбаума и Дэвида Шмойса.

Глава 12
Локальный поиск

В двух предыдущих главах рассматривались методы работы с вычислительно-неразрешимыми задачами: в главе 10 — выявление структурированных особых случаев *NP*-сложных задач, а в главе 11 — разработка аппроксимирующих алгоритмов с полиномиальным временем. В этой главе рассматривается третья и последняя тема, относящаяся к этой области: разработка *алгоритмов локального поиска*.

Локальный поиск — чрезвычайно универсальный прием; этим термином описываются любые алгоритмы, которые последовательно «исследуют» пространство возможных решений, перемещаясь за один шаг от текущего решения к другому, «ближнему». Общность и гибкость этого метода имеют свои преимущества: алгоритм на базе локального поиска можно без особых трудностей разработать почти для любой вычислительно сложной задачи; с другой стороны, часто бывает очень трудно сказать что-нибудь конкретное или доказуемое о качестве решений, находимых алгоритмом локального поиска, и, соответственно, очень сложно определить, хороший или плохой алгоритм локального поиска используется в каждом конкретном случае.

Эти достоинства и недостатки будут отражены в ходе рассмотрения локального поиска в этой главе. Алгоритмы локального поиска обычно представляют собой эвристики, предназначенные для нахождения хороших (но не обязательно оптимальных) решений вычислительных задач, и для начала мы поговорим о том, как выглядит поиск таких решений на глобальном уровне. Полезная интуитивно понятная аналогия существует в принципе минимизации энергии в физике, поэтому мы сначала исследуем этот вопрос. Стиль изложения в этой главе несколько отличается от того, что был в книге до настоящего момента; мы представим некоторые алгоритмы, обсудим их на качественном уровне, но честно признаем, что мало что из этого мы можем доказать.

Впрочем, в отдельных случаях нам удастся доказать некоторые свойства алгоритмов локального поиска и связать их производительность с оптимальным решением. Эта тема займет центральное место в завершающей части этой главы: мы начнем с рассмотрения случая (динамики нейронных сетей Хопфилда), в котором локальный поиск предоставляет естественную точку зрения на поведение сложного процесса; затем мы сосредоточим внимание на *NP*-сложных задачах, для которых локальный поиск помогает разработать эффективные алгоритмы с доказуемыми гарантиями аппроксимации. Глава завершается обсуждением

других типов локального поиска: *динамикой наилучших ответов* и *равновесиями Нэша*, естественным образом встречающимися при изучении систем с множеством взаимодействующих агентов.

12.1. Задача оптимизации в перспективе

Основные концепции локального поиска были разработаны учеными, мыслившими аналогиями с физикой. Рассматривая широкий спектр сложных вычислительных задач, требующих минимизации некоторой величины, они рассуждали следующим образом: физические системы постоянно решают задачу минимизации, стремясь к минимуму потенциальной энергии. Чему можно научиться, наблюдая за минимизацией в природе? Не подскажет ли она какие-нибудь новые алгоритмы?

Потенциальная энергия

Если бы мир выглядел так, как нам рассказывают в начальном курсе механики, он состоял бы исключительно из хоккейных шайб, скользящих по льду, и шаров, скатывающихся по наклонным поверхностям. Шайба скользит, потому что ее толкнули; но почему шары скатываются вниз? Из ньютоновской механики известно, что шар стремится минимизировать свою *потенциальную энергию*. В частности, если шар имеет массу m и падает на расстояние h, он теряет потенциальную энергию, пропорциональную mh. Итак, если отпустить шар у верхнего края воронкообразного углубления (рис. 12.1), его потенциальная энергия будет минимальной в самой нижней точке.

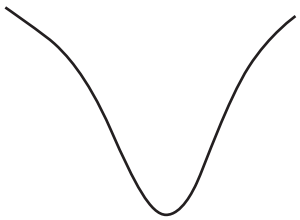


Рис. 12.1. Когда поверхность потенциальной энергии имеет структуру простой воронки, найти нижнюю точку достаточно легко

При усложнении структуры поверхности проявляются дополнительные сложности. Возьмем «двойную воронку» на рис. 12.2. Точка A находится ниже точки B, поэтому она является предпочтительной для нахождения шара в состоянии покоя. Но если шар скатывается с точки C, он не сможет преодолеть барьер между двумя воронками и окажется в B. В таких случаях говорят, что шар попал в *локальный минимум*: он находится в самой нижней точке, если рассматривать окрестности текущего положения; но если отступить и взглянуть на ситуацию в целом, мы видим, что *глобальный минимум* не достигнут.

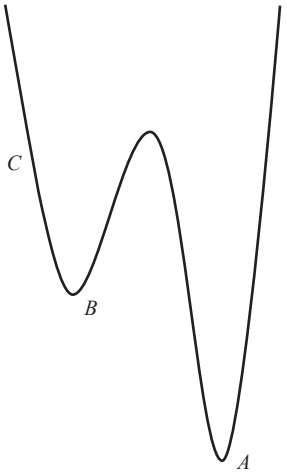


Рис. 12.2. Большинство поверхностей сложнее простых воронок; например, в этой «двойной воронке» существуют большой глобальный минимум и меньший локальный минимум

Конечно, очень большие физические системы также должны стремиться к минимизации своей энергии. Представьте, например, что вы взяли несколько граммов однородного вещества, нагрели его и изучаете его поведение во времени. Чтобы точно отразить потенциальную энергию образца, теоретически необходимо представить поведение каждого атома вещества и его взаимодействие с ближними атомами. Но также будет полезно рассмотреть свойства системы в целом (то есть всего образца), и здесь на помощь приходит статистическая механика. Мы еще вернемся к статистической механике, а пока просто заметим, что концепция «поверхности потенциальной энергии» предоставляет полезную и наглядную аналогию для процесса, который используется в больших физических системах для минимизации энергии. Следовательно, хотя в действительности для представления настоящей поверхности потенциальной энергии потребовалось бы многомерное пространство с огромным количеством измерений, мы можем воспользоваться одномерным условным представлением для обсуждения различий между локальными и глобальными энергетическими минимумами, окружающих их «потенциальных ям» и «высоты» энергетических барьеров между ними.

Остывание расплавленного материала с попыткой формирования идеального кристаллического тела в действительности представляет собой процесс перехода совокупности атомов в состояние глобального минимума потенциальной энергии. Этот процесс может быть очень сложным, и большое количество локальных минимумов в типичной поверхности потенциальной энергии представляет многочисленные отклонения, которые могут привести к сбоям при поиске глобального минимума системой. Следовательно, вместо простого примера на рис. 12.2, который содержит всего один неправильный вариант, нам следует беспокоиться о поверхностях, схематически изображенных на рис. 12.3. Это своего рода «гребенка», в которой локальные минимумы подстерегают систему на всем пути от верха до низа.

Рис. 12.3. Обобщенная поверхность потенциальной энергии может содержать множество локальных минимумов, усложняющих поиск глобального минимума, как в изображенной на рисунке «гребенке»

Связь с оптимизацией

Этот подход к минимизации энергии в действительности базируется на нескольких основных положениях: физическая система может находиться в одном из многочисленных состояний; ее энергия является функцией текущего состояния; небольшое возмущение в заданном состоянии приводит к «соседнему» состоянию. Структура связи этих соседних состояний вместе со структурой энергетической функции определяет энергетическую поверхность.

Под этим углом мы снова взглянем на задачу вычислительной минимизации. В типичной задаче такого рода имеется большое (как правило, имеющее экспоненциальный размер) множество C возможных решений. Также понадобится *функция стоимости* $c(\cdot)$, оценивающая качество каждого решения; для решения $S \in C$ его стоимость записывается в виде $c(S)$. Целью является поиск решения $S^* \in C$ с наименьшим возможным значением (S^*).

До настоящего момента мы рассматривали подобные задачи именно так. А теперь добавим в решения понятие *соседских отношений*, отражающих идею о том, что одно решение S' может быть получено незначительным изменением другого решения S. Запись $S \sim S'$ означает, что S' является *соседним решением* S, а множество соседей S, то есть $\{S' : S \sim S'\}$, будет обозначаться $N(S)$. Нас в первую очередь интересуют симметричные соседские отношения, хотя основные рассматриваемые принципы также распространяются и на асимметричные соседские отношения. Здесь принципиально то, что хотя множество C возможных решений и функция стоимости $c(\cdot)$ предоставляются в спецификации задачи, мы свободны выбирать соседские отношения по своему усмотрению.

Алгоритм локального поиска получает эту конфигурацию, включая соседские отношения, и работает по следующей высокоуровневой схеме. Он постоянно поддерживает текущее решение $S \in C$. На каждом шаге он выбирает соседа S' решения S, объявляет S' новым текущим решением и повторяет процесс. На протяжении всего выполнения алгоритма запоминается решение с наименьшей стоимостью из всех

встречавшихся; итак, по мере выполнения постепенно находятся все лучшие и лучшие решения. Суть алгоритма локального поиска заключается в выборе соседских отношений и в формулировке правила выбора соседнего решения на каждом шаге.

Итак, соседские отношения можно рассматривать как определяющий (обычно ненаправленный) граф множества всех возможных решений, в котором ребра соединяют соседние пары решений. В этом случае алгоритм локального поиска может рассматриваться как обход графа с попыткой перемещения в направлении хорошего решения.

Локальный поиск в задаче о вершинном покрытии

Без конкретной задачи все эти рассуждения выглядят довольно туманно; рассмотрим работу локального поиска на примере задачи о вершинном покрытии. Учтите, что задача о вершинном покрытии является хорошим примером, но существует много других задач оптимизации, которые с таким же успехом можно использовать в качестве примера.

Итак, имеется граф $G = (V, E)$; множество C возможных решений состоит из всех подмножеств S множества V, образующих вершинные покрытия. Например, всегда выполняется $V \in C$. Стоимость $c(S)$ вершинного покрытия S равна его размеру; таким образом, минимизация стоимости вершинного покрытия эквивалентна нахождению покрытия с минимальным размером. Наконец, в наших примерах алгоритмов локального поиска будет использоваться очень простое соседское отношение: мы говорим, что $S \sim S'$, если решение S' может быть получено из S добавлением или удалением одного узла. Таким образом, наши алгоритмы локального поиска будут обходить пространство возможных вершинных покрытий, добавляя или удаляя из текущего решения узел на каждом шаге и пытаясь найти как можно меньшее вершинное покрытие.

У этого соседского отношения есть одно полезное свойство:

(12.1) Каждое вершинное покрытие S имеет не более n соседних решений.

Утверждение доказывается попросту тем, что каждое соседнее решение S получается добавлением или удалением узла. Из (12.1) следует, что в процессе выбора можно эффективно исследовать все возможные соседние решения S.

Для начала рассмотрим очень простой алгоритм локального поиска — *метод градиентного спуска*. Градиентный спуск начинает с полного вершинного покрытия V и использует следующее правило выбора соседнего решения.

Обозначим S текущее решение. Если существует соседнее решение S' со строго меньшей стоимостью, то выбрать соседа, имеющего как можно меньшую стоимость. В противном случае завершить работу алгоритма.

Итак, метод градиентного спуска двигается строго вниз, пока может; когда дальнейшее движение становится невозможным, он останавливается.

Мы видим, что градиентный спуск завершается точно в тех решениях, которые являются *локальными минимумами*: то есть в таких решениях S, что для всех соседних S' выполняется $c(S) \leq c(S')$.

Это определение очень точно соответствует нашему понятию локального минимума на энергетических поверхностях: это те точки, для которых одношаговое возмущение не улучшает функции стоимости.

Как наглядно представить поведение алгоритма локального поиска в контексте энергетических поверхностей, которые были представлены ранее? Начнем с градиентного спуска. Конечно, простейший экземпляр вершинного покрытия представляется n-узловым графом без ребер. Пустое множество является оптимальным решением (ребер для покрытия нет), а градиентный спуск превосходно справляется с нахождением этого решения: он начинает с полного множества вершин V и продолжает удалять узлы, пока не останется ни одного. В самом деле, множество вершинных покрытий для этого графа без ребер естественно соответствует воронке на рис. 12.1: единственный локальный минимум также является глобальным минимумом, и к нему существует нисходящий путь от любой точки.

Когда метод градиентного спуска работает неправильно? Рассмотрим «граф-звезду» G, состоящий из узлов $x_1, y_1, y_2, ..., y_{n-1}$, в котором узел x_1 соединен с каждым из узлов y_i. Минимальное вершинное покрытие для G представляет собой одноэлементное множество $\{x_1\}$, а градиентный спуск может достигнуть этого решения, последовательно удаляя $y_1, ..., y_{n-1}$ в любом порядке. Но если градиентный спуск начнет с удаления x_1, он немедленно оказывается в тупике: ни один узел y_i не может быть удален без разрушения свойства вершинного покрытия, поэтому единственным соседним решением является полное множество узлов V, обладающее более высокой стоимостью. Алгоритм «застревает» в локальном минимуме $\{y_1, y_2, ..., y_{n-1}\}$, обладающем очень высокой стоимостью по сравнению с глобальным минимумом.

В графическом виде мы оказываемся в ситуации, соответствующей «двойной воронке» на рис. 12.2. Глубокая воронка соответствует оптимальному решению $\{x_1\}$, а мелкая воронка соответствует локальному минимуму $\{y_1, y_2, ..., y_{n-1}\}$. Спуск по неправильно выбранному склону, выбранному в самом начале, может привести к неправильному минимуму. Ситуация легко обобщается до двух минимумов с любыми относительными глубинами. Допустим, имеется двудольный граф G с узлами $x_1, x_2, ..., x_k$ и $y_1, y_2, ..., y_l$, where $k < \ell$ и существует ребро из каждого узла вида x_i в узел вида y_j. Тогда существуют два локальных минимума, соответствующих вершинным покрытиям $\{x_1, ..., x_k\}$ и $\{y_1, ..., y_l\}$. Какой из этих минимумов будет обнаружен при выполнении градиентного спуска, зависит исключительно от того, будет ли первым удален элемент вида x_i или y_j.

Для более сложных графов часто бывает полезно подумать о том, какую поверхность они представляют; и наоборот, иногда можно взглянуть на поверхность и подумать о том, какой граф мог бы ее породить.

Например, какой граф мог бы породить экземпляр задачи о вершинном покрытии с поверхностью вроде «гребенки» на рис. 12.3? Одним из таких графов мог бы быть n-узловой путь, в котором n — нечетное число, а узлы помечены в порядке $v_1, v_2, ..., v_n$. Уникальное минимальное вершинное покрытие S^* состоит из всех узлов v_i, для которых i четно. Наряду с ним существует множество локальных оптимумов. Например, рассмотрим вершинное покрытие $\{v_2, v_3, v_5, v_6, v_8, v_9, ...\}$, в котором опущен каждый третий узел. Это вершинное покрытие существенно больше S^*; при этом из него невозможно удалить ни один узел, сохранив покрытие всех ребер.

Как выясняется, найти минимальное вершинное покрытие S^* методом градиентного спуска, начиная с полного вершинного покрытия V, очень трудно: после удаления одного узла v_i с четным значением i возможность нахождения глобального оптимума S^* полностью теряется. Таким образом, различие между четными и нечетными узлами создает множество ошибочных путей для локального поиска, что и придает общей поверхности вид «гребенки». Конечно, между впадинами на рисунке и локальными оптимумами не существует прямого соответствия; как мы предупреждали ранее, рис. 12.3 всего лишь дает наглядное представление о происходящем.

Но мы видим, что даже для графов с очень простой структурой градиентный спуск слишком прямолинеен для алгоритма локального поиска. Обратимся к более эффективным алгоритмам локального поиска, которые используют аналогичные соседские отношения, но включают возможность «выхода» из локального минимума.

12.2. Алгоритм Метрополиса и имитация отжига

Первые идеи улучшенного алгоритма локального поиска были представлены в работе Метрополиса, Розенблата и Теллера (Metropolis, Rosenbluth, Teller, 1953). Они рассматривали задачу моделирования поведения физической системы в соответствии с принципами статистической механики. Базовая модель из этой области подразумевает, что вероятность нахождения физической системы в состоянии с энергией E пропорциональна функции Гиббса–Больцмана $e^{-E/(kT)}$, где $T > 0$ — температура, а $k > 0$ — константа. Присмотримся к этой функции. Для любой температуры T функция монотонно убывает по энергии E, поэтому из формулы следует, что физическая система находится в низкоэнергетическом состоянии с большей вероятностью, чем в высокоэнергетическом. Теперь рассмотрим эффект температуры T; при малых T вероятность низкоэнергетического состояния существенно выше вероятности высокоэнергетического состояния. Однако при высокой температуре разность между двумя вероятностями становится очень малой и система почти с равной вероятностью может оказаться в любом состоянии.

Алгоритм Метрополиса

Метрополис вместе с другими авторами предложил следующий метод пошагового моделирования системы при фиксированной температуре T. В любой момент времени модель хранит текущее состояние системы и пытается сгенерировать новое состояние, применяя возмущение к текущему состоянию. Будем считать, что нас интересуют только состояния системы, «достижимые» из некоторого фиксированного исходного состояния через последовательность мелких возмущений; предполагается, что множество таких состояний C конечно. За один шаг сначала генерируется небольшое случайное возмущение в текущем состоянии S системы,

приводящее к новому состоянию S'. Пусть $E(S)$ и $E(S')$ обозначают энергии S и S' соответственно. Если $E(S') \leq E(S)$, то текущее состояние заменяется на S'. В противном случае пусть $\Delta E = E(S') - E(S) > 0$. Текущее состояние заменяется состоянием S' с вероятностью $e^{-\Delta E/(kT)}$, а в противном случае текущим состоянием остается S.

Метрополис и др. доказали, что их моделирующий алгоритм обладает следующим свойством (чтобы не отвлекаться надолго, мы опускаем доказательство; на самом деле оно является прямым следствием некоторых базовых свойств случайных блужданий):

(12.2) Обозначим

$$Z = \sum_{S \in C} e^{-E(S)/(kT)}.$$

Пусть для состояния S запись $f_S(t)$ обозначает долю первых t шагов, в которых модель пребывает в состоянии S. Тогда предел $f_S(t)$ при стремлении t к ∞, с вероятностью, стремящейся к 1, равен $\frac{1}{Z} \cdot e^{-E(S)/(kT)}$.

Именно такой факт нам и нужен; фактически он означает, что модель проводит приблизительно нужное время в каждом состоянии в соответствии с уравнением Гиббса–Больцмана.

Чтобы использовать эту общую схему для разработки алгоритма локального поиска для задач минимизации, можно воспользоваться аналогией из раздела 12.1, в которой состояния системы соответствуют потенциальным решениям, а энергия соответствует стоимости. Работа алгоритма Метрополиса обладает парой свойств, очень полезных для алгоритма локального поиска: он склонен к «нисходящим» перемещениям, но также допускает небольшие «восходящие» перемещения с малой вероятностью. Такой алгоритм сможет двигаться дальше даже при заходе в локальным минимум. Более того, как выражено в (12.2), он глобально смещен в направлении решений с меньшей стоимостью. Ниже приводится конкретная формулировка алгоритма Метрополиса для задачи минимизации.

```
Начать с исходного решения S_0, констант k и T
За один шаг:
   Пусть S – текущее решение
   Случайно выбрать S' среди соседей S с равномерным распределением
   Если c(S') ≤ c(S)
      Обновить S ← S'
   Иначе
      С вероятностью e^(-(c(S')-c(S))/(kT))
         Обновить S ← S'
      Иначе
         Оставить S без изменений
   Конец Если
```

Таким образом, для экземпляра задачи о вершинном покрытии, состоящего из графа-звезды из раздела 12.1 (x_1 соединяется с каждым из узлов $y_1, ..., y_{n-1}$), мы видим, что алгоритм Метрополиса быстро выходит из локального минимума, воз-

никающего при удалении x_1: соседнее решение, в котором x_1 возвращается обратно, будет сгенерировано и принято с положительной вероятностью. В более сложных графах алгоритм Метрополиса тоже может до некоторой степени исправить неверные решения, принятые в ходе выполнения.

В то же время алгоритм Метрополиса не всегда ведет себя так, как нужно, причем даже в очень простых ситуациях. Вернемся к самому первому из рассмотренных графов — графу G без ребер. Градиентный спуск решает этот экземпляр без всяких проблем, последовательно удаляя узлы, пока не удалит все. Но хотя алгоритм Метрополиса начинает работать именно так, при приближении к глобальному максимуму он начинает буксовать. Рассмотрим ситуацию, в которой текущее решение содержит только c узлов, где c намного меньше общего количества узлов n. С очень высокой вероятностью соседнее решение, сгенерированное алгоритмом Метрополиса, будет иметь размер $c + 1$, а не $c - 1$, и с разумной вероятностью это перемещение вверх будет принято. Таким образом, по мере работы алгоритма сокращать размер вершинного покрытия становится все труднее и труднее; при приближении к дну воронки начинаются «метания».

Это поведение также проявляется и в более сложных ситуациях, и не столь очевидными способами; конечно, странно видеть его в таком простом случае. Чтобы понять, как справиться с проблемой, мы вернемся к физической аналогии, заложенной в основу алгоритма Метрополиса, и спросим: какой смысл имеет параметр температуры в контексте оптимизации?

T можно рассматривать как одномерную «рукоятку», поворот которой управляет готовностью алгоритма принимать повышающие перемещения. При очень больших T вероятность принять повышающее перемещение стремится к 1, и алгоритм Метрополиса ведет себя как случайное блуждание, фактически не учитывающее функцию стоимости. С другой стороны, при приближении T к 0 повышающие перемещения почти никогда не принимаются и алгоритм Метрополиса ведет себя практически идентично градиентному спуску.

Имитация отжига

Ни одна из крайних температур — ни очень низкая, ни очень высокая — не является эффективным методом решения задач минимизации в целом. Этот принцип также проявляется в физических системах: если взять твердое тело и нагреть его до очень высокой температуры, трудно ожидать сохранения стройной кристаллической структуры, даже если она предпочтительна с энергетической точки зрения; и это можно объяснить большим значением kT в выражении $e^{-E(S)/(kT)}$, с которым огромное количество менее выгодных состояний становится слишком вероятным. С этой же точки зрения можно рассматривать «метания» алгоритма Метрополиса в простом экземпляре задачи о вершинном покрытии: он пытается найти самое низкое энергетическое состояние при слишком высокой температуре, когда все конкурирующие состояния имеют слишком высокую вероятность. С другой стороны, если взять расплав и очень быстро заморозить его, не стоит также рассчитывать на формирование идеальной кристаллической структуры, скорее вы

получите деформированную структуру с множеством дефектов. Дело в том, что при очень малых T мы слишком близко подходим к области градиентного спуска, и система захватывается в одном из многочисленных провалов своей гребенчатой энергетической поверхности. Интересно заметить, что при очень малых T утверждение (12.2) показывает, что в пределе большая часть времени случайных блужданий проводится в низшем энергетическом состоянии. Проблема в том, что случайное блуждание тратит слишком много времени, чтобы хотя бы отдаленно приблизиться к этому пределу.

В начале 1980-х годов ученые изучали связи между минимизацией энергии и комбинаторной оптимизацией. Киркпатрик, Гелатт и Веччи (Kirkpatrick, Gelatt, Vecchi, 1983) размышляли над обсуждаемой темой и задали следующий вопрос: как решить эту задачу для физических систем и какой тип алгоритма предполагает такое решение? В физических системах для перевода материала в кристаллическое состояние используется процесс, называемый *отжигом*: материал постепенно охлаждается с очень высокой температуры, что дает ему достаточно времени для достижения равновесия на промежуточных убывающих температурах. Таким образом удается избежать энергетических минимумов, с которыми материал сталкивается на всем протяжении процесса охлаждения, и в конечном итоге достичь глобального оптимума.

Этот процесс можно попытаться смоделировать на алгоритмическом уровне; так появился алгоритмический метод, известный как *имитация отжига*. Метод имитации отжига основан на выполнении алгоритма Метрополиса с постепенным снижением значения T в ходе выполнения. Конкретный способ обновления T называется по естественным причинам *планом охлаждения*; при его планировании учитывается целый ряд факторов. Формально план охлаждения представляет собой функцию τ, отображающую $\{1, 2, 3, ...\}$ на положительные вещественные числа; на итерации i алгоритма Метрополиса в определении вероятности используется температура $T = \tau(i)$.

На качественном уровне очевидно, что имитация отжига допускает большие изменения в решении на ранних стадиях его выполнения, при высокой температуре. Затем в процессе поиска температура понижается, чтобы снизить вероятность отмены уже происшедших изменений. Имитация отжига также может рассматриваться как попытка оптимизации альтернативы, следующей из (12.2). Согласно (12.2), значения T, сколь угодно близкие к 0, обеспечивают наивысшую вероятность решений с минимальной стоимостью; однако (12.2) само по себе ничего не говорит о скорости сходимости используемых ей функций $f_S(t)$. Как выясняется, эти функции сходятся намного быстрее для больших значений T; чтобы быстро найти решения с минимальной стоимостью, полезно ускорить сходимость, начав процесс при больших T, с последующим сокращением его для повышения вероятности оптимальных решений. Хотя, насколько нам известно, физические системы достигают минимального энергетического состояния через процесс отжига, метод имитации отжига не предоставляет гарантий нахождения оптимального решения. Чтобы понять, почему это так, рассмотрим ситуацию с двойной воронкой на рис. 12.2. Если две воронки занимают равную площадь, то при высоких температурах система с равной вероятностью может оказаться

в любой из двух. При снижении температуры переходы между двумя воронками постоянно усложняются. Нет никаких гарантий, что в конце процесса система окажется на дне более глубокой воронки.

В области имитации отжига остается много нерешенных задач, связанных как с доказательством свойств ее поведения, так и с определением диапазона настроек, в котором этот метод хорошо работает на практике. Некоторые общие аспекты основаны на вероятностных задачах, выходящих за рамки книги. Проведя некоторое время за изучением локального поиска на очень общем уровне, в нескольких ближайших разделах мы займемся изучением практических ситуаций, в которых возможны достаточно сильные гарантии относительно поведения алгоритмов локального поиска и находимых ими локальных оптимумов.

12.3. Применение локального поиска в нейронных сетях Хопфилда

До настоящего момента мы рассматривали локальный поиск как метод поиска глобального оптимума в вычислительной задаче. Однако в отдельных случаях тщательный анализ спецификации задачи показывает, что в действительности требуется найти произвольный *локальный* оптимум. Следующая задача дает пример такой ситуации.

Задача

В этом разделе рассматривается задача поиска устойчивых конфигураций в нейронных сетях Хопфилда — простой модели ассоциативной памяти, в которой большое количество элементов объединяется в сеть, а соседние элементы пытаются согласовывать свои состояния. Конкретно сеть Хопфилда может рассматриваться как ненаправленный граф $G = (V, E)$ с целочисленным весом для каждого ребра e; каждый вес может быть как положительным, так и отрицательным. *Конфигурация* сети S определяется как присваивание значения -1 или $+1$ каждому узлу u; это значение будет называться *состоянием* s_u узла u. Смысл конфигурации заключается в том, что каждый узел, представляющий элемент нейронной сети, пытается выбрать одно из двух возможных состояний («да» или «нет»; «истина» или «ложь»), и этот выбор зависит от состояния соседей. Каждое ребро сети устанавливает требование к своим конечным точкам: если u соединяется с v ребром отрицательного веса, то u и v находятся в одинаковом состоянии, а если u соединяется с v ребром положительного веса, то u и v стремятся находиться в противоположных состояниях. Абсолютное значение $|w_e|$ обозначает *силу* требования; мы будем называть $|w_e|$ *абсолютным весом* ребра e.

К сожалению, в системе может не быть единой конфигурации, соблюдающей требования всех ребер. Для примера рассмотрим три узла a, b и c, взаимно соединенных ребрами с весом 1. Какую бы конфигурацию мы ни выбрали, два ребра

будут находиться в одинаковом состоянии, нарушая требование о противоположности состояний.

С учетом этого обстоятельства мы немного снизим уровень требований. В отношении заданной конфигурации ребро $e = (u, v)$ называется *хорошим*, если устанавливаемое им требование соблюдается состояниями его конечных точек: либо $w_e < 0$ и $s_u = s_v$, либо $w_e > 0$ и $s_u \neq s_v$. В противном случае ребро e называется *плохим*. Заметим, что условие «ребро e является хорошим» очень компактно выражается в следующем виде: $w_e s_u s_v < 0$. Узел u в заданной конфигурации называется *реализованным*, если общий абсолютный вес всех хороших ребер, инцидентных u, по крайней мере не меньше общего абсолютного веса всех плохих ребер, инцидентных u. Это определение можно записать в виде

$$\sum_{v:e=(u,v)\in E} w_e s_u s_v \leq 0.$$

Наконец, конфигурация называется *устойчивой*, если все узлы в ней реализованы.

Почему такие конфигурации называются «устойчивыми»? Термин происходит из рассмотрения сети с точки зрения отдельного узла u. Сам по себе узел u имеет выбор только между состояниями −1 и +1; как и все узлы, он хочет учесть как можно больше требований ребер (в соответствии с метрикой абсолютного веса). Допустим, узел u спрашивает: следует ли мне перейти в противоположное состояние? Мы видим, что если u изменяет состояние (при сохранении состояний всех остальных узлов), то все хорошие ребра, инцидентные u, становятся плохими, а все плохие ребра, инцидентные u, становятся хорошими. Итак, чтобы максимизировать вес хороших ребер, находящихся под его прямым управлением, узел u должен перейти в обратное состояние в том, и только в том случае, если он не реализован. Другими словами, устойчивой конфигурацией является та, в которой нет отдельных узлов, для которых были бы причины перейти в противоположное состояние.

А теперь мы приходим к основному вопросу: всегда ли в сети Хопфилда существует устойчивая конфигурация, и если да, то как ее найти?

Разработка алгоритма

Алгоритм, который будет разработан в этом разделе, доказывает следующий результат.

(12.3) У каждой сети Хопфилда имеется устойчивая конфигурация, которая может быть найдена за время, полиномиальное по n и $W = \sum_e |w_e|$.

Вы увидите, что устойчивые конфигурации крайне естественно встречаются в качестве локальных оптимумов определенных процедур локального поиска в сетях Хопфилда. Чтобы убедиться в том, что утверждение (12.3) не так уж тривиально, стоит заметить, что оно перестает быть истинным при некоторых естественных модификациях модели. Например, представьте, что мы определяем направленную сеть Хопфилда точно так, как описано выше, но с одним исключением — каждое ребро является направленным, а каждый узел определяет, являет-

ся он реализованным или нет, проверяя только те ребра, для которых он является начальным. В этом случае сеть может и не иметь устойчивой конфигурации. Для примера возьмем направленную версию сети из трех узлов, упоминавшуюся ранее: имеются три узла a, b, c, соединенные направленными ребрами (a, b), (b, c), (c, a), все ребра имеют вес 1. Если все узлы находятся в одном состоянии, все они будут нереализованными; а если состояние одного узла отличается от состояния двух других, то узел, находящийся непосредственно перед ним, будет нереализованным. Следовательно, в этой направленной сети не существует конфигурации, в которой все узлы были бы реализованы.

Очевидно, доказательство (12.3) должно где-то зависеть от ненаправленной природы сети.

Чтобы доказать (12.3), мы проанализируем простую итеративную процедуру поиска устойчивой конфигурации, которую назовем *алгоритмом переключения состояния*.

```
Пока текущая конфигурация не является устойчивой
   Должен существовать нереализованный узел
   Выбрать нереализованный узел u
   Переключить состояние u
Конец Пока
```

Пример выполнения алгоритма, приводящий к устойчивой конфигурации, изображен на рис. 12.4.

Анализ алгоритма

Очевидно, если только что определенный нами алгоритм переключения состояния завершился, была достигнута устойчивая конфигурация. Впрочем, не так очевидно то, что он действительно завершится. В предыдущем примере с направленным графом этот процесс будет просто перебирать три узла, бесконечно переключая их состояния. Сейчас мы докажем, что алгоритм переключения состояния всегда завершается, и приведем границу для количества итераций, необходимых для завершения. Тем самым будет доказано утверждение (12.3). Ключом к доказательству того, что этот процесс завершается, служит идея, использованная в нескольких предыдущих ситуациях: нужно найти метрику прогресса, то есть величину, которая строго увеличивается с каждым переключением состояния и имеет абсолютную верхнюю границу. Она может быть использована для ограничения количества итераций.

Вероятно, самой естественной метрикой прогресса могло бы стать количество реализованных узлов: если оно увеличивается при каждом переключении нереализованного узла, процесс будет выполняться не более n итераций перед завершением в устойчивой конфигурации. К сожалению, эта метрика не подходит. Действительно, при переключении нереализованного узла v узел становится реализованным, но некоторые из ранее реализованных соседей могут стать нереализованными, что приведет к общему снижению количества реализованных узлов. В действительности это происходит в одной из итераций на рис. 12.4: когда средний узел изменяет состояние, оба его (ранее реализованных) нижних соседа становятся нереализованными.

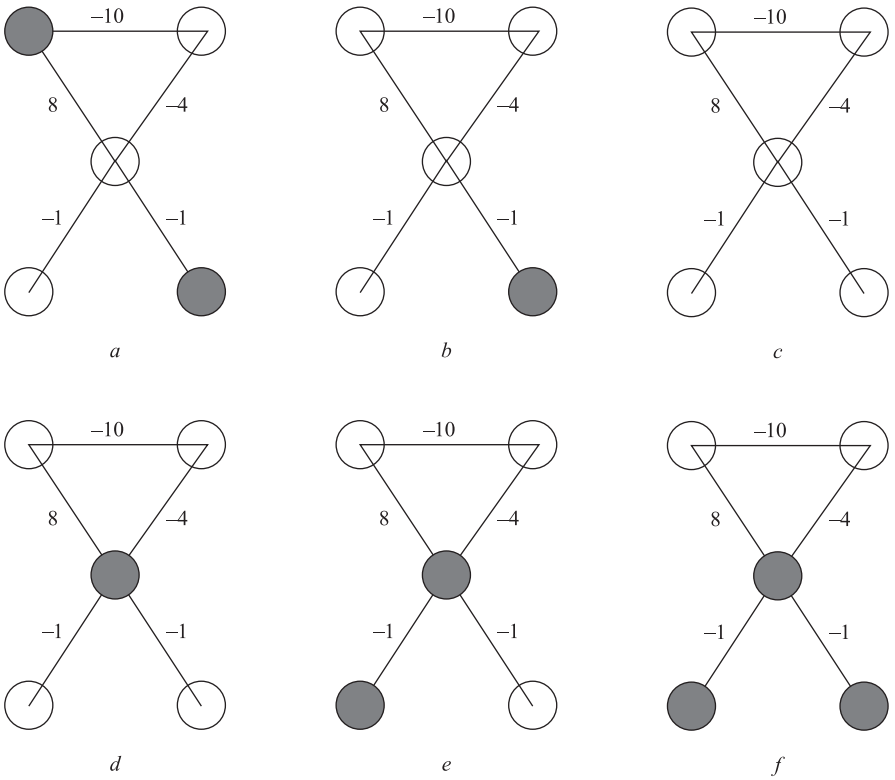


Рис. 12.4. Последовательность выполнения алгоритма переключения состояния для сети Хопфилда из пяти узлов, приводящая к устойчивой конфигурации. (Состояние узлов обозначается черным или белым цветом)

Также факт завершения невозможно доказать тем аргументом, что каждый узел изменяет состояние не более одного раза в ходе выполнения алгоритма: снова обратившись к примеру на рис. 12.4, мы видим, что узел в правом нижнем углу изменяет состояние дважды. (А также существуют более сложные примеры, в которых один узел может изменять состояние многократно.)

Однако существует менее очевидная метрика прогресса, которая изменяется с каждым изменением состояния нереализованного узла. А именно: для заданной конфигурации S мы определяем $\Phi(S)$ как общий абсолютный вес всех хороших ребер в сети, то есть

$$\Phi(S) = \sum_{\text{good } e} |w_e|.$$

Очевидно, для любой конфигурации S выполняется $\Phi(S) \geq 0$ (так как $\Phi(S)$ — сумма положительных целых чисел) и $\Phi(S) \leq W = \sum_e |w_e|$ (так как в крайнем случае каждое ребро является хорошим).

Теперь предположим, что в неустойчивой конфигурации S мы выбираем узел u, который является нереализованным, и изменяем его состояние, получая конфигурацию S'. Что можно сказать об отношениях между $\Phi(S')$ и $\Phi(S)$? Вспомним, что при переключении состояния u все хорошие ребра, инцидентные u, становятся плохими; все плохие ребра, инцидентные u, становятся хорошими; а все ребра, для которых u не является конечной точкой, остаются без изменений. Если обозначить g_u и b_u общий абсолютный вес соответственно хороших и плохих ребер, инцидентных u, то имеем

$$\Phi(S') = \Phi(S) - g_u + b_u.$$

Но так как узел u был нереализованным в S, мы также знаем, что $b_u > g_u$; а поскольку и b_u, и g_u являются целыми числами, имеем $b_u \geq g_u + 1$. Следовательно,

$$\Phi(S') \geq \Phi(S) + 1.$$

Значение Φ начинается с некоторого неотрицательного целого числа, увеличивается на 1 с каждым изменением состояния и не может превысить W. Следовательно, процесс выполняется не более W итераций, и при его завершении мы должны иметь устойчивую конфигурацию. Кроме того, при каждой итерации нереализованный узел выявляется с использованием полиномиального по n количества арифметических итераций; из этого также следует, что граница времени выполнения полиномиальна по n и W.

Итак, сутью доказательства существования устойчивых конфигураций в конечном итоге оказывается локальный поиск. Сначала мы определили целевую функцию Φ, которую требуется максимизировать. Конфигурации были возможными решениями этой задачи максимизации, и мы определили, какие две конфигурации S и S' должны считаться соседними: S' должна получаться из S переключением одного состояния.

Затем мы изучили поведение простого итеративного алгоритма локального поиска («перевернутую» форму градиентного спуска, так как в данном случае используется задача максимизации); при этом было обнаружено следующее:

(12.4) Любой локальный максимум в алгоритме переключения состояния, максимизирующий Φ, является устойчивой конфигурацией.

Следует заметить, что хотя наш алгоритм доказывает существование устойчивой конфигурации, время выполнения оставляет желать лучшего при больших абсолютных весах. А именно: по аналогии с тем, что мы видели в задаче о сумме подмножеств и в первом алгоритме максимального потока, полученный здесь алгоритм полиномиален только по фактической величине весов, а не по размеру их двоичных представлений. Для очень больших весов время выполнения может стать неприемлемым.

В настоящее время простые обходные решения неизвестны. Вопрос о существовании алгоритма, строящего устойчивые состояния за время, полиномиальное по n и $\log W$ (вместо n и W), или с количеством примитивных арифметических операций, полиномиальным только по n (независимо от W), остается открытым.

12.4. Аппроксимация задачи о максимальном разрезе с применением локального поиска

Теперь обсудим ситуацию, в которой алгоритм локального поиска применятся для нахождения доказуемой гарантии аппроксимации для задачи оптимизации. Для этого мы проанализируем структуру локальных оптимумов и установим границу качества локально оптимальных решений относительно глобального оптимума. Рассматриваемая задача о максимальном разрезе тесно связана с задачей поиска устойчивых конфигураций для сетей Хопфилда, которые встречались нам в предыдущем разделе.

Задача

В задаче о максимальном разрезе имеется ненаправленный граф $G = (V, E)$ с положительными целочисленными весами каждого ребра e. Для разбиения (A, B) множества вершин $w(A, B)$ обозначает общий вес ребер, один конец которых принадлежит A, а другой принадлежит B:

$$w(A, B) = \sum_{\substack{e=(u,v) \\ u \in A, v \in B}} w_e.$$

Целью является поиск разбиения (A, B) вершинного покрытия, максимизирующего $w(A, B)$. Задача о максимальном разрезе является *NP*-сложной в том смысле, что для взвешенного графа G и границы β принятие решения о том, существует ли разбиение (A, B) вершин G с $w(A, B) \geq \beta$, выполняется с *NP*-сложностью. В то же время, конечно, задача о максимальном разрезе напоминает задачу о минимальном разрезе *s–t* для потоковых сетей, имеющую полиномиальное решение. Основная причина ее неразрешимости обусловлена тем фактом, что мы стремимся максимизировать, а не минимизировать вес проходящих через разрез ребер.

И хотя задача нахождения устойчивой конфигурации сети Хопфилда не была оптимизационной задачей как таковой, мы видим, что задача о максимальном разрезе с ней тесно связана. На языке сетей Хопфилда задача о максимальном разрезе представляет собой экземпляр, в котором все веса ребер положительны (а не отрицательны), а конфигурации состояний узлов S естественно соответствуют разбиениям (A, B): узлы имеют состояние -1 в том, и только в том случае, если они принадлежат множеству A, и состояние $+1$ — в том, и только в том случае, если они принадлежат множеству B. Целью является такое распределение состояний, при котором как можно большая часть веса приходится на хорошие ребра — те, у которых конечные точки находятся в разных состояниях. В такой формулировке задача о максимальном разрезе направлена на максимизацию величины $\Phi(S)$, которая использовалась в доказательстве (12.3) в том случае, когда все веса ребер положительны.

Разработка алгоритма

Алгоритм переключения состояния, использованный для сетей Хопфилда, предоставляет алгоритм локального поиска для аппроксимации целевой функции максимального разреза $\Phi(S) = w(A, B)$. В контексте разбиений он говорит следующее: если существует узел u, для которого суммарный вес ребер из u в узлы на соответствующей стороне разбиения превышает общий вес ребер из u в узлы на другой стороне разбиения, то узел u следует переместить на другую сторону разбиения.

Введем понятие «ближнего соседства»: разбиения (A, B) и (A', B') называются *ближними соседними решениями*, если (A', B') можно получить из (A, B) перемещением одного узла с одной стороны разбиения на другую. Естественно задать два основных вопроса:

- Можно ли сказать что-то конкретное о качестве локальных оптимумов в окружении ближнего соседства?
- Так как ближнее соседство определяется предельно просто, какие варианты соседства могут предоставить более сильные алгоритмы локального поиска для максимального разреза?

Сначала мы ответим на первый из этих вопросов, а вторым займемся в следующем разделе.

Анализ алгоритма

Следующий результат отвечает на первый вопрос, показывая, что локальные оптимумы в ближнем соседстве предоставляют решения, соответствующие гарантированной границе аппроксимации.

(12.5) Пусть (A, B) — разбиение, которое является локальным оптимумом для задачи о максимальном разделе в ближайшем соседстве, а (A^*, B^*) — глобально-оптимальное разбиение. Тогда $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.

Доказательство. Пусть $W = \sum_e w_e$. Мы также немного расширим систему обозначений: для двух узлов u и v запись w_{uv} будет обозначать w_e, если существует ребро e, соединяющее u и v, и 0 в противном случае.

Для любого узла $u \in A$ должно выполняться условие

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv},$$

так как в противном случае узел u должен быть перемещен на другую сторону разбиения, и разбиение (A, B) не будет локально оптимальным. Просуммируем эти неравенства для всех $u \in A$; любое ребро, оба конца которого принадлежат A, будет находиться в левой части ровно двух таких неравенств, тогда как любое ребро, один конец которого принадлежит A, а другой принадлежит B, будет находиться в правой части ровно одного из таких неравенств. Следовательно, мы получаем

$$2\sum_{\{u,v\}\subseteq A} w_{uv} \le \sum_{u\in A, v\in B} w_{uv} = w(A,B). \tag{12.1}$$

Аналогичные рассуждения можно применить к множеству B, получая

$$2\sum_{\{u,v\}\subseteq B} w_{uv} \le \sum_{u\in A, v\in B} w_{uv} = w(A,B). \tag{12.2}$$

Сложив неравенства (12.1) и (12.2) и разделив результат на 2, получаем:

$$\sum_{\{u,v\}\subseteq A} w_{uv} + \sum_{\{u,v\}\subseteq B} w_{uv} \le w(A,B). \tag{12.3}$$

В левой стороне неравенства (12.3) учитываются веса всех ребер, не переходящих из A в B; следовательно, если прибавить $w(A, B)$ к обеим сторонам (12.3), левая сторона станет равна W. Правая сторона превращается в $2w(A, B)$, поэтому имеем $W \le 2w(A, B)$, или $w(A, B) \ge \frac{1}{2}W$. ■

Обратите внимание: в доказательстве (12.5) мы особенно не задерживались на оптимальном разбиении (A^*, B^*); в действительности было доказано более сильное утверждение о том, что в любом локально оптимальном решении в ближнем соседстве по крайней мере половина общего веса ребер в графе пересекает разбиение.

Утверждение (12.5) доказывает, что локальный оптимум представляет собой 2-аппроксимацию максимального разреза. Это наводит на мысль, что локальная оптимизация может быть хорошим алгоритмом для приближенной максимизации значения разреза. Тем не менее нужно рассмотреть еще одно обстоятельство: время выполнения. Как было показано в конце раздела 12.3, алгоритм переключения состояния только псевдополиномиальный, и вопрос о том, возможно ли найти локальный оптимум за полиномиальное время, остается открытым. Однако в данном случае мы можем добиться практически такого же результата, просто остановив алгоритм при отсутствии «достаточно значительных» улучшений.

Пусть (A, B) — разбиение с весом $w(A, B)$. Для фиксированного $\epsilon > 0$ переключение одного узла будет называться *значительным*, если оно улучшает значение разреза минимум на $\frac{2\varepsilon}{n}w(A,B)$, где $n = |V|$. Теперь рассмотрим версию алгоритма переключения состояния, которая принимает только значительные переключения и завершается при отсутствии таких переключений, даже если текущее разбиение не является локальным оптимумом. Утверждается, что эта версия алгоритма приведет почти к не худшей аппроксимации и выполняется за полиномиальное время. Прежде всего можно расширить предыдущее доказательство и продемонстрировать, что полученный разрез почти не хуже. Достаточно добавить $\frac{2\varepsilon}{n}w(A,B)$ к каждому неравенству, так как нам известно лишь об отсутствии значительных переключений.

(12.6) Пусть (A, B) — разбиение, для которого невозможно значительное переключение, а (A^*, B^*) — глобально-оптимальное разбиение. В этом случае $(2 + \epsilon)$ $w(A, B) \geq w(A^*, B^*)$.

Теперь обратимся ко времени выполнения.

(12.7) Версия алгоритма переключения состояния, принимающая только значительные переключения, завершается максимум после $O(\epsilon^{-1} n \log W)$ переключений (в предположении, что веса целочисленны, а $W = \sum_e w_e$.

Доказательство. Каждое переключение улучшает целевую функцию минимум на множитель $(1 + \epsilon/n)$. Так как $(1 + 1/x)^x \geq 2$ для любого $x \geq 1$, мы видим, что $(1 + \epsilon/n)^n/\epsilon \geq 2$, а следовательно, целевая функция возрастает не менее чем вдвое каждые n/ϵ переключений. Вес не может превысить W, а следовательно, он может удваиваться не более $\log W$ раз. ■

12.5. Выбор соседского отношения

В начале главы говорилось о том, что алгоритмы локального поиска в действительности базируются на двух основных ингредиентах: выборе соседского отношения и правиле выбора соседнего решения на каждом шаге. В разделе 12.2 мы занимались вторым ингредиентом: и алгоритм Метрополиса, и имитация отжига получают готовое соседское отношение и изменяют способ выбора соседнего решения.

Какие аспекты следует учитывать при выборе соседского отношения? Этот выбор может быть весьма непростым, хотя на высоком уровне альтернатива описывается достаточно тривиально.

(i) Соседское окружение решения должно быть достаточно широким, чтобы алгоритм не застревал в плохих локальных оптимумах.

(ii) Соседское окружение решения не должно быть слишком большим, потому что мы хотим иметь возможность эффективно искать в множестве соседей возможные локальные перемещения.

Если бы первый пункт был единственной проблемой, то все решения можно было бы просто сделать соседями друг друга — тогда локальных оптимумов вообще не будет, а глобальный оптимум всегда будет находиться в одном шаге! Второй пункт выявляет (очевидный) недостаток такого подхода: если бы соседское окружение текущего решения состояло из всех возможных решений, то парадигма локального поиска вообще не приносила никакой пользы и сводилась бы к простому перебору соседского окружения методом «грубой силы».

Вообще говоря, мы уже встречали один случай, в котором выбор правильного соседского отношения оказывал огромное влияние на разрешимость задачи, хотя этот факт тогда явно не отмечался: речь идет о задаче о двудольном паросочетании. Вероятно, простейшее соседское отношение для паросочетаний выглядит так: M' является соседом M, если M' может быть получено вставкой или удалением одного ребра в M. В соответствии с этим определением мы получаем «поверхности»

с множеством зубцов, как и в примерах вершинного покрытия, которые приводились выше; и по этому определению можно получить локально оптимальные паросочетания, имеющие только половину размера максимального сочетания.

Но предположим, мы попытаемся определить более сложное (и даже асимметричное) соседское отношение: M' является соседом M, если при создании соответствующей потоковой сети M' может быть получено из M одним увеличивающим путем. Что можно сказать о паросочетании M, если оно является локальным максимумом с этим соседским отношением? В этом случае увеличивающего пути не существует, поэтому M в действительности должно быть (глобальным) максимальным паросочетанием. Другими словами, с таким соседским отношением единственными локальными максимумами являются глобальные максимумы, так что прямой градиентный подъем приведет к максимальному паросочетанию. Если поразмыслить над тем, что делает алгоритм Форда–Фалкерсона в нашем сведении от двудольного паросочетания к максимальному потоку, это выглядит логично: размер паросочетания строго возрастает на каждом шаге, и нам никогда не приходится «отступать» из локального максимума. Следовательно, тщательно выбирая соседское отношение, мы превратили зазубренную поверхность оптимизации в простую воронку.

Конечно, не стоит ожидать, что всегда все будет так хорошо складываться. Например, так как задача о вершинном покрытии является *NP*-полной, было бы странно, если бы она допускала соседские отношения, которые одновременно порождают «удобные» поверхности, и соседские окружения с возможностью эффективного поиска. Рассмотрим несколько возможных соседских отношений в контексте задачи о максимальном разрезе из предыдущего раздела. Контрасты между этими соседскими отношениями характерны для проблем, возникающих в общей теме алгоритмов локального поиска для вычислительно сложных задач разбиения графов.

Алгоритмы локального поиска при разбиении графов

В разделе 12.4 был рассмотрен алгоритм переключения состояния для задачи о максимальном разрезе; было показано, что локально оптимальные решения обеспечивают 2-аппроксимацию. Перейдем к соседским отношениям, порождающим соседские окружения большего размера по сравнению с правилом одного переключения и соответственно пытающимся сократить распространенность локальных оптимумов. Возможно, самым естественным обобщением является соседское окружение с k-переключением для $k \geq 1$: разбиения (A, B) и (A', B') называются соседними по правилу k-переключения, если (A', B') можно получить из (A, B) перемещением не более k узлов с одной стороны разбиения на другую.

Очевидно, если (A, B) и (A', B') являются соседями по правилу k-переключения, то они также являются соседями по правилу k'-переключения для всех $k' > k$. Следовательно, если разбиение (A, B) является локальным оптимумом по правилу k'-переключения, то оно также является локальным оптимумом по правилу

k-переключения для всех $k < k'$. Но сокращение множества локальных оптимумов посредством повышения величины k обходится дорого: для просмотра множества соседей (A, B) по правилу k-переключения необходимо рассмотреть все $\Theta(n^k)$ способов перемещения до k узлов на другую сторону разбиения. Затраты времени становятся неприемлемыми даже при небольших значениях k.

Керниган и Лин (1970) предложили альтернативный способ генерирования соседних решений; он обладает существенно большей вычислительной эффективностью, но все при этом позволяет выполнять крупномасштабные преобразования решений за один шаг. Их метод, который мы будем называть *эвристикой К-Л*, определяет соседей разбиения (A, B) по следующей n-фазной процедуре.

- В фазе 1 выбирается один узел для переключения — так, чтобы значение полученного решения было как можно больше. Переключение выполняется даже в том случае, если значение решения убывает относительно $w(A, B)$. Узел, изменивший состояние, *помечается*, а полученное решение обозначается (A_1, B_1).
- В начале фазы k для $k > 1$ мы имеем разбиение (A_{k-1}, B_{k-1}), и $k - 1$ узлов помечены. Один непомеченный узел выбирается для переключения таким образом, что значение полученного решения является максимальным среди всех возможных. (И снова это происходит даже в том случае, если в результате значение решения уменьшится.) Переключенный узел помечается, а полученное решение обозначается (A_k, B_k).
- После n фаз каждый узел окажется помеченным; это указывает на то, что он переключался ровно один раз. Соответственно последнее разбиение (A_n, B_n) в действительности является зеркальным отображением исходного разбиения (A, B): $A_n = B$ и $B_n = A$. Наконец, эвристика К-Л определяет $n - 1$ разбиений (A_1, B_1), ..., (A_{n-1}, B_{n-1}) как соседей (A, B). Следовательно, (A, B) является локальным оптимумом по эвристике К-Л в том, и только в том случае, если $w(A, B) \geq w(A_i, B_i)$ для $1 \leq i \leq n - 1$. Итак, мы видим, что эвристика К-Л определяет очень длинную серию переключений, даже если ситуация на первый взгляд ухудшается, в надежде на то, что некоторое разбиение (A_i, B_i), сгенерированное по пути, окажется лучше (A, B). Но даже при том, что она генерирует соседей, очень отличных от (A, B), выполняется только n переключений и на каждое тратится время всего $O(n)$. С вычислительной точки зрения это намного разумнее правила k-переключения для больших значений k. Более того, эвристика К-Л на практике показала отличные результаты несмотря на то, что формальный анализ ее свойств в значительной степени остается открытой проблемой.

12.6.* Классификация на базе локального поиска

В этом разделе рассматривается более сложный пример применения локального поиска при разработке аппроксимирующих алгоритмов, связанный с задачей о сегментации изображения, которая рассматривалась среди практических применений

сетевых потоков в разделе 7.10. Более сложная версия сегментации изображения, которая будет рассматриваться здесь, дает пример того, как для получения хорошего быстродействия алгоритма локального поиска приходится использовать довольно сложную структуру соседского окружения для множества решений. Как вы увидите, естественное соседское окружение «переключения состояния», описанное в предыдущем разделе, может приводить к очень плохим локальным оптимумам. Для получения хорошего быстродействия мы воспользуемся экспоненциально большим соседским окружением. Одна из проблем больших соседских окружений заключается в том, что последовательный поиск по всем соседям текущего решения для его улучшения становится неприемлемым. Потребуется более сложный алгоритм поиска улучшенного соседа (если он существует).

Задача

Вспомним базовую задачу сегментации изображения, приведенную как пример использования потоков в разделе 7.10. Тогда задача сегментации изображения была сформулирована как задача *разметки*; наша цель заключалась в пометке (то есть классификации) каждого пиксела как принадлежащего к переднему плану или фону изображения. На тот момент было понятно, что это очень простая формулировка задачи и было бы неплохо иметь заняться более сложными задачами разметки, например сегментацией областей изображения в зависимости от их расстояния от камеры. По этой причине мы рассмотрим задачу разметки с более чем двумя метками. Попутно мы создадим инфраструктуру классификации, возможности применения которой не ограничиваются пикселами изображения.

При определении задачи сегментации с двумя метками «передний план/фон» мы в конечном итоге пришли к следующей формулировке. Имеется граф $G = (V, E)$, в котором V соответствует пикселам изображения, а цель заключается в классификации каждого узла V по двум возможным классам: фону и переднему плану. Ребра представляют пары узлов, которые с большой вероятностью принадлежат одному классу (например, потому что они расположены вблизи друг от друга), и для каждого ребра (i, j) задан штраф $p_{ij} \geq 0$ за размещение i и j в разных классах. Кроме того, имеется информация о вероятности того, принадлежит узел или пиксел переднему плану или фону. Эти вероятности были преобразованы в штрафы за отнесение узла к классу, которому он принадлежит с меньшей вероятностью. Задача заключалась в нахождении разметки узлов, минимизировавшей общие штрафы за разделение и отнесение к классу. Мы показали, что задача минимизации решается посредством вычисления минимального разреза. В оставшейся части раздела эта формулировка задачи будет называться *бинарной сегментацией изображения*.

Сейчас будет сформулирована аналогичная задача классификации/разметки с более чем двумя классами (метками). Как выясняется, эта задача является *NP*-сложной, и мы разработаем алгоритм локального поиска, в котором локальные оптимумы являются 2-аппроксимациями для лучшей разметки. Обобщенная задача разметки, рассматриваемая в этом разделе, формулируется следующим об-

разом. Имеются граф $G = (V, E)$ и множество L из k меток. Целью является пометка каждого узла в V одной из меток из множества L с целью минимизации некоторого штрафа. На выбор оптимальной разметки влияют два конкурирующих фактора. Для каждого ребра $(i, j) \in E$ действует *штраф за разделение* $p_{ij} \geq 0$ за пометку двух узлов i и j разными метками. Кроме того, узлы с большей вероятностью имеют одни метки вместо других. Это условие выражается штрафом за назначение. Для каждого узла $i \in V$ и каждой метки $a \in L$ действует неотрицательный штраф $c_i(a) \geq 0$ за назначение метки a узлу i. (Эти штрафы являются аналогами вероятностей из задачи бинарной сегментации, не считая того, что здесь они рассматриваются как минимизируемые затраты.) *Задача разметки* заключается в нахождении разметки f: $V \to L$, минимизирующей общий штраф:

$$\Phi(f) = \sum_{i \in V} c_i\left(f(i)\right) + \sum_{(i,j) \in E: f(i) \neq f(j)} p_{ij}.$$

Заметим, что задача разметки только с двумя метками в точности совпадает с задачей сегментации изображений из раздела 7.10. Для трех меток задача разметки уже является *NP*-сложной, хотя этот факт мы доказывать не будем.

Наша цель — разработать алгоритм локального поиска для этой задачи, в котором локальные оптимумы являются хорошими аппроксимациями оптимального решения. Пример также наглядно покажет, как важно выбирать хорошее соседское окружение для определения алгоритма локального поиска. Существует много возможных вариантов для соседских отношений, и вы увидите, что некоторые из них работают намного лучше других. В частности, для получения гарантий аппроксимации будет использовано довольно сложное определение соседского окружения.

Разработка алгоритма

Первая попытка: правило одного переключения

Простейшим и, наверное, самым естественным выбором соседского отношения является правило одного переключения из алгоритма переключения состояния для задачи о максимальном разрезе: две разметки являются соседними, если одну из них можно получить из другой изменением метки одного узла. К сожалению, это соседство может привести к достаточно слабым локальным оптимумам в нашей задаче, даже всего с двумя метками.

Это выглядит несколько странно, потому что правило хорошо работало в задаче о максимальном разрезе. Однако наша задача связана с задачей о минимальном разрезе. Собственно, задача о минимальном разрезе s–t соответствует частному случаю, в котором используются всего две метки, а s и t — единственные узлы со штрафами назначения. Нетрудно увидеть, что этот алгоритм переключения состояния не является хорошим аппроксимирующим алгоритмом для задачи о минимальном разрезе. Из рис. 12.5 видно, как ребра, инцидентные s, могут образовать глобальный оптимум, тогда как ребра, инцидентные t, могут образовать намного худший локальный оптимум.

Вторая попытка: рассмотрение двух меток за раз

Здесь мы разработаем алгоритм локального поиска, в котором окружение определяется намного сложнее. Интересная особенность этого алгоритма заключается в том, что он позволяет каждому решению иметь экспоненциальное количество соседей. На первый взгляд это противоречит общему правилу «соседское окружение решения не должно быть слишком большим», как утверждалось в разделе 12.5. Однако здесь мы будем работать с окружением более элегантно. Сохранение малого размера соседского окружения хорошо работает тогда, когда вы планируете проводить поиск улучшающего локального шага методом «грубой силы»; а на этот раз мы воспользуемся вычислением минимального разреза с полиномиальным временем, чтобы определить, встречается ли улучшение среди экспоненциально многочисленных соседей решения.

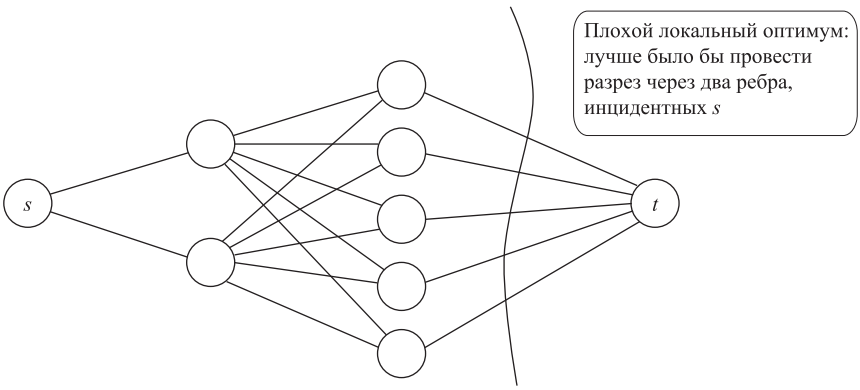


Рис. 12.5. Экземпляр задачи о минимальном разрезе s–t, в котором пропускные способности всех ребер равны 1

Идея локального поиска заключается в применении алгоритма с полиномиальным временем для бинарной сегментации изображения с целью улучшения локальных шагов. Начнем с простейшей реализации этой идеи, которая не всегда дает хорошую гарантию аппроксимации. Для разметки f выберем две метки $a, b \in L$ и ограничим внимание узлами, имеющими метки a или b в разметке f. За один локальный шаг любое подмножество этих узлов может переключить метки из a в b или из b в a. Или в более формальном определении, две разметки f и f' считаются соседними, если существуют две метки $a, b \in L$, такие, что для всех остальных меток $c \notin \{a, b\}$ и всех узлов $i \in V$ условие $f(i) = c$ выполняется в том, и только в том случае, если $f'(i) = c$. Следует заметить, что состояние f может иметь сколь угодно много соседей, так как произвольное подмножество узлов с метками a и b может переключать свою метку. Однако при этом действует следующее утверждение:

(12.8) Если разметка f не является локально оптимальной для описанного выше соседского окружения, то сосед с меньшим штрафом может быть найден за k^2 вычислений минимального разреза.

Доказательство. Количество пар разных меток меньше k^2, поэтому каждую пару можно проверить по отдельности. Для пары меток $a, b \in L$ рассмотрим задачу

нахождения улучшенной разметки посредством перестановки меток узлов между *a* и *b*. Она в точности совпадает с задачей сегментации для двух меток в подграфе узлов, которым в *f* назначены метки *a* или *b*. Воспользуемся алгоритмом, разработанным для бинарной сегментации изображений, для поиска лучшей из таких измененных разметок. ■

Это соседское окружение намного лучше варианта с одним переключением, который был рассмотрен в начале. Например, оно предоставляет оптимальное решение случая с двумя метками. Однако даже с улучшенным окружением локальные оптимумы могут быть плохими, как показано на рис. 12.6. В этом примере есть три узла *s*, *t* и *z*, которые должны сохранить свои исходные метки. Все остальные узлы лежат на одной из сторон треугольника; они должны сохранить одну из двух меток, связанных с узлами на концах этой стороны. Эти требования легко выражаются назначением каждому узлу очень большого штрафа за назначение для неразрешенных меток. Штрафы за разделение определяются следующим образом: тонким ребрам на рисунке соответствует штраф 1, а толстым — большой штраф *M*. Теперь заметим, что разметка на иллюстрации имеет штраф $M + 3$, но при этом является локально оптимальной. У (глобально) оптимальной разметки штраф равен всего 3, а для ее получения на рисунке достаточно изменить метки обоих узлов рядом с *s*.

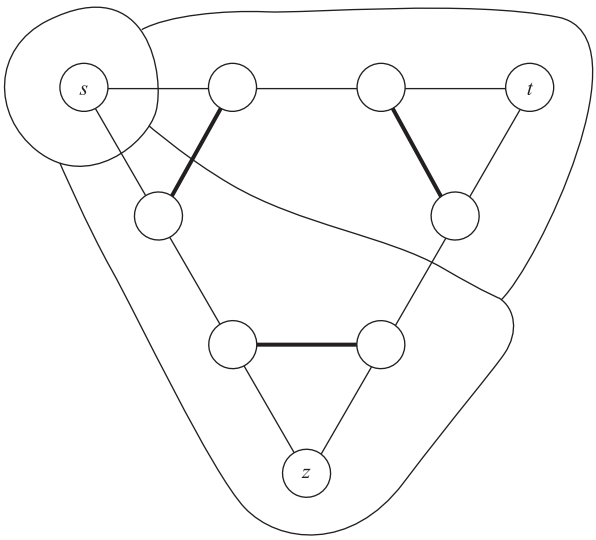


Рис. 12.6. Плохой локальный оптимум для алгоритма локального поиска, учитывающего только две метки

Работоспособное соседское окружение при локальном поиске

Теперь мы определим другое соседское окружение, приводящее к хорошему алгоритму аппроксимации. Локальный оптимум на рис. 12.6 наводит на мысль о том,

что могло бы быть хорошим соседским окружением: нужно иметь возможность переназначения меток узлов за один шаг. Для этого необходимо найти соседское отношение, которое было бы достаточно широким для выполнения этого свойства, но при этом позволяло найти улучшающий локальный шаг за полиномиальное время.

Рассмотрим разметку f. В процессе локального шага нашего нового алгоритма должно происходить следующее: мы выбираем одну метку $a \in L$ и ограничиваем рассмотрение узлами, *не имеющими* метки a в разметке f. За один локальный шаг любому подмножеству этих узлов разрешается изменить свои метки на a. Или в более формальном определении, две разметки f и f' считаются соседними, если существует метка $a \in L$, такое что для всех узлов $i \in V$ либо $f'(i) = f(i)$, либо $f'(i) = a$. Обратите внимание на то, что это соседское отношение не является симметричным; другими словами, f не удастся получить обратно из f' за один шаг. Теперь мы покажем, что для любой разметки f любого соседа можно найти за k вычислений минимального разреза, а локальный оптимум этого соседского окружения является 2-аппроксимацией разметки с минимальным штрафом.

Поиск хорошего соседа

Чтобы найти лучшего соседа, мы проверяем каждую метку по отдельности. Начнем с метки a. Утверждается, что лучшее переназначение, в котором узлы могут изменить свои метки на a, может быть найдено посредством вычисления минимального разреза. Построение графа минимального разреза $G' = (V', E')$ аналогично вычислению минимального разреза, разработанному для бинарной сегментации изображений. Тогда мы определили источник s и сток t для представления двух меток. Здесь мы тоже введем источник и сток, но источник s будет представлять метку a, а сток t будет фактически представлять другой вариант, имеющийся у узлов, а именно сохранение их исходных меток. Идея заключается в том, чтобы найти минимальный разрез в G' и заменить метки всех узлов на стороне s разреза меткой a, тогда как все узлы на стороне t сохранят свои исходные метки.

Для каждого узла G в новом множестве V' имеется соответствующий узел, а в E' добавляются ребра (i, t) и (s, i), как это делалось на рис. 7.18 из главы 7 в случае двух меток. Ребро (i, t) имеет пропускную способность $c_i(a)$, так как разрезание ребра (i, t) приводит к размещению узла i на стороне источника, а следовательно, соответствует пометке узла i меткой a. Ребро (i, s) будет иметь пропускную способность $c_i(f(i))$, если $f(i) \neq a$, или она будет выражаться очень большим числом M (или $+\infty$), если $f(i) = a$. Разрезание ребра (i, t) помещает узел i на сторону стока, а следовательно, соответствует сохранению узлом i исходной метки $f(i) \neq a$. Большая пропускная способность M предотвращает размещение узлов i с $f(i) = a$ на стороне стока.

В построении для задачи с двумя метками мы добавляли ребра между узлами V и использовали штрафы за разделение как пропускные способности. Этот способ хорошо работает для узлов, разделенных разрезом, или узлов на стороне источника, одновременно имеющих метку a. Но если i и j находятся на стороне стока, то соединяющее их ребро еще не разрезано, но i и j разделены, если $f(i) \neq f(j)$. Чтобы ре-

шить эту проблему, мы дополним построение G' следующим образом. Для ребра (i, j), если $f(i) = f(j)$ или один из узлов i или j имеет метку a, в E' добавляется ребро (i, j) с пропускной способностью p_{ij}. Для ребер $e = (i, j)$, у которых $f(i) \neq f(j)$ и ни один узел не имеет метку a, чтобы правильно закодировать через граф G', что i и j остаются разделенными, даже если находятся на стороне стока, придется действовать иначе. Для каждого такого ребра e в V' добавляется дополнительный узел e, соответствующий ребру e, и ребра (i, e), (e, j) и (e, s) с пропускной способностью p_{ij}. Эти ребра изображены на рис. 12.7.

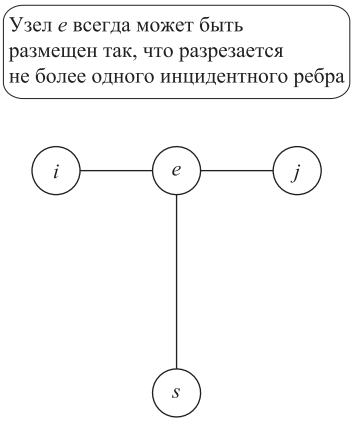


Рис. 12.7. Построение для ребра $e = (i, j)$ с $a \neq f(i) \neq f(j) \neq a$

(12.9) Для заданной разметки f и метки a минимальный разрез в графе $G' = (V', E')$ соответствует соседу разметки f с минимальным штрафом, полученному заменой меток подмножества узлов на a. В результате сосед f с минимальным штрафом может быть найден посредством k вычислений минимального разреза, по одному для каждой метки в L.

Доказательство. Пусть (A, B) — разрез s–t в G'. Большое значение M гарантирует, что разрез с минимальной пропускной способностью не разрежет никакие из этих ребер с высокой пропускной способностью. Теперь рассмотрим узел e в G', соответствующий ребру $e = (i, j) \in E$. Узел $e \in V'$ имеет три прилегающих ребра, каждое из которых имеет пропускную способность p_{ij}. Для любого разбиения остальных узлов e можно разместить так, что разрезано будет не более одного из этих трех ребер. Назовем разрез *хорошим*, если никакое ребро с пропускной способностью M не разрезается и для всех узлов, соответствующих ребрам в E, разрезается не более одного из прилегающих ребер. К настоящему моменту мы обосновали, что все разрезы с минимальной пропускной способностью являются хорошими.

Хорошие разрезы s–t в G' находятся в однозначном соответствии с переназначениями меток f, полученными заменой метки подмножества узлов на a. Рассмотрим пропускную способность хорошего разреза. Ребра (s, i) и (i, t) добавляют в пропускную способность разреза в точности штраф за назначение. Ребра (i, j), напрямую соединяющие узлы в V, вносят в точности штраф за разделение узлов

в соответствующей разметке: p_{ij}, если они разделены, и 0 в противном случае. Наконец, рассмотрим ребро $e = (i, j)$ с соответствующим узлом $e \in V'$. Если оба узла i и j находятся на стороне источника, ни одно из трех ребер, прилегающих к e, не будет разрезано, а во всех остальных случаях разрезается ровно одно из этих ребер. Итак, три ребра, прилегающи к e, добавляют к разрезу в точности величину штрафа за разделение между i и j в соответствующей разметке. В результате пропускная способность хорошего разреза в точности совпадает со штрафом соответствующей разметки, а следовательно, разрез с минимальной пропускной способностью соответствует лучшему переназначению меток f. ■

Анализ алгоритма

Наконец, необходимо рассмотреть качество локальных оптимумов при таком определении соседских отношений. Вспомните, что в двух предыдущих попытках определения соседского окружения выяснилось, что в обоих случаях мы приходим к плохим локальным оптимумам. Теперь мы покажем, что любой локальный оптимум с новым соседским отношением является 2-аппроксимацией минимального возможного штрафа.

Начнем с рассмотрения оптимальной разметки f^*; пусть для метки $a \in L$ $V^*_a = \{i : f^*(i) = a\}$ обозначает множество узлов с меткой a в f^*. Рассмотрим локально оптимальную разметку f. Чтобы получить соседа f_a разметки f, мы заменим метки всех узлов в V^*a на a. Разметка f является локально оптимальной, а следовательно, этот сосед f_a имеет не меньший штраф: $\Phi(f_a) \geq \Phi(f)$. Рассмотрим разность $\Phi(f_a) - \Phi(f)$, которая, как мы знаем, не отрицательна. Какие величины вносят свой вклад в эту разность? Единственное возможное изменение в штрафах за назначение может происходить от узлов в V^*_d; для каждого $i \in V^*_a$ изменение равно $c_i(f^*(i)) - c_i(f(i))$. Штрафы за разделение между двумя разметками различаются только в ребрах (i, j), по крайней мере один конец которых находится в V^*_d. Следующее неравенство учитывает эти различия.

(12.10) Для разметки f и ее соседа f_a

$$\Phi(f_a) - \Phi(f) \leq \sum_{i \in V^*_a} \left[c_i\left(f^*(i)\right) - c_i\left(f(i)\right) \right] + \sum_{(i,j) \text{leaving } V^*_a} p_{ij} - \sum_{\substack{(i,j) \text{in or leaving } V^*_a \\ f(i) \neq f(j)}} p_{ij}.$$

Доказательство. Изменение в штрафах за назначение равно $\sum_{i \in V^*_a} c_i\left(f^*(i)\right)$. Штраф за разделение для ребра (i, j) может различаться между двумя разметками только в том случае, если по крайней мере один конец ребра (i, j) принадлежит V^*_d. Общий штраф за разделение в разметке f для таких ребер равен в точности

$$\sum_{\substack{(i,j) \text{in or leaving } V^*_a \\ f(i) \neq f(j)}} p_{ij},$$

тогда как в разметке f_a штраф за разделение не превышает

$$\sum_{(i,j)\text{leaving } V_a^*} p_{ij}$$

для этих ребер. (Обратите внимание: последнее выражение всего лишь определяет верхнюю границу, так как ребро (i, j), выходящее из V^*_a другой конец которого находится в a, не участвует в штрафе за разделение f_a.) ■

Теперь можно переходить к доказательству основного утверждения.

(12.11) Для любой локально оптимальной разметки f и любой другой разметки $f^* - \Phi(f) \geq 2\Phi(f^*)$.

Доказательство. Пусть f_a — сосед f, определенный ранее переназначением меток узлов на a. Разметка f является локально оптимальной, поэтому имеем $\Phi(f_a) - \Phi(f) \geq 0$ для всех $a \in L$. Используем (12.10) для ограничения $\Phi(f_a) - \Phi(f)$, а затем просуммируем полученные неравенства для всех меток с получением следующего результата:

$$0 \leq \sum_{a \in L} \left(\Phi(f_a) - \Phi(f)\right)$$

$$\leq \sum_{a \in L} \left[\sum_{i \in V_a^*} c_i\left(f^*(i)\right) - c_i\left(f(i)\right) + \sum_{(i,j)\text{leaving } V_a^*} p_{ij} - \sum_{\substack{(i,j)\text{in or leaving } V_a^* \\ f(i) \neq f(j)}} p_{ij} \right].$$

Сгруппируем положительные слагаемые в левой части, а отрицательные — в правой. В левой части получаем $c_i(f^*(i))$ для всех узлов i, что в точности равно штрафу за назначение f^*. Кроме того, слагаемое p_{ij} учитывается дважды для каждого из ребер, разделенных f^* (по одному разу для каждой из двух меток $f^*(i)$ и $f^*(j)$).

В правой части получаем $c_i(f(i))$ для всех узлов i, что в точности равно штрафу за назначение f. Кроме того, учитываются слагаемые p_{ij} для ребер, разделенных f. Каждый такой штраф за разделение учитывается хотя бы один раз, а возможно и дважды, если разделение также встречается в f^*.

В итоге получаем следующее.

$$2\Phi\left(f^*\right) \geq \sum_{a \in L} \left[\sum_{i \in V_a^*} c_i\left(f^*(i)\right) + \sum_{(i,j)\text{leaving } V_a^*} p_{ij} \right],$$

$$\geq \sum_{a \in L} \left[\sum_{i \in V_a^*} c_i\left(f(i)\right) + \sum_{\substack{(i,j)\text{in or leaving } V_a^* \\ f(i) \neq f(j)}} p_{ij} \right] \geq \Phi(f).$$

Граница в утверждении доказана. ■

Мы доказали, что все локальные оптимумы являются хорошими аппроксимациями для разметок с минимальным штрафом. Осталось разобраться еще с одним вопросом: насколько быстро алгоритм находит локальный оптимум? Вспомните, что в случае задачи о максимальном разрезе нам пришлось прибегнуть

к разновидности алгоритма, которая принимала только большие улучшения, так как повторные локальные улучшения могли не обеспечить полиномиальное время выполнения. То же самое происходит и здесь: пусть $\epsilon > 0$ — константа. Для заданной разметки f мы будем считать соседнюю разметку f' *значительным улучшением*, если $\Phi(f') \leq (1 - \epsilon/3k)\,\Phi(f)$. Чтобы алгоритм выполнялся за полиномиальное время, следует принимать только значительные улучшения и завершать его выполнение, когда они невозможны. После максимум $\epsilon^{-1}k$ значительных улучшений штраф уменьшается с постоянным множителем; следовательно, алгоритм завершится за полиномиальное время. **Доказательство** (12.11) нетрудно адаптировать для того, чтобы доказать следующее.

(12.12) Для любого фиксированного $\epsilon > 0$ версия алгоритма локального поиска, принимающая только значительные улучшения, завершается за полиномиальное время и приводит к такой разметке f, что $\Phi(f) \leq (2 + \epsilon)\,\Phi(f^*)$ для любой другой разметки f^*.

12.7. Динамика наилучших ответов и равновесия Нэша

До настоящего момента мы рассматривали локальный поиск как метод решения оптимизационных задач только с единственной целью — другими словами, применяли локальные операции к потенциальному решению для минимизации его общей стоимости. Однако существует много других настроек, в которых решение некоторой задачи строится коллективным взаимодействием большого количества агентов, каждый из которых имеет собственные цели. Решение, полученное в таких обстоятельствах, часто отражает процесс «перетягивания каната», которое к нему привело, когда каждый агент пытается сместить решение в благоприятном для себя направлении. Вы увидите, что такие взаимодействия могут рассматриваться как разновидность процедуры локального поиска; аналогии локальных минимумов также имеют естественный смысл, но наличие множественных агентов и множественных целей создает новые проблемы.

Область теории игр предоставляет естественную основу для обсуждения того, что происходит в ситуациях стратегического взаимодействия множества агентов, то есть, когда каждый пытается оптимизировать свою целевую функцию. Чтобы продемонстрировать возникающие проблемы, мы рассмотрим практический пример из области сетевой маршрутизации; попутно будут представлены некоторые понятия, играющие важную роль в области более общей теории игр.

Задача

В такой сети, как Интернет, часто встречаются ситуации, в которых сразу несколько узлов пытаются установить связь с одним узлом-источником s. Например, источник s может генерировать некий поток данных, который желают получать

все узлы, — по аналогии со схемой сетевых взаимодействий «один ко многим», называемой *многоадресной передачей* (multicast). Описанная ситуация моделируется представлением сети в виде направленного графа $G = (V, E)$, со стоимостью $c_e \geq 0$ каждого ребра. Имеется выделенный узел-источник $s \in V$ и набор из k агентов, расположенных в разных *терминальных узлах* $t_1, t_2, ..., t_k \in V$. Для простоты мы не будем различать агентов и узлы, в которых они находятся; другими словами, агенты будут рассматриваться как $t_1, t_2, ..., t_k$.

Каждый агент t_j стремится построить путь P_j из s в t_j с минимальной общей стоимостью. При отсутствии взаимодействий между объектами задача распадается на k отдельных задач о кратчайшем пути: каждый агент t_j находит путь s–t_j, минимизирующий общую стоимость всех ребер, и использует его как свой путь P_j. В этой задаче интересна возможность «совместного использования» стоимости ребер агентами. Допустим, после того, как все агенты выберут свои пути, агенту t_j достаточно заплатить «свою долю» стоимости каждого ребра e на своем пути; иначе говоря, вместо того, чтобы платить c_e для каждого ребра e в P_i, он платит величину c_e, разделенную на количество агентов, пути которых содержат e. Таким образом у агентов появляется стимул для использования перекрывающихся путей, потому что они могут выиграть от разбивки стоимости ребер. (Эта модель подходит для конфигураций, в которых использование ребра несколькими агентами не приводит к значительному снижению качества передачи из-за перегрузки или увеличения задержки. Если эффекты задержки начинают действовать, вводится компенсирующий штраф за совместное использование; эта ситуация также приводит к интересным алгоритмическим вопросам, но мы пока будем придерживаться текущей формулировки, в которой совместное использование приносит только выгоду.)

Динамика наилучших ответов и равновесия Нэша: определения и примеры

Чтобы понять, как возможность совместного использования влияет на поведение агентов, для начала рассмотрим пару очень простых примеров на рис. 12.8. В примере *a* у каждого из двух агентов есть два варианта построения пути: средний маршрут через v и внешний маршрут с использованием одного ребра. Предположим, каждый агент начинает с исходного пути, но постоянно пересчитывает текущую ситуацию, чтобы решить, возможно ли перейти на лучший путь.

Допустим, в примере *a* два агента начинают с использования внешних путей. Затем t_1 не видит преимущества в переходе на другой путь (так как $4 < 5 + 1$), но у t_2 такое преимущество есть (так как $8 > 5 + 1$), поэтому t_2 обновляет свой путь с переходом на середину. Когда это происходит, с точки зрения t_1 ситуация изменяется: внезапно у t_1 также появляется выгода от перехода, так как он сможет оплатить лишь часть стоимости среднего пути, и для него стоимость пути сокращается до $2,5 + 1 < 4$. Соответственно t_1 переходит на новый путь. В ситуации, в которой обе стороны используют средний путь, ни у кого нет стимула для перехода, поэтому такое решение может считаться устойчивым.

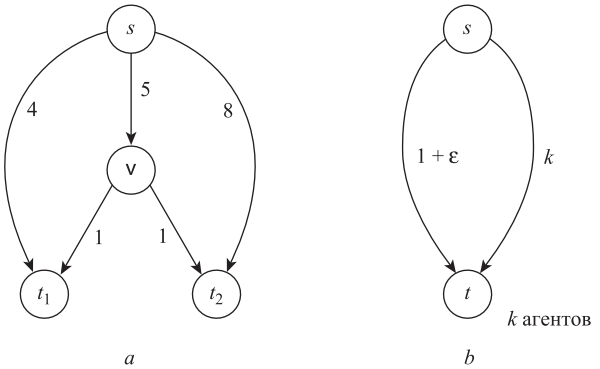


Рис. 12.8. Совместное использование среднего пути в интересах обоих агентов (a). Для всех агентов было бы лучше использовать левое ребро. Но если все k агентов начнут с правого ребра, ни один из них не захочет в одностороннем порядке перейти справа налево (b); другими словами, решение, в котором все агенты совместно используют ребро справа, является плохим равновесием Нэша

Обсудим два определения из области теории игр, в которых отражено происходящее в этом простом примере. Хотя мы по-прежнему будем ориентироваться на конкретную задачу маршрутизации при многоадресной передаче, эти определения подходят для любых конфигураций с несколькими агентами (каждый из которых обладает своей целью), взаимодействующими для выработки коллективного решения. Соответственно в определениях будут использованы эти общие термины.

- Прежде всего, в представленном примере каждый агент был всегда готов улучшить свое решение в ответ на изменения, внесенные другими агентами. Мы будем называть этот процесс *динамикой наилучших ответов*. Другими словами, нас интересует динамическое поведение процесса, в котором каждый агент обновляет свое состояние в зависимости от своего наилучшего ответа на текущую ситуацию.
- Кроме того, нас особенно интересуют *устойчивые решения*, в которых лучший ответ каждого агента должен оставаться неизменным. Такое решение, в котором ни у одного агента нет стимула для отклонения, называется *равновесием Нэша* (по имени математика Джона Нэша, получившего Нобелевскую премию по экономике за свою новаторскую работу по этой концепции). Следовательно, в примере *a* решение, в котором оба агента используют средний путь, является равновесием Нэша. Обратите внимание: равновесия Нэша точно соответствуют решениям, в которых завершается динамика наилучших ответов.

Пример на рис. 12.8, *b* демонстрирует возможность существования нескольких равновесий Нэша. В этом примере все k агентов находятся в общем узле t (то есть $t_1 = t_2 = ... = t_k = t$), и из s в t ведут два параллельных ребра с разной стоимостью. Решение, в котором все агенты используют левое ребро, представляет собой равновесие Нэша, где каждый агент платит $(1 + \varepsilon)/k$. Решение, в котором все агенты используют правое ребро, также является равновесием Нэша, хотя в этом случае каждый агент платит $k/k = 1$. Этот факт подчеркивает один важный аспект дина-

мики наилучших ответов: если агенты смогут как-то согласовать одновременный переход с правого ребра на левое, в целом они от этого выиграют. Однако в динамике наилучших ответов каждый агент оценивает только последствия от своего одностороннего перемещения. По сути, агент не может делать никакие предположения относительно будущих действий других агентов (в среде Интернета он может даже ничего не знать о существовании таких агентов или их текущих решениях), поэтому агент желает выполнять только те обновления, которые приводят к немедленному улучшению ситуации для него самого.

Для количественного выражения смысла, в котором одно из равновесий Нэша на рис. 12.8, *b* лучше другого, полезно ввести еще одно определение. Решение называется *социальным оптимумом*, если оно минимизирует общую стоимость для всех агентов. Такое решение могло бы быть принято центральным органом, с точки зрения которого все агенты имеют одинаковый приоритет, и поэтому качество общего решения оценивается простым суммированием их затрат. Следует заметить, что в *a* и *b* существует социальный оптимум, который также является равновесием Нэша, хотя в *b* также существует второе равновесие Нэша с много большей стоимостью.

Связь с локальным поиском

К этому моменту постепенно начинает проступать связь с локальным поиском. Множество агентов, следующих динамике наилучших ответов, участвует в своего рода процессе градиентного спуска, исследуя «поверхность» возможных решений в стремлении к минимизации отдельных стоимостей. Равновесия Нэша являются естественными аналогиями локальных минимумов в этом процессе: это решения, для которых невозможны улучшающие перемещения. «Локальная» природа поиска также очевидна, поскольку агенты обновляют свои решения только тогда, когда это приводит к немедленному улучшению.

Впрочем, даже с учетом сказанного стоит отметить ряд критических отличий от стандартного локального поиска. В начале главы мы могли легко обосновать, что алгоритм градиентного спуска для комбинаторной задачи должен завершиться в локальном минимуме: каждое обновление уменьшало стоимость решения, а поскольку количество возможных решений было конечным, серия обновлений не могла продолжаться бесконечно. Другими словами, сама функция стоимости предоставляла метрику прогресса, необходимую для обоснования завершения.

С другой стороны, в динамике наилучших ответов каждый агент имеет собственную целевую функцию, которую он пытается минимизировать, поэтому не так очевидно, какой общий «прогресс» происходит, например, при обновлении агентом t_i своего пути из s. Конечно, для t_i прогресс существует, поскольку его стоимость пути снижается, но снижение может быть скомпенсировано еще большим возрастанием стоимости у другого агента. Для примера рассмотрим сеть на рис. 12.9. Если оба агента начинают со среднего пути, то у t_1 будет стимул для перехода на внешний путь; его стоимость уменьшается с 3,5 до 3, но в процессе стоимость t_2 увеличивается с 3,5 до 6. (Когда это произойдет, t_2 также переместится

на внешний путь, и это решение — при котором оба узла используются внешними путями — является уникальным равновесием Нэша.)

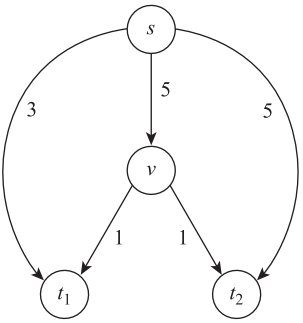


Рис. 12.9. Сеть, в которой уникальное равновесие Нэша отличается от социального оптимума

Впрочем, в некоторых ситуациях эффекты возрастания стоимости динамики наилучших ответов могут быть намного худшими. Взгляните на рис. 12.10: имеются k агентов, каждый из которых имеет возможность выбрать общий внешний путь со стоимостью $1 + \varepsilon$ (для некоторого малого числа $\varepsilon > 0$) или собственный альтернативный путь. Альтернативный путь для t_j имеет стоимость $1/j$. Теперь предположим, что мы начинаем с решения, в котором все агенты совместно используют внешний путь. Каждый агент платит $(1 + \varepsilon)/k$, и это решение минимизирует общую стоимость по всем агентам. Но с применением динамики наилучших ответов, начиная с этого решения, события начинают стремительно развиваться. Сначала t_k переключается на свой альтернативный путь, так как $1/k < (1+\varepsilon)/k$. В результате теперь внешний путь используется только $k - 1$ агентами, поэтому $t_k - 1$ переходит на альтернативный путь, так как $1/(k-1) < (1 + \varepsilon)/(k-1)$. После этого переходит $t_k - 2$, потом $t_k - 3$, и т. д., пока все k агентов не будут использовать альтернативные пути прямо из s. Здесь изменения временно прекращаются из-за следующего факта.

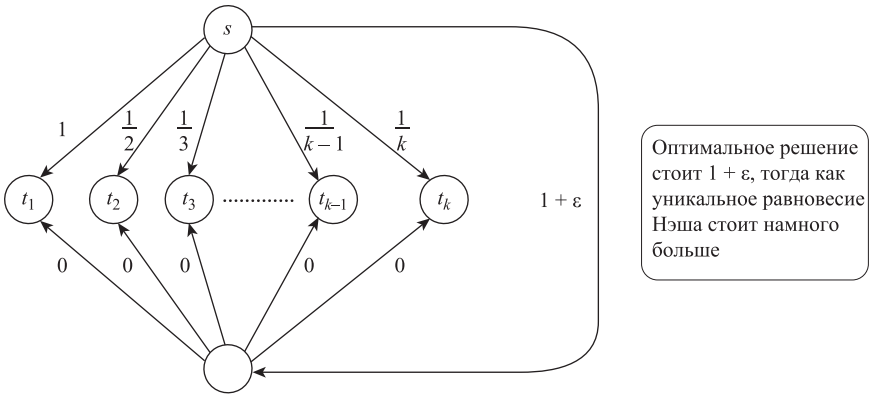


Рис. 12.10. Сеть, в которой уникальное равновесие Нэша стоит в $H(k) = \Theta(\log k)$ раз больше социального оптимума

(12.13) Решение на рис. 12.10, в котором каждый агент использует свой прямой путь от s, является равновесием Нэша; более того, в данном экземпляре оно является уникальным равновесием Нэша.

Доказательство. Чтобы убедиться в том, что данное решение является равновесием Нэша, достаточно проверить, что ни у одного агента нет причин для перехода с его текущего пути. Но это очевидно, поскольку все агенты платят не более 1, а единственный другой вариант — (в настоящее время свободный) внешний путь — имеет стоимость $1 + \varepsilon$.

А теперь предположим, что существует другое равновесие Нэша. Чтобы оно отличалось от рассматриваемого решения, в нем должен быть задействован по крайней мере один из агентов, использующих внешний путь. Пусть t_{j_1}, t_{j_2}, ..., t_{j_ℓ} — агенты, использующие внешний путь, где $j_1 < j_2 < \cdots < j_\ell$. Тогда все эти агенты платят $(1 + \epsilon)/\ell$. Но обратите внимание на то, что $j_\ell \geq \ell$, и у агента t_{j_ℓ} есть возможность заплатить только $1/j_\ell \leq 1/\ell$ за счет использования альтернативного пути прямо из s. Следовательно, у t_{j_ℓ} есть стимул для отклонения от текущего решения, а значит, решение не может быть равновесием Нэша. ■

Рисунок 12.8, *b* уже показал, что может существовать равновесие Нэша с общей стоимостью гораздо худшей, чем у социального оптимума, но примеры на рис. 12.9 и 12.10 наглядно раскрывают еще более важный момент: общая стоимость для всех агентов даже в *самом выгодном* решении с равновесием Нэша может быть хуже общей стоимости в социальном оптимуме. Насколько хуже? Общая стоимость социального оптимума в этом примере равна $1 + \varepsilon$, тогда как стоимость уникального равновесия Нэша составляет $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k} = \sum_{i=1}^{k} \frac{1}{i}$. Это выражение уже встречалось в главе 11, где мы определили его как *гармоническое число* $H(k)$ и показали, что его асимптотическое значение равно $H(k) = \Theta(\log k)$.

Эти примеры поясняют, что социальный оптимум не следует рассматривать как аналогию глобального минимума в традиционной процедуре локального поиска. При стандартном локальном поиске глобальный минимум всегда является устойчивым решением, так как никакие улучшения невозможны. Однако социальный оптимум может быть неустойчивым решением, так как для этого достаточно, чтобы всего один агент был заинтересован в отклонении.

Два основных вопроса

Динамика наилучших ответов может проявлять разнообразные варианты поведения, и мы уже видели несколько примеров, демонстрирующих разные аспекты. Будет полезно сделать шаг назад, оценить наш текущий уровень понимания и задать некоторые основные вопросы. Мы объединим их в две группы.

- **Существование равновесия Нэша**. На данный момент мы еще не располагаем доказательствами того, что решение с равновесием Нэша хотя бы *существует* в каждом экземпляре задачи многоадресной передачи. Самый естественный кандидат на метрику прогресса — общая стоимость всех агентов — не обязательно убывает при обновлении одним агентом его пути.

С учетом этого обстоятельства даже неясно, как обосновать, что динамика наилучших ответов должна завершиться. Почему не может возникнуть цикл, при котором агент t_1 улучшает свое решение за счет t_2, а затем t_2 улучшает свое решение за счет t_1, и т. д. до бесконечности? В самом деле, нетрудно определить другие задачи, в которых возникает именно такая ситуация и в которых равновесия Нэша не существует. Чтобы обосновать, что динамика наилучших ответов ведет к равновесию Нэша в данном случае, необходимо разобраться, что же такого особенного в задаче маршрутизации.

- **Цена устойчивости**. До настоящего момента равновесие Нэша рассматривалось в основном в роли «наблюдателей»: фактически мы «выпускаем» агентов на граф в произвольной начальной точке и смотрим, что они сделают. Но если рассматривать происходящее с точки зрения разработчиков протокола, пытаясь определить процедуру построения агентами путей из s, можно пойти по следующему пути: для заданного множества агентов, находящихся в узлах t_1, t_2, ..., t_k, можно предложить множество путей (по одному для каждого агента), обладающее двумя свойствами:

  (i) Множество путей образует решение с равновесием Нэша;

  (ii) С учетом (i) общая стоимость для всех агентов настолько мала, насколько это возможно.

  Конечно, в идеале нам хотелось бы добиться наименьшей общей стоимости, как при социальном оптимуме. Но если предложить социальный оптимум, который не является равновесием Нэша, он не будет устойчивым: агенты начнут отклоняться и строить новые пути. Таким образом, свойства (i) и (ii) совместно представляют попытку нашего протокола найти оптимум с учетом устойчивости и отыскать лучшее решение, от которого ни один агент не захочет отойти.

  По этой причине для заданного экземпляра задачи определяется *цена устойчивости* как отношение стоимости лучшего решения с равновесием Нэша к стоимости социального оптимума. Эта характеристика отражает рост стоимости, обусловленный требованием о том, что наше решение должно быть устойчивым в контексте личного интереса каждого агента.

  Эту пару вопросов можно задать практически для любой задачи, в которой агенты с собственными интересами генерируют коллективное решение. Мы сейчас ответим на оба вопроса для задачи маршрутизации при многоадресной передаче. Как выясняется, пример на рис. 12.10 отражает некоторые критические аспекты задачи в целом. Мы покажем, что для любого экземпляра динамика наилучших ответов, начинающаяся с социального оптимума, приводит к равновесию Нэша, стоимость которого увеличивается не более чем с коэффициентом $H(k) = \Theta(\log k)$.

Поиск хорошего равновесия Нэша

Сначала мы покажем, что динамика наилучших ответов в нашей задаче всегда приводит к равновесию Нэша. Заодно выясняется, что наш метод решения этой задачи также предоставляет необходимые средства для ограничения цены устойчивости.

Ключевая идея заключается в том, что общую стоимость для всех агентов не обязательно использовать как метрику прогресса, по которой ограничивается количество шагов динамики наилучших ответов. Вместо нее подойдет любая величина, которая строго убывает при обновлении пути любым агентом и которая может уменьшаться конечное количество раз. С учетом этого мы попытаемся сформулировать метрику, обладающую таким свойством. Эта метрика может быть не такой сильной, как общая стоимость с ее интуитивным смыслом, но это нормально — лишь бы она делала то, что нужно.

Для начала более подробно разберемся, почему общая стоимость агентов не подходит. Возьмем простой пример: агент t_j в настоящее время совместно использует вместе с x другими агентами путь, состоящий из одного ребра e. (Конечно, в общем случае пути агентов длиннее, но пути из одного ребра упрощают анализ данного примера.) Допустим, агент t_j решает, что для снижения стоимости стоит переключиться на путь, состоящий из одного ребра f, которое в настоящее время не используется ни одним агентом. Чтобы это произошло, должно выполняться условие $c_f < c_e/(x + 1)$. Теперь в результате переключения общая стоимость для всех агентов возрастает на c_f: ранее $x + 1$ агентов вносили свой вклад в стоимость c_e, и никто не оплачивал стоимость c_f; после переключения x агентов продолжают совместно платить полную стоимость c_e, а t_j теперь оплачивает дополнительную стоимость c_f.

Чтобы рассматривать происходящее как метрику прогресса, необходимо заново определить, что же понимать под «прогрессом». В частности, будет полезно иметь метрику, которая могла бы компенсировать дополнительную стоимость c_f некоторым выражением того, что общая «потенциальная энергия» системы снизилась на $c_e/(x + 1)$. Это позволило бы нам рассматривать перемещение t_j как снижение суммы, так как $c_f < c_e/(x + 1)$. Для этого можно определить для каждого ребра e «потенциал» с тем свойством, что при уменьшении количества агентов, использующих e, с $x + 1$ до x, этот потенциал уменьшается на $c_e/(x + 1)$. (Соответственно потенциал должен увеличиваться на ту же величину при увеличении количества агентов, использующих e, с x до $x + 1$).

Интуиция подсказывает, что потенциал следует определить так, чтобы при x агентах на ребре e потенциал уменьшался на c_e/x, когда первый агент перестает использовать e; на $c_e/(x - 1)$ — когда следующий перестает использовать e; на $c_e/(x - 2)$ — для следующего, и т. д. Задача легко решается выбором потенциала $c_e(1/x + 1/(x - 1) + ... + 1/2 + 1)=c_e \cdot H(x)$. Конкретнее, *потенциал* множества путей $P_1, P_2, ..., P_k$ обозначается $\Phi(P_1, P_2, ..., P_k)$ и определяется следующим образом. Пусть для каждого ребра e количество агентов, пути которых используют ребро e, обозначается x_e. Тогда

$$\Phi(P_1, P_2,\ldots, P_k) = \sum_{e\in E} c_e \cdot H(x_e).$$

(Мы определяем гармоническое число $H(0)$ равным 0, чтобы вклад ребер, не входящих в пути, был равен 0.)

Следующее утверждение устанавливает, что Φ действительно может использоваться в качестве метрики прогресса.

(12.14) Предположим, имеется текущее множество путей $P_1, P_2, ..., P_k$ и агент t_j обновляет свой путь с P_j на P_j'. Тогда новый потенциал $\Phi(P_1, ..., P_{j-1}, P_j', P_{j+1}, ..., P_k)$ строго меньше старого потенциала $\Phi(P_1, ..., P_{j-1}, P_j, P_{j+1}, ..., P_k)$.

Доказательство. Прежде чем агент t_j изменит свой путь с P_j на P_j', он платит $\sum_{e \in P_j} c_e / x_e$, так как стоимость каждого ребра e разделяется с $x_e - 1$ другими агентами. После переключения он продолжает платить свою стоимость для ребер в пересечении $P_j \cap P_j'$, а также платит $c_f / (x_f + 1)$ для каждого ребра $f \in P_j' - P_j$. Таким образом, тот факт, что t_j рассматривает это переключение как улучшение, означает, что

$$\sum_{f \in P_j' - P_j} \frac{c_f}{x_f + 1} < \sum_{e \in P_j - P_j'} \frac{c_e}{x_e}.$$

Теперь спросим себя, что происходит с потенциальной функцией Φ. Она изменяется только на ребрах $P_j' - P_j$ и $P_j - P_j'$. На первом множестве она увеличивается на

$$\sum_{f \in P_j' - P_j} c_f \left[H(x_f + 1) - H(x_f) \right] = \sum_{f \in P_j' - P_j} \frac{c_f}{x_f + 1},$$

а на втором уменьшается на

$$\sum_{e \in P_j - P_j'} c_e \left[H(x_e) - H(x_e - 1) \right] = \sum_{e \in P_j - P_j'} \frac{c_e}{x_e}.$$

Итак, тот критерий, что t_j используется для переключения путей, в точности соответствует утверждению о том, что общее увеличение строго меньше общего уменьшения, а следовательно, потенциал Φ в результате переключения пути t_j уменьшается. ■

Для каждого агента t_j существует конечное число вариантов выбора пути, и из (12.14) следует, что динамика наилучших ответов никогда не сможет вернуться к множеству путей $P_1, ..., P_k$ после того, как покинет его из-за улучшающего перемещения некоторого агента. Таким образом, нам удалось показать следующее:

(12.15) Динамика наилучших ответов всегда приводит к множеству путей, образующему решение с равновесием Нэша.

Ограничение цены устойчивости

Потенциальная функция Φ также оказывается очень полезной для определения цены стабильности. Дело в том, что хотя Φ и не равна общей стоимости, сформированной всеми агентами, она достаточно близка к ней.

Чтобы убедиться в этом, обозначим $C(P_1, ..., P_k)$ полную стоимость для всех агентов при выборе путей $P_1, ..., P_k$. Эта величина просто представляет собой сумму c_e по всем ребрам, входящим в объединение этих путей, поскольку стоимость каждого такого ребра полностью покрывается агентами, в пути которых оно входит.

Теперь отношение между функцией стоимости C и потенциальной функцией Φ может быть выражено следующим образом:

(12.16) Для любого множества путей $P_1,...,P_k$ выполняется

$$C(P_1, ..., P_k) \le \Phi(P_1, ..., P_k) \le H(k) \cdot C(P_1, ..., P_k).$$

Доказательство. Вспомните, что ранее количество путей, содержащих ребро e, обозначалось x_e. Для сравнения C с Φ мы также определяем E^+ как множество всех ребер, принадлежащих минимум одному из путей $P_1, ..., P_k$. Тогда по определению C имеем $C(P_1, ..., P_k) = \sum_{e \in E} + c_e$.

Обратите внимание на простой факт: $x_e \le k$ для всех e. Теперь мы просто записываем

$$C(P_1, ..., P_k) = \sum_{e \in E^+} c_e \le \sum_{e \in E^+} c_e H(x_e) = \Phi(P_1, ..., P_k)$$

и

$$\Phi(P_1, ..., P_k) = \sum_{e \in E^+} c_e H(x_e) \le \sum_{e \in E^+} c_e H(k) = H(k) \cdot C(P_1, ..., P_k). \blacksquare$$

Это позволяет определить границу для цены устойчивости:

(12.17) В каждом экземпляре существует решение с равновесием Нэша, для которого общая стоимость всех агентов превышает стоимость социального оптимума не более чем на множитель $H(k)$.

Доказательство. Для получения желаемого равновесия Нэша мы начнем с социального оптимума, состоящего из путей $P^*_1, ..., P^*_k$, и выполним динамику наилучших ответов. Согласно (12.15), это должно привести к завершению в равновесии Нэша $P_1, ..., P_k$.

Во время выполнения динамики наилучших ответов общая стоимость для всех агентов может возрастать, но, согласно (12.14), потенциальная функция уменьшается. Следовательно, $\Phi(P_1, ..., P_k) \le \Phi(P_1^*, ..., P_k^*)$.

И это практически все, что необходимо, потому что для любого множества путей величины C и Φ различаются не более чем на множитель $H(k)$. А именно:

$$C(P_1, ..., P_k) \le \Phi(P_1, ..., P_k) \le \Phi(P_1^*, ..., P_k^*) \le H(k) \cdot C(P_1^*, ..., P_k^*). \blacksquare$$

Итак, мы показали, что равновесие Нэша всегда существует и всегда существует равновесие Нэша, общая стоимость которого лежит в пределах множителя $H(k)$ от социального оптимума. Пример на рис. 12.10 показывает, что в худшем случае улучшить границу $H(k)$ не удастся.

Хотя эти утверждения очень хорошо резюмируют некоторые аспекты задачи, остается ряд вопросов, ответы на которые до сих пор неизвестны. Особенно интересен вопрос о том, возможно ли построить равновесие Нэша для этой задачи за полиномиальное время. Следует заметить, что наше доказательство существования равновесия Нэша утверждает лишь то, что в процессе перебора множества путей динамика наилучших ответов никогда не может посетить одно множество дважды, а следовательно, не может выполняться бесконечно. Однако множество возможных путей может быть экспоненциально большим, поэтому такая

формулировка не дает алгоритма с полиномиальным временем. Помимо вопроса об эффективном нахождении любого равновесия Нэша, также остается открытым вопрос об эффективном нахождении равновесия Нэша, достигающего границы $H(k)$ относительно социального оптимума в соответствии с гарантиями (12.17).

Также важно повторить один факт, уже упоминавшийся ранее: нетрудно найти задачи, для которых динамики наилучших ответов могут выполняться в бесконечном цикле и для которых равновесие Нэша может не существовать. Нам повезло, что динамика наилучших ответов могла рассматриваться как итеративное улучшение *потенциальной функции*, гарантирующее продвижение к равновесию Нэша, но дело в том, что такие потенциальные функции существуют не во всех задачах со взаимодействием агентов.

Наконец, интересно сравнить то, чем мы здесь занимались, с задачей, рассматривавшейся ранее в этой главе — поиском устойчивой конфигурации в сети Хопфилда. Если вы припомните обсуждение той задачи, мы анализировали процесс, в котором каждый узел «переключался» между двумя возможными состояниями, стремясь к увеличению общего веса «хороших» ребер, инцидентных ему. Это можно рассматривать как экземпляр динамики наилучших ответов для задачи, в которой у каждого узла имеется целевая функция, стремящаяся к максимизации метрики веса хороших ребер. Однако продемонстрировать схождение динамики наилучших ответов для сети Хопфилда было намного проще, чем в данном случае: тогда оказалось, что процесс переключения состояния был «замаскированной» формой локального поиска, у которого целевая функция получалась простым суммированием целевых функций всех узлов, — по сути, аналог общей стоимости для всех агентов служил метрикой прогресса. В текущем случае именно из-за того, что эта функция общей стоимости не могла служить метрикой прогресса, нам пришлось прибегнуть к более сложным средствам.

Упражнения с решениями

Упражнение с решением 1

Задача о выборе центров из главы 11 тоже может послужить примером для изучения эффективности алгоритмов локального поиска.

Ниже приведен простой пример локального поиска для выбора центров (на самом деле это типичная стратегия для разных задач, связанных с размещением). В этой задаче имеется множество мест $S = \{s_1, s_2, \ldots, s_n\}$ на плоскости, и мы хотим выбрать множество из k центров $C = \{c_1, c_2, \ldots, c_k\}$, у которых радиус покрытия (наибольшее расстояние, на которое жителям любого из мест приходится перемещаться до ближайшего центра) как можно меньше.

Начнем с произвольного выбора k точек на плоскости в качестве центров $c_1, c_2, \ldots, c_k$. Далее поочередно выполняются следующие два шага:

(i) для заданного множества из k центров $c_1, c_2, \ldots, c_k$ множество S делится на k множеств: для $i = 1, 2, \ldots, k$ множество Si определяется как совокупность всех мест, для которых c_i является ближайшим центром;

(ii) для этого разбиения S на k множеств строятся новые центры, как можно более «центральные» относительно них. Для каждого множества S_i находится наименьший круг на плоскости, содержащий все точки S_i, и центр c_i определяется как центр этого круга.

Если шаги (i) и (ii) приводят к строгому уменьшению радиуса покрытия, то выполняется следующая итерация; в противном случае алгоритм останавливается.

Чередование шагов (i) и (ii) основано на естественном взаимодействии между местами и центрами: на шаге (i) производится как можно лучшая разбивка мест для заданных центров, а на шаге (ii) выбирается как можно лучшее расположение центров для заданной разбивки мест. Помимо своей роли как эвристики для размещения, этот тип двухшагового взаимодействия также закладывает основу для алгоритмов локального поиска в статистике, где этот метод (по причинам, в которые мы углубляться не будем) называется *методом максимизации ожидания*.

(a) Докажите, что алгоритм локального поиска в конечном итоге завершается.

(b) Рассмотрим следующее утверждение.

Существует абсолютная константа $b > 1$ (не зависящая от конкретного экземпляра ввода), такая, что при завершении алгоритма локального поиска радиус покрытия его решения не более чем в b раз превышает оптимальный радиус покрытия. Решите, истинно или ложно это утверждение, и приведите доказательство или опровержение.

Доказательство. Первая мысль, которая приходит в голову при доказательстве части (a): множество радиусов покрытия убывает при каждой итерации; они не могут упасть ниже оптимальных значений, так что итерации должны завершиться. Но здесь необходима осторожность, потому что мы имеем дело с вещественными числами. А что, если радиусы покрытия уменьшаются при каждой итерации, но на все меньшую и меньшую величину, и алгоритм может выполняться сколь угодно долго при условии, что радиусы покрытия сходятся к некоторой величине?

Впрочем, справиться с этой проблемой не так уж сложно. Заметим, что радиус покрытия в конце шага (ii) каждой итерации полностью определяется текущим разбиением мест на множества $S_1, S_2, ..., S_k$. Существует конечное число способов разбиения мест на k множеств, и если бы алгоритм локального поиска выполнялся более этого количества итераций, он должен был бы произвести одно разбиение в двух итерациях. Но тогда в обеих итерациях был бы создан одинаковый радиус покрытия, а это противоречит предположению о том, что радиус покрытия строго убывает между итерациями. Это доказывает, что алгоритм всегда завершается. (Обратите внимание: он устанавливает только экспоненциальную границу количества итераций, так как количество разбиений множества мест на k множеств экспоненциально.)

Чтобы опровергнуть утверждение из части (b), будет достаточно найти пример выполнения алгоритма, в котором итерации «застревают» в конфигурации с очень большим радиусом покрытия. Сделать это несложно: для любой константы $b > 1$ рассмотрим множество S из четырех точек на плоскости, образующих высокий узкий прямоугольник с шириной w и высотой $h = 2bw$. Например, это могут быть четыре точки $(0,0)$, $(0,h)$, (w,h), $(w,0)$.

Теперь предположим, что $k = 2$, и начнем с двух центров слева и справа от прямоугольника (допустим, $(-1, h/2)$ и $(w + 1, h/2)$). Первая итерация проходит следующим образом:

- на шаге (i) S делится на две точки S_1 в левой части прямоугольника (с x-координатой 0) и две точки S_2 в правой части прямоугольника (с x-координатой w);
- на шаге (ii) центры размещаются в средних точках S_1 и S_2 (то есть $(0,h/2)$ и $(w,h/2)$).

Можно убедиться в том, что при следующей итерации разбиение S не изменится, а следовательно, местонахождение центров тоже не изменится; здесь алгоритм завершается на локальном минимуме.

Радиус покрытия этого решения равен $h/2$, тогда как в оптимальном решении центры должны располагаться в средних точках верхней и нижней стороны прямоугольника с радиусом покрытия $w/2$. Таким образом, в нашем решении радиус покрытия в $h/w = 2b > b$ раз больше оптимума.

Упражнения

1. Рассмотрим задачу нахождения устойчивого состояния в нейронной сети Хопфилда в специальном случае, когда все веса ребер положительны. Это соответствует задаче максимального разреза, которая обсуждалась ранее в этой главе: для каждого ребра e в графе G конечные точки предпочитают находиться в противоположных состояниях.

   Теперь предположим, что граф G является связным и двудольным; узлы можно разбить на множества X и Y так, что один конец каждого ребра принадлежит X, а другой Y. Тогда существует естественная «лучшая» конфигурация для сети Хопфилда, в которой все узлы X имеют состояние +1, а все узлы Y имеют состояние −1. В этом случае все ребра являются хорошими, так как их концы находятся в противоположных состояниях.

   Вопрос в следующем: в этом частном случае, когда лучшая конфигурация настолько очевидна, сможет ли алгоритм переключения состояния, описанный в тексте (пока остаются нереализованные узлы, выбрать один и переключить его состояние), всегда найти его конфигурацию? Приведите доказательство или пример входного экземпляра, начальной конфигурации и выполнения алгоритма переключения состояния, завершающегося в состоянии, в котором не все ребра являются хорошими.

2. Вспомните, что для задачи, в которой целью является максимизация некоторой используемой величины, у градиентного спуска имеется естественная «восходящая» аналогия, в которой происходят многократные переходы от текущего решения к решению со строго большей величиной. Вполне естественно называть эту аналогию *алгоритмом градиентного подъема*.

   По строгой симметрии замечания, сделанные в этой главе по поводу градиентного спуска, переносятся на градиентный подъем: для многих задач выполнение

алгоритма может закончиться в локальном оптимуме, который оказывается не слишком хорошим. Но иногда встречаются задачи (как, например, задачи о максимальном разрезе и разметке), в которых алгоритм локального поиска предоставляет очень сильную гарантию: каждый локальный оптимум близок по значению к глобальному оптимуму. Если рассмотреть задачу о двудольном паросочетании, выясняется, что аналогичное явление встречается и в этом случае.

Итак, рассмотрим следующий алгоритм градиентного подъема для нахождения паросочетания в двудольном графе:

Пока существует ребро, конечные точки которого свободны, добавить его в текущее паросочетание. Когда таких ребер не останется, завершить с локально оптимальным паросочетанием.

(a) Приведите пример двудольного графа G, для которого этот алгоритм градиентного подъема не возвращает максимальное паросочетание.

(b) Имеются M и M' — паросочетания в двудольном графе G. Предположим, что $|M'| > 2|M|$. Покажите, что существует ребро $e' \in M'$, для которого $M \cup \{e'\}$ является паросочетанием в G.

(c) Используйте (b) для доказательства того, что любое локально оптимальное паросочетание в двудольном графе G, возвращаемое алгоритмом градиентного подъема, не менее по крайней мере *половины* максимального паросочетания в G.

3. Крупная биотехническая компания проводит эксперименты на двух дорогих высокопроизводительных машинах — обозначим их M_1 и M_2 (считается, что оборудование машин идентично). Каждый день появляется множество заданий, которые должны быть выполнены, и каждое задание должно быть назначено на одну из двух машин. Вам предложено помочь в составлении алгоритма распределения заданий между машинами, при котором ежедневные нагрузки остаются сбалансированными. Задача формулируется следующим образом: имеются n заданий, на выполнение каждого задания j требуется время t_j. Задания необходимо разбить на две группы A и B; множество A передается на машину M_1, а множество B — на машину M_2. Время, необходимое для обработки всех заданий на двух машинах, равно $T_1 = \sum_{j \in A} t_j$ и $T_2 = \sum_{j \in B} t_j$. Требуется добиться того, чтобы две машины были заняты приблизительно одинаковое время, то есть минимизировать $|T_1 - T_2|$.

Консультант, ранее участвовавший в работе, показал, что задача является *NP*-сложной (посредством сведения от задачи о сумме подмножеств). Теперь компания ищет хороший алгоритм локального поиска и предлагает следующий способ: начать с произвольного назначения заданий на две машины (допустим, задания $1, n/2$ на машину M_1, остальные на машину M_2). Локальными переходами являются перемещения одного задания с одной машины на другую, причем задания перемещаются только тогда, когда перемещение приводит к абсолютному уменьшению времени обработки. Вам предложено ответить на основные вопросы относительно эффективности этого алгоритма.

(a) Первый вопрос: насколько эффективно такое решение? Предположим, не существует одного задания, доминирующего по затратам времени обра-

ботки, то есть $t_j \leq \frac{1}{2}\sum_{i=1}^{n} t_i$ для всех заданий j. Докажите, что для каждого локально оптимального решения время работы двух машин приблизительно сбалансировано: $\frac{1}{2}T_1 \leq T_2 \leq 2T_1$.

(b) На следующем месте стоит время выполнения алгоритма: как часто задания будут перемещаться туда и обратно между двумя машинами? Вы предлагаете внести в алгоритм небольшое изменение. Если при локальном переходе много разных заданий перемещается с одной машины на другую, то алгоритм всегда должен перемещать задание j с максимальным t_j. Докажите, что в этом варианте каждое задание будет перемещено не более одного раза (а следовательно, локальный поиск завершится не более чем за n переходов).

(c) Наконец, ваши работодатели интересуются, нужно ли заниматься поисками улучшенного алгоритма. Приведите пример, в котором описанный выше алгоритм локального поиска не приводит к оптимальному решению.

4. Рассмотрим задачу распределения нагрузки из раздела 11.1. Ваши знакомые управляют группой веб-серверов; они разработали для задачи эвристику локального поиска, отличную от алгоритмов, описанных в главе 11.

   Напомним, что в формулировке задачи имеются m машин M_1, ...,M_m и каждое задание требуется назначить на машину. Нагрузка i-го задания обозначается t_i. *Периодом обработки* называется максимальная нагрузка по всем машинам:

$$\max_{machines\ M_i} \sum_{jobs\ j\ assigned\ to\ M_i} t_j.$$

   Эвристика локального поиска работает следующим образом: алгоритм начинает с произвольного распределения заданий между машинами и пытается многократно применить следующую «перестановку»:

   Пусть $A(i)$ и $A(j)$ — задания, назначенные на машины M_i и M_j соответственно. Чтобы выполнить перестановку M_i и M_j, следует выбрать подмножества заданий $B(i) \subseteq A(j)$ и $B(j) \subseteq A(j)$ и «переставить» их между двумя машинами: то есть $A(i)$ заменяется на $A(i) \cup B(j) - B(i)$, а $A(j)$ заменяется на $A(j) \cup B(i) - B(j)$. (Разрешается $B(i) = A(i)$, или $B(i)$ может быть пустым множеством; аналогично для $B(j)$.)

   Рассмотрим перестановку, примененную к машинам M_i и M_j. Предположим, нагрузки на M_i и M_j перед перестановкой равны T_i и T_j соответственно, а после перестановки — T'_i и T'_j. Перестановка называется *улучшающей*, если $\max(T'_i, T'_j) < \max(T_i, T_j)$, другими словами, если большая из двух задействованных нагрузок строго уменьшилась. Распределение заданий между машинами называется *устойчивым*, если не существует улучшающей перестановки, начинающейся с текущего распределения.

   Таким образом, эвристика локального поиска просто продолжает выполнять улучшающие перестановки до тех пор, пока не будет достигнуто устойчивое распределение; в этот момент полученное устойчивое распределение возвращается как решение.

Пример. Допустим, имеются две машины: в текущем распределении на машину M_1 назначены задания с размерами 1, 3, 5, 8, а на машину M_2 — задания с размерами 2, 4. В этом случае единственная улучшающая перестановка определяет $B(1)$ как множество из одного задания с размером 8, а $B(2)$ — из задания с размером 2. После перестановки этих двух множеств полученное распределение состоит из заданий с размерами 1, 2, 3, 5 на машине M_1 и заданий с размерами 4, 8 на машине M_2. Это распределение является устойчивым (а также имеет оптимальный период обработки 12).

(a) Для этого определения не существует однозначных гарантий того, что эта эвристика локального поиска всегда завершается. А что, если она будет бесконечное перебирать распределения, которые не являются устойчивыми?

Докажите, что в действительности эта эвристика локального поиска завершается за конечное число шагов и приводит к устойчивому распределению для любого экземпляра.

(b) Покажите, что любое устойчивое распределение имеет период обработки, отличающийся от минимально возможного периода обработки не более чем в 2 раза.

Примечания и дополнительная литература

Киркпатрик, Гелатт и Веччи (Kirkpatrick, Gelatt, Vecchi, 1983) разработали метод имитации отжига на основе алгоритма Метрополиса и др. (Metropolis et al., 1953), предназначенного для моделирования физических систем. При этом они выделили аналогию между энергетическими поверхностями и пространствами решений вычислительных задач.

В сборнике под редакцией Аартса и Ленстры (Aarts, Lenstra, 1997) рассматриваются многочисленные примеры использования методов локального поиска в алгоритмических задачах. Нейронные сети Хопфилда были предложены Хопфилдом (Hopfield, 1982) и проанализированы более подробно в книге Хайкина (Haykin, 1999). Эвристику разбиения графа из раздела 12.5 разработали Керниган и Лин (Kernighan, Lin, 1970).

Классификацию алгоритмов локального поиска, основанная на задаче разметки, разработали Бойков, Векслер и Забих (Boykov, Veksler, Zabih, 1999). Дальнейшие результаты и вычислительные эксперименты рассматриваются в диссертации Векслера (Veksler, 1999).

Задача многоагентной маршрутизации из раздела 12.7 поднимает вопросы, находящиеся на пересечении алгоритмов и теории игр — области, связанной с общими вопросами стратегического взаимодействия между агентами. Книга Осборна (Osborne, 2003) содержит введение в теорию игр; алгоритмические аспекты вопроса изложены в работах Пападимитриу (Papadimitriou, 2001) и Тардоса (Tardos, 2004), а также в диссертации и последующей книге Рафгардена (Roughgarden, 2002, 2004). Применение потенциальных функций для доказательства существования равновесия Нэша имеет долгую историю в теории игр (Beckmann, McGuire, Winsten, 1956; Rosenthal, 1973), а потенциальные функции были применены для анализа динамик наилучших ответов Мондерером и Шепли (Monderer, Shapley, 1996). Граница цены устойчивости для задачи маршрутизации из раздела 12.7 предложена Аншелевичем и др. (Anshelevich, 2004).

Глава 13
Рандомизированные алгоритмы

Идея «случайности» процесса не нова; само понятие возникло давно в истории человеческой мысли. Оно отражено в азартных играх и страховом деле — и то и другое происходит из древних времен. Но хотя столь же интуитивно понятные дисциплины, такие как геометрия или логика, изучаются математическими методами уже несколько тысяч лет, область математического изучения вероятности на удивление молода; первые попытки ее серьезной формализации были предприняты в XVII веке. Конечно, компьютерная наука существует на много меньшем историческом отрезке, и случайности в ней уделяется внимание с первых дней.

Темы рандомизация и вероятностного анализа проникают во многие области компьютерной науки, включая разработку алгоритмов, и когда речь заходит о случайных процессах в контексте вычислений, обычно имеется в виду одна из двух точек зрения. Первая рассматривает случайное поведение реального мира: здесь часто изучаются традиционные алгоритмы, сталкивающиеся со случайно сгенерированными входными данными. Такие методы часто называются *анализом среднего случая*, потому что поведение алгоритма изучается на «средних» входных данных (подвергнутых воздействию некоторого случайного процесса) вместо худшего случая.

Вторая точка зрения связана со случайным поведением самих алгоритмов: мир всегда поставляет одни и те же входные данные худшего случая, но алгоритму разрешено принимать случайные решения при обработке ввода. Таким образом, рандомизация в этом методе действует исключительно внутри алгоритма и не требует новых допущений относительно природы входных данных. Именно этой концепции *рандомизированных алгоритмов* посвящена данная глава.

Чем может быть полезен алгоритм, принимающий случайные решения? Во-первых, рандомизация расширяет возможности используемой модели. Эффективные детерминированные алгоритмы, которые всегда дают правильный ответ, могут рассматриваться как особый случай эффективных рандомизированных алгоритмов, которые дают правильный ответ с очень высокой вероятностью; они также являются особым случаем рандомизированных алгоритмов, которые всегда работают правильно и при этом, *как ожидается*, выполняются эффективно. Даже при получении худших входных данных алгоритм, выполняющий свою «внутрен-

нюю» рандомизацию, может компенсировать некоторые нежелательные аспекты худшего случая. Следовательно, задачи, которые не решаются эффективными детерминированными алгоритмами, могут не устоять перед рандомизированными алгоритмами.

Но это еще не все. Мы рассмотрим рандомизированные алгоритмы для некоторых задач, для которых существуют сравнительно эффективные детерминированные алгоритмы. Даже в таких ситуациях рандомизация часто проявляет немалую мощь: например, она может быть концептуально проще или она может обеспечить функционирование алгоритма при минимальных потребностях в хранении внутреннего состояния или информации о прошлых событиях. Преимущества рандомизации расширяются при рассмотрении крупных компьютерных систем и сетей с множеством гибких взаимодействий между процессами, другими словами, в *распределенных системах*. Здесь случайное поведение части отдельных процессов может сократить объем необходимого обмена информации или синхронизации; рандомизация часто оказывается ценным инструментом для нарушения симметрии между процессами, снижающим вероятность конфликтов и появления «горячих точек». Многие из примеров будут взяты из ситуаций такого рода: управление доступом к общему ресурсу, распределение нагрузки по нескольким процессорам, маршрутизация пакетов в сети и т. д. Даже небольшой опыт работы с рандомизированными эвристиками может сильно пригодиться при анализе больших систем.

При изучении рандомизированных алгоритмов возникает естественная проблема: они требуют значительных знаний в области теории вероятности. Конечно, всегда лучше знать больше, чем меньше, и некоторые алгоритмы действительно базируются на сложных вероятностных идеях. Но в этой главы мы также постараемся показать, что для понимания многих известных алгоритмов из этой области о теории вероятности в действительности достаточно знать совсем немного. Вы увидите, что существует небольшое количество часто применяемых вероятностных инструментов, и в этой главе мы постараемся разрабатывать такие инструменты наряду с алгоритмами. В конечном итоге опыт использования этого инструментария не менее важен, чем понимание конкретных алгоритмов.

13.1. Первое применение: разрешение конфликтов

Начнем с первого применения рандомизированных алгоритмов — разрешения конфликтов в распределенных системах. Этот пример демонстрирует общий стиль анализа, который будет использоваться во многих дальнейших алгоритмах. В частности, он дает возможность попрактиковаться в основных операциях, относящихся к *событиям* и их вероятностям, анализу пересечений событий с использованием *независимости* и объединений событий. Для полноты мы приведем краткую сводку основных концепций в завершающем разделе этой главы (раздел 13.15).

Задача

Допустим, имеется n процессов $P_1, P_2, ..., P_n$, конкурирующих за доступ к одной базе данных. Будем считать, что время делится на *кванты*. В одном кванте времени с базой данных может работать не более чем один процесс; если два и более процесса пытаются одновременно обратиться к базе данных, эти процессы «блокируются» до конца кванта. Следовательно, хотя каждый процесс хочет как можно чаще обращаться к базе данных, пытаться обращаться к базе данных всем процессам в каждом кванте бессмысленно; в этом случае все процессы будут постоянно находиться в заблокированном состоянии. Нужен справедливый механизм распределения квантов между процессами, чтобы все процессы имели возможность регулярно работать с базой данных.

Если передача данных между процессами реализуется достаточно просто, можно представить себе всевозможные средства для прямого разрешения конфликта. Но предположим, что процессы вообще не могут передавать информацию друг другу; как определить протокол, по которому они смогут «поочередно» работать с базой данных?

Разработка рандомизированного алгоритма

Рандомизация предоставляет естественный протокол для этой задачи, который определяется очень просто. Для некоторого числа $p > 0$, которое будет определено ниже, каждый процесс пытается обратиться к базе данных в каждом кванте с вероятностью p независимо от решений других процессов. Итак, если в каком-либо кванте ровно один процесс выдает такое обращение, его попытка завершается успешно; если обращения поступают от двух и более процессов, они блокируются; и если ни одной попытки не было, то квант «пропадает». Такая стратегия, рандомизирующая поведение каждого процесса из множества одинаковых процессов, лежит в основе уже упоминавшейся парадигмы нарушения симметрии: если все процессы действуют синхронно, многократно пытаясь обратиться к базе данных в одно и то же время, никакого прогресса не будет; рандомизация же позволяет «сгладить» конкурентную борьбу.

Анализ алгоритма

Как и во многих других примерах использования рандомизации, алгоритм в этом случае формулируется очень просто; основной интерес представляет прежде всего анализ его эффективности.

Определение основных событий

Анализ вероятностных систем такого рода полезно начать с определения основных событий и их вероятностей. Первое событие: для заданного процесса P_i и заданного кванта времени t событие $A[i, t]$ обозначает попытку обращения P_i к базе данных

в кванте *t*. Известно, что каждый процесс пытается обратиться к базе данных с вероятностью *p*, так что вероятность этого события для любых *i* и *t* равна $\Pr[A[i, t]] = p$. Для каждого события также существует *дополняющее событие*, которое означает, что основное событие не произошло; в данном случае дополняющее событие $\overline{A[i, t]}$ означает, что процесс P_i не пытался обратиться к базе данных в кванте *t*, а его вероятность равна

$$\Pr\left[\overline{A[i, t]}\right] = 1 - \Pr[A[i, t]] = 1 - p.$$

Главный вопрос — *удастся ли* процессу обратиться к базе данных в заданном кванте. Обозначим это событие $S[i, t]$. Очевидно, процесс P_i должен хотя бы попытаться обратиться к базе данных в кванте *t*. Успешное обращение означает, что процесс P_i попытался обратиться к базе данных в кванте *t*, а все остальные процессы не пытались обращаться к базе данных в кванте *t*. Следовательно, $S[i, t]$ равно пересечению события $A[i,t]$ с дополняющими событиями $\overline{A[j, t]}$ для $j \neq i$:

$$S[i, t] = A[i, t] \cap \left(\bigcap_{j \neq i} \overline{A[j, t]}\right).$$

Все события в этом пересечении независимы по определению протокола разрешения конфликтов. Следовательно, вероятность $S[i, t]$ вычисляется умножением вероятностей всех событий в пересечении:

$$\Pr[S[i, t]] = \Pr[A[i, t]] \cdot \prod_{j \neq i} \Pr\left[\overline{A[i, t]}\right] = p(1-p)^{n-1}.$$

Мы получили удобное выражение в замкнутой форме для вероятности того, что P_i успешно обратится к базе данных в кванте *t*; теперь уместно задаться вопросом, как выбрать *p* для максимизации вероятности успеха. Сначала заметим, что вероятность успеха равна 0 для крайних случаев $p = 0$ и $p = 1$ (в которых либо процессы вообще не пытаются обратиться к базе данных, либо каждый процесс пытается обратиться к базе данных в каждом кванте, так что в итоге все процессы блокируются). Функция $f(p) = p(1-p)^{n-1}$ положительна для значений *p* из диапазона от 0 до 1, а ее производная $f'(p) = (1-p)^{n-1} - (n-1)p(1-p)^{n-2}$ равна нулю только в точке $p = 1/n$, где и достигается максимум. Таким образом, вероятность успеха максимизируется при выборе $p = 1/n$. (Стоит заметить, что $1/n$ — естественный и интуитивный вариант в том случае, если в каждом кванте попытка обращения должна исходить ровно от одного процесса.)

Выбирая $p = 1/n$, получаем $\Pr[S[i, t]] = \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}$. Полезно примерно представить асимптотическое поведение этого выражения с помощью следующего факта из курса математического анализа.

(13.1) (a) Функция $\left(1-\frac{1}{n}\right)^n$ монотонно сходится от $\frac{1}{4}$ к $\frac{1}{e}$ при увеличении n, начиная с 2.

(b) Функция $\left(1-\frac{1}{n}\right)^{n-1}$ монотонно сходится от $\frac{1}{2}$ к $\frac{1}{e}$ при увеличении n, начиная с 2.

Из (13.1) следует, что $1/(en) \le \Pr[S[i, t]] \le 1/(2n)$, а следовательно, значение $\Pr[S[i, t]]$ асимптотически равно $\Theta(1/n)$.

Ожидание успешного обращения со стороны конкретного процесса

Рассмотрим этот протокол с оптимальным значением вероятности обращения $p = 1/n$. Допустим, нас интересует, сколько времени должно пройти, чтобы процесс P_i успешно обратился к базе данных хотя бы один раз. Из предыдущих вычислений видно, что вероятность успеха в любом отдельном кванте не очень хороша при достаточно больших n. Но как насчет серии из нескольких квантов?

Обозначим $\mathcal{F}[i, t]$ «событие неудачи», при котором процесс P_i не добивается успеха во всех квантах с 1 до t. Очевидно, оно представляет собой обычное пересечение дополняющих событий $\overline{S[i,r]}$ для $r = 1, 2, ..., t$. Кроме того, поскольку все эти события независимы, вероятность $\mathcal{F}[i,t]$ вычисляется умножением:

$$\Pr[\mathcal{F}[i,t]] = \Pr\left[\bigcap_{r=1}^{t}\overline{S[i,r]}\right] = \prod_{r=1}^{t}\Pr\left[\overline{S[i,r]}\right] = \left[1-\frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}\right]^t.$$

Эти вычисления дают значение искомой вероятности, но если и дальше так продолжать, придется иметь дело с чрезвычайно сложными выражениями, поэтому важно переходить на асимптотическое мышление. Вспомните, что вероятность успеха после одного кванта равна $\Theta(1/n)$, а точнее, она лежит в границах от $1/(en)$ до $1/(2n)$. Используя приведенное выше выражение, имеем

$$\Pr[F[i,t]] = \prod_{r=1}^{t}\Pr\left[\overline{S[i,r]}\right] \le \left(1-\frac{1}{en}\right)^t.$$

Теперь заметим, что если задать $t = en$, то мы получим выражение, которое можно напрямую подставить в (13.1). Конечно, en не будет целым числом, поэтому мы возьмем $t = \lceil en \rceil$ и получим:

$$\Pr[\mathcal{F}[i,t]] \le \left(1-\frac{1}{en}\right)^{\lceil en \rceil} \le \left(1-\frac{1}{en}\right)^{en} \le \frac{1}{e}.$$

Это чрезвычайно компактное и полезное асимптотическое утверждение: вероятность того, что процесс P_i не добьется успеха ни в одном из квантов с 1 по $\lceil en \rceil$,

ограничивается сверху константой e^{-1} независимо от n. Теперь при увеличении t с очень малым коэффициентом вероятность того, что P_i не добьется успеха ни в одном из квантов с 1 по t, резко падает: если задать $t = \lceil en \rceil \cdot (c \ln n)$, то получаем

$$\Pr[\mathcal{F}[i,t]] \leq \left(1 - \frac{1}{en}\right)^t = \left(\left(1 - \frac{1}{en}\right)^{\lceil en \rceil}\right)^{c \ln n} \leq e^{-c \ln n} = n^{-c}.$$

Итак, асимптотически ситуацию можно рассматривать следующим образом: после $\Theta(n)$ квантов вероятность того, что процесс P_i еще не смог обратиться к базе данных, ограничивается константой; а с этого момента до $\Theta(n \ln n)$ эта вероятность падает до очень малой величины, ограничиваемой обратной полиномиальной зависимостью от n.

Ожидание успешного обращения всех процессов

Наконец, мы добрались до вопроса, неявно подразумевавшегося в общей постановке задачи: сколько квантов должно пройти, чтобы все процессы с достаточно высокой вероятностью обратились к базе данных хотя бы один раз?

Чтобы ответить на него, будем считать, что выполнение протокола привело к неудаче после t квантов, если какой-то процесс так и не смог обратиться к базе данных. Обозначим $\mathcal{F}_t$ событие неудачи протокола после t квантов; наша цель — найти разумно малое значение t, для которого значение $\Pr[\mathcal{F}_t]$ мало.

Событие $\mathcal{F}_t$ происходит в том, и только в том случае, если происходит одно из событий $\mathcal{F}[i, t]$; мы можем записать

$$F_t = \bigcup_{i=1}^{n} F[i, t].$$

Ранее мы рассматривали пересечения независимых событий, с которыми было очень просто работать; на этот раз речь идет об объединении событий, которые независимыми не являются. Точное вычисление вероятностей подобных объединений может быть очень сложным, и во многих случаях достаточно воспользоваться простой *границей объединения*, которая гласит, что вероятность объединения событий ограничивается сверху суммой их вероятностей:

(13.2) (Граница объединения) Для заданных событий $\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_n$ выполняется

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] \leq \sum_{i=1}^{n} \Pr[\mathcal{E}_i].$$

Обратите внимание: речь не идет о равенстве; однако верхняя граница достаточно хороша в подобных ситуациях, когда объединение в левой части представляет «плохое событие», которого мы пытаемся избежать, и мы хотим ограничить его вероятность в контексте составляющих «плохих событий» в правой части.

Для текущей задачи вспомните, что $\mathcal{F}_t = \bigcup_{i=1}^{n} \mathcal{F}[i,t]$, а следовательно,

$$\Pr[\mathcal{F}_t] \le \sum_{i=1}^{n} \Pr[\mathcal{F}[i,t]].$$

Выражение в правой части представляет собой сумму n слагаемых с одинаковыми значениями; чтобы вероятность $\mathcal{F}_t$ была достаточно малой, необходимо убедиться в том, что каждое слагаемое в правой части существенно меньше $1/n$. Из предшествующего обсуждения мы видим, что выбор $t = \Theta(n)$ недостаточно хорош, поскольку каждое слагаемое в правой части ограничивается только константой. Если выбрать $t = \lceil en \rceil \cdot (c \ln c)$, то $\Pr[\mathcal{F}[i,t]] \le n^{-c}$ для всех i — именно то, что нужно. А конкретно, выбор $t = 2\lceil en \rceil \ln n$ дает нам

$$\Pr[\mathcal{F}_t] \le \sum_{i=1}^{n} \Pr[\mathcal{F}[i,t]] \le n \cdot n^{-2} = n^{-1}.$$

Таким образом, нам удалось показать следующее:

(13.3) С вероятностью не менее $1-n^{-1}$ всем процессам удастся успешно обратиться к базе данных по крайней мере один раз за $t = 2\lceil en \rceil \ln n$ квантов.

Интересно заметить, что если бы мы выбрали значение t, равное $qn \ln n$ для очень малого значения q (вместо фактически использованного коэффициента $2e$), тогда для $\Pr[\mathcal{F}[i,t]]$ была бы получена верхняя граница, большая n^{-1}, а следовательно, соответствующая верхняя граница для общей вероятности неудачи $\Pr[\mathcal{F}_t]$ была бы больше 1, — очевидно, такая верхняя граница полностью бесполезна. Однако, как мы видели, выбирая все большие и большие значения для коэффициента q, можно снизить верхнюю границу $\Pr[\mathcal{F}_t]$ до n^{-c} для любой нужной константы c — это очень малая верхняя граница. Итак, в некотором смысле все «действие» в границе объединения быстро происходит в период, когда $t = \Theta(n \ln n)$; с изменением скрытой константы внутри $\Theta(\cdot)$ граница объединения переходит от нулевой полезной информации до предоставления в высшей степени сильной границы вероятности.

Может, это просто артефакт применения границы объединения к нашей верхней границе, или данное свойство присуще наблюдаемому процессу? Мы не будем приводить (довольно громоздкие) вычисления, но при желании можно показать, что для t, равного малой константе, умноженной на $n \ln n$, существует заметная вероятность того, что какому-то процессу так и не удалось обратиться к базе данных. Следовательно, быстрое падение значения $\Pr[\mathcal{F}_t]$ действительно происходит в диапазоне $t = \Theta(n \ln n)$. Для этой задачи, как и для многих задач такого типа, мы в действительности находим асимптотически «правильное» значение t, несмотря на использование слабой на первый взгляд границы объединения.

13.2. Нахождение глобального минимального разреза

Идея рандомизации естественным образом возникает в приведенном примере — модели с множеством процессов, которые не могут взаимодействовать напрямую. Теперь рассмотрим задачу для графов, в которой тема рандомизации возникает довольно неожиданно, так как для этой задачи существуют вполне разумные детерминированные алгоритмы.

Задача

Для заданного ненаправленного графа $G = (V, E)$ *разрез* определяется как разбиение V на два непустых подмножества A и B. Ранее, при рассмотрении сетевых потоков, мы работали с похожим определением разреза *s–t*: тогда для направленного графа $G = (V, E)$ с выделенными узлами источника и стока s и t, разрез *s–t* определялся как разбиение V на такие множества A и B, что $s \in A$ и $t \in B$. Наше новое определение отличается от того: граф стал ненаправленным, и в нем нет ни источника, ни стока.

Для разреза (A, B) в ненаправленном графе G размер (A, B) равен количеству ребер, один конец которых принадлежит A, а другой принадлежит B. *Глобальным минимальным разрезом* называется разрез минимального размера. Термин «глобальный» указывает на то, что разрешены любые разрезы графа; нет ни источника, ни стока. Таким образом, глобальный минимальный разрез может рассматриваться как естественная характеристика «надежности»; это минимальное количество ребер, удаление которых приводит к потере связности графом. Сначала мы убедимся в том, что методы сетевого потока действительно достаточны для нахождения глобального минимального разреза.

(13.4) Существует алгоритм с полиномиальным временем для нахождения глобального минимального разреза в ненаправленном графе G.

Доказательство. Начнем со сходства разрезов в ненаправленных графах и разрезов *s–t* в направленных графах и с того факта, что нам известен оптимальный способ нахождения последних.

Заданный ненаправленный граф $G = (V, E)$ необходимо преобразовать так, чтобы в нем были направленные ребра, источник и сток. Каждое ненаправленное ребро $e = (u, v) \in E$ заменяется двумя противоположно ориентированными направленными ребрами, $e' = (u, v)$ и $e'' = (v, u)$, каждое из которых имеет пропускную способность 1. Обозначим полученный направленный граф G'.

Теперь допустим, что мы выбрали два произвольных узла $s, t \in V$ и нашли минимальный разрез *s–t* в G'. Легко убедиться в том, что если (A, B) является минимальным разрезом в G', то (A, B) также является разрезом минимального размера в G *среди всех разрезов, отделяющих s от t*. Но мы знаем, что глобальный минимальный разрез в G должен отделять s от *чего-то*, так как обе стороны A и B не пусты, и s принадлежит только одной из них. Соответственно мы фиксируем

любой узел $s \in V$ и вычисляем минимальный разрез s–t в G' для всех остальных узлов $t \in V - \{s\}$. Это требует $n - 1$ вычислений направленного минимального разреза, лучшим из которых будет глобальный минимальный разрез графа G. ■

Алгоритм в (13.4) создает сильное впечатление, что нахождение глобального минимального разреза в ненаправленном графе в каком-то смысле является *более сложной* задачей, чем поиск минимального разреза s–t в потоковой сети, так как нам приходится $n - 1$ раз вызывать процедуру для решения последней задачи в нашем методе для решения первой. Но оказывается, это всего лишь иллюзия. Серия все более простых алгоритмов, разработанных в конце 1980-х и начале 1990-х годов, показала, что глобальные минимальные разрезы в ненаправленных графах могут вычисляться так же эффективно, как разрезы s–t, и даже еще эффективнее, причем для этого не нужны ни улучшающие пути, ни даже концепция потока. Кульминационным моментом в этой области стало открытие Дэвидом Каргером в 1992 году *алгоритма стягивания* — рандомизированного метода, качественно более простого по сравнению со всеми предшествующими алгоритмами нахождения глобальных минимальных разрезов. В самом деле, алгоритм настолько прост, что на первый взгляд даже трудно поверить, что он действительно работает.

Разработка алгоритма

Сейчас мы опишем алгоритм стягивания в его простейшей форме. Эта версия, хотя и работает за полиномиальное время, не принадлежит к числу самых эффективных алгоритмов для глобальных минимальных разрезов. Впрочем, последующие оптимизации алгоритма существенно улучшили его время выполнения.

Алгоритм стягивания работает со связным *мультиграфом* $G = (V, E)$; это ненаправленный граф, который может иметь несколько «параллельных» ребер между одной парой узлов. Сначала он случайным образом выбирает в G ребро $e = (u, v)$ и «стягивает» его, как показано на рис. 13.1. В результате появляется новый граф G', в котором u и v порождают новый узел w; все остальные узлы сохраняют свою исходную «личность». Все ребра, у которых один конец равен u, а другой равен v, удаляются из G'. Все остальные ребра e остаются в G', но если один конец ребра равен u или v, то этот конец заменяется новым узлом w. Обратите внимание: даже если в G любые два узла соединены не более чем одним ребром, в G' могут появиться параллельные ребра.

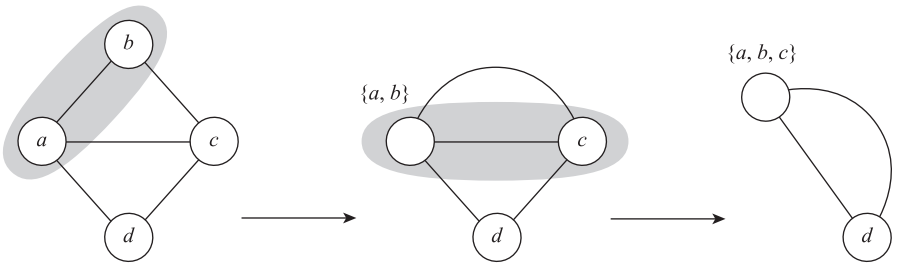


Рис. 13.1. Применение алгоритма стягивания к графу из четырех узлов

Алгоритм стягивания продолжает рекурсивно выполняться с G', случайно выбирая ребра (с равномерным распределением) и стягивая их. По мере продолжения рекурсии вершины G' должны рассматриваться как *суперузлы*: каждый суперузел w соответствует подмножеству $S(w) \subseteq V$, «поглощенному» в результате стягиваний, породивших w. Алгоритм завершается при достижении графа G', состоящего из двух суперузлов v_1 и v_2 (предположительно соединенных несколькими параллельными ребрами). Каждый суперузел v_i имеет соответствующее подмножество $S(v_i) \subseteq V$, которое состоит из стянутых в него узлов, и эти два множества $S(v_1)$ и $S(v_2)$ образуют разбиение V. Далее $(S(v_1), S(v_2))$ выводится как разрез, найденный алгоритмом.

```
Применение алгоритма стягивания к мультиграфу G = (V,E):
   Для каждого узла v хранится
         множество S(v) узлов, стянутых в v
      В исходном состоянии S(v) = {v} для каждого v
   Если G содержит два узла v1 и v2, вернуть разрез (S(v1),S(v2))
   Иначе случайно-равномерно выбрать ребро e = (u, v) of G
      Пусть G' – граф, полученный в результате стягивания e,
         в котором новый узел z_uv заменяет u и v
      Определить S(z_uv) = S(u)∪S(v)
      Рекурсивно применить алгоритм стягивания к G'
   Конец Если
```

Анализ алгоритма

Алгоритм принимает случайные решения, поэтому существует некоторая вероятность того, что глобальный минимальный разрез будет обнаружен, и некоторая вероятность того, что он не будет обнаружен. На первый взгляд каждется, что вероятность успеха экспоненциально мала. В конце концов, существует экспоненциальное количество возможных разрезов G; почему минимальному разрезу должно отдаваться предпочтение? Но сначала мы покажем, что вероятность успеха только полиномиально мала. Из этого следует, что если выполнить алгоритм полиномиальное количество раз и вернуть лучший разрез, найденный по всем выполнениям, можно получить глобальный минимальный разрез с высокой вероятностью.

(13.5) Алгоритм стягивания возвращает глобальный минимальный разрез графа G с вероятностью не менее $1/\binom{n}{2}$.

Доказательство. Возьмем глобальный минимальный разрез (A, B) графа G и предположим, что он имеет размер k; иначе говоря, существует множество Φ из k ребер, один конец которых принадлежит A, а другой принадлежит B. Мы хотим предоставить нижнюю границу вероятности того, что алгоритм стягивания вернет разрез (A, B).

Что может пойти не так на первом шаге алгоритма стягивания? Проблема возникнет при стягивании ребра из Φ. В этом случае узел из A и узел из B будут слиты в один суперузел и разрез (A, B) уже не может быть получен на выходе алгоритма.

И наоборот, если стягивается ребро, не входящее в Ф, то остается вероятность того, что разрез (A, B) будет возвращен успешно.

Итак, нам нужна верхняя граница вероятности того, что стягивается ребро из Ф, а для этого потребуется нижняя граница размера E. Обратите внимание: если любой узел v имеет степень ниже k, то разрез $(\{v\}, V-\{v\})$ будет иметь размер меньше k, а это противоречит нашему предположению о том, что (A, B) является глобальным минимальным разрезом. Тогда степень каждого узла в G не менее k, а значит, $[E] \geq \frac{1}{2}kn$. Следовательно, вероятность того, что стягивается ребро из Ф, не превышает

$$\frac{k}{\frac{1}{2}kn} = \frac{2}{n}.$$

Теперь рассмотрим ситуацию после j итераций, когда текущий граф G' содержит $n - j$ суперузлов, и предположим, что ни одно ребро в Ф еще не стягивалось. Каждый разрез G' является разрезом G, поэтому каждому суперузлу G' инцидентны не менее k ребер. Следовательно, G' содержит не менее $\frac{1}{2}k(n-j)$ ребер, а вероятность того, что ребро из Ф будет стянуто на следующей итерации $j + 1$, не превышает

$$\frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}.$$

Разрез (A,B) будет возвращен алгоритмом, если ни одно ребро из Ф не было стянуто на любой из итераций 1,2, ..., $n - 2$. Если взять за E_j событие «ребро из Ф не было стянуто на итерации j», то мы показали, что $\Pr[\mathcal{E}_1] \geq 1-2/n$ и $\Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_j] \geq 1-2(n-j)$. Нас интересует нижняя граница величины $\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_{n-2}]$, и «раскрутка» формулы условной вероятности позволяет убедиться в том, что она равна

$$\begin{aligned}
&\Pr[\mathcal{E}_1] \cdot \Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \cdots \Pr[\mathcal{E}_{j+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_j] \cdots \Pr[\mathcal{E}_{n-2} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cdots \cap \mathcal{E}_{n-3}] \\
&\geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right) \cdots \left(1-\frac{2}{n-j}\right) \cdots \left(1-\frac{2}{3}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \cdots \left(\frac{2}{4}\right)\left(\frac{1}{3}\right) = \frac{2}{n(n-1)} = \binom{n}{2}^{-1}.
\end{aligned}$$

■

Итак, теперь мы знаем, что одно выполнение алгоритма стягивания не найдет глобальный минимальный разрез с вероятностью не более $\left(1-1/\binom{n}{2}\right)$. Конечно, это

число очень близко к 1, но вероятность успеха можно усилить простым повторением алгоритма с независимым случайным решением и выбором лучшего из найденных разрезов. Согласно (13.1), если выполнить алгоритм $\binom{n}{2}$ раз, то вероятность того, что глобальный минимальный разрез не будет найден ни в одном выполнении, не превышает

$$\left(1-1/\binom{n}{2}\right)^{\binom{n}{2}} \le \frac{1}{e}.$$

Дальнейшие повторения позволяют легко опустить вероятность неудачи ниже $1/e$: если повторить алгоритм $\binom{n}{2}\ln n$ раз, то вероятность того, что глобальный минимальный разрез не будет найден, не превысит $e^{-\ln n} = 1/n$.

Общее время выполнения, необходимое для обеспечения высокой вероятности успеха, полиномиально по n, так как каждое выполнение алгоритма стягивания занимает полиномиальное время и алгоритм выполняется полиномиальное количество раз. Его время выполнения относительно велико по сравнению с методами нахождения потока, так как $\Theta(n^2)$ независимых запусков выполняются не менее $\Omega(n^2)$ раз. Мы решили описать эту версию алгоритма стягивания как самую простую и элегантную; было показано, что некоторые оптимизации организации многократных запусков способны существенно улучшить время выполнения.

Дальнейший анализ: количество глобальных минимальных разрезов

Анализ алгоритма стягивания дает на удивление простой ответ на следующий вопрос: имеется ненаправленный граф $G = (V, E)$ с n узлами, какое максимальное количество глобальных минимальных разрезов он может иметь (как функции от n)?

Очевидно, что для направленной потоковой сети количество минимальных разрезов s–t может быть экспоненциальным по n. Например, возьмем направленный граф с узлами s, t, v_1, v_2, ..., v_n и ребрами с единичной пропускной способностью (s,v_i) и (v_i,t) для каждого i. Тогда s в сочетании с любым подмножеством $\{v_1, v_2, ..., v_n\}$ образует сторону источника в минимальном разрезе, а следовательно, всего существуют 2^n минимальных разрезов s–t.

Но в том, что касается глобальных минимальных разрезов в ненаправленном графе, ситуация выглядит совершенно иначе. Немного поэкспериментировав на примерах, вы убедитесь в том, что цикл из n узлов имеет $\binom{n}{2}$ глобальных минимальных разрезов (полученных разрезанием любых двух ребер), и на первый взгляд неясно, как построить ненаправленный граф с большим количеством.

Сейчас мы покажем, что анализ алгоритма стягивания немедленно дает ответ на этот вопрос и доказывает, что цикл из n узлов в действительности является крайним случаем.

(13.6) Ненаправленный граф $G = (V,E)$ из n узлов содержит не более $\binom{n}{2}$ глобальных минимальных разрезов.

Доказательство. В доказательстве (13.5) установлено даже больше, чем заявлено в утверждении. Допустим, имеется граф G, а C_1, ..., C_r — все его глобальные минимальные разрезы. Пусть E_i обозначает событие «C_i возвращается алгоритмом стягивания», а $\mathcal{E} = \bigcup_{i=1}^{r} \mathcal{E}_i$ — событие «алгоритм возвращает любой глобальный минимальный разрез».

Тогда, хотя в (13.5) просто утверждается, что $\Pr[\mathcal{E}] \geq 1/\binom{n}{2}$, доказательство фактически демонстрирует, что для всех i выполняется $\Pr[\mathcal{E}] \geq 1/\binom{n}{2}$. События в паре $\mathcal{E}_i$ и $\mathcal{E}_j$ являются взаимоисключающими (так как при любом выполнении алгоритма возвращается только один разрез), поэтому, согласно границе объединения для взаимоисключающих событий (13.49), имеем

$$\Pr[\mathcal{E}] = \Pr\left[\bigcup_{i=1}^{r} \mathcal{E}_i\right] = \sum_{i=1}^{r} \Pr[\mathcal{E}_i] \geq r/\binom{n}{2}.$$

Но очевидно, что $\Pr[\mathcal{E}] \leq 1$, а следовательно, должно выполняться $r \leq \binom{n}{2}$. ■

13.3. Случайные переменные и ожидания

До настоящего момента наш анализ рандомизированных алгоритмов и процессов базировался на выявлении некоторых «плохих событий» и ограничении их вероятности. Такой анализ относится к *качественному* типу в том смысле, что алгоритм либо выполняется успешно, либо нет. *Количественный* анализ учтет некоторые характеристики, связанные с поведением алгоритма (например, его время выполнения, или качество предоставляемых решений), и постарается определить *ожидаемые* размеры этих характеристик для случайных решений, принимаемых алгоритмом. Чтобы такой анализ стал возможен, необходимо ввести фундаментальное понятие *случайной переменной*.

Для заданного вероятностного пространства случайной переменной X называется функция из пространства выборки в пространство натуральных чисел, такая, что для любого натурального числа j получение значения j множеством $X^{-1}(j)$ всех точек выборки является событием. Конструкция $Pr[X = j]$ может использоваться как неформальное сокращение для $Pr[X^{-1}(j)]$; фактически мы задаем вопрос

о вероятности того, что X примет заданное значение, которое рассматривается как значение «случайной переменной».

Для заданной случайной переменной X часто представляет интерес ее *ожидание* — «среднее значение», принимаемое X. Определим его по формуле

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j],$$

в предположении, что при расхождении суммы оно принимает значение ∞. Итак, например, если X принимает каждое из значений $\{1, 2, ..., n\}$ с вероятностью $1/n$, то

$$E[X] = 1(1/n) + 2(1/n) + ... + n(1/n) = \binom{n+1}{2} / n = (n+1)/2.$$

Пример: ожидание первого успеха

В следующем примере правильно выбранная случайная переменная позволяет оценить некоторый аналог «времени выполнения» простого случайного процесса. Допустим, вы бросаете монетку, на которой с вероятностью $p > 0$ выпадает «орел», а с вероятностью $1 - p$ выпадает «решка». Результаты разных бросков монетки независимы. Если бросать монетку до тех пор, пока не выпадет «орел», какое ожидаемое число бросков придется выполнить? Чтобы ответить на этот вопрос, обозначим X случайную переменную, равную количеству выполненных бросков. Для $j > 0$ имеем $Pr[X = j] = (1 - p)^{j-1}p$: чтобы этот процесс состоял ровно из j шагов, на первых $j - 1$ бросках должна выпасть «решка», а на j-м броске должен выпасть «орел». Применяя определение, получаем

$$E[X] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j] = \sum_{j=1}^{\infty} j(1-p)^{j-1} p = \frac{p}{1-p} \sum_{j=1}^{\infty} j(1-p)^{j} = \frac{p}{1-p} \cdot \frac{(1-p)}{p^2} = \frac{1}{p}.$$

Таким образом, мы получили следующий интуитивно понятный результат.

(13.7) При многократном выполнении независимых испытаний в эксперимент, каждое из которых завершается успешно с вероятностью $p > 0$, ожидаемое количество испытаний, которые должны быть выполнены до первого успеха, составляет $1/p$.

Линейность ожидания

В разделах 13.1 и 13.2 мы разбивали события на объединения более простых событий и работали с вероятностями этих более простых событий. Этот прием также эффективно работает со случайными переменными и базируется на принципе *линейности ожидания*.

(13.8) (Линейность ожидания). Для двух случайных переменных X и Y, определенных в одном вероятностном пространстве, можно определить $X + Y$ как слу-

чайную переменную, равную $X(\omega) + Y(\omega)$ для точки выборки ω. Для любых X и Y выполняется $E[X + Y] = E[X] + E[Y]$.

Доказательство не приводится, оно достаточно простое. Мощь (13.8) в основном обусловлена тем, что утверждение относится к сумме *любых* случайных переменных; никакие ограничивающие предположения не нужны. В результате, если потребуется вычислить ожидание сложной случайной переменной X, мы можем сначала записать ее как сумму более простых случайных переменных $X = X_1 + X_2 + ... + X_n$, вычислить каждое $E[X_i]$, а затем определить $E[X] = \sum E[X_i]$. Рассмотрим несколько примеров практического применения этого принципа.

Пример: угадывание карт

Чтобы поразить ваших друзей, вы предлагаете им перетасовать колоду из 52 карт, а затем открывать по одной карте. Вы пытаетесь угадать, какая карта будет открыта следующей. К сожалению, ваши способности к ясновидению оставляют желать лучшего, а память не позволяет точно запомнить уже открытые карты, поэтому ваша стратегия заключается в простом случайном равномерном выборе карты из полной колоды. Сколько попыток окажутся успешными?

Рассмотрим более общую формулировку задачи: колода состоит из n разных карт, а X — случайная переменная, равная количеству правильных предсказаний. Существует неожиданно простой способ вычисления X: случайная переменная X_i (для $i = 1, 2, ..., n$) определяется равной 1, если i-е предсказание было правильным, и 0 в противном случае. При этом $X = X_1 + X_2 + ... + X_n$, и

$$E[X_i] = 0 \cdot \Pr[X_i = 0] + 1 \cdot \Pr[X_i = 1] = \Pr[X_i = 1] = \frac{1}{n}.$$

Стоит отметить полезный факт, который неявно следует из предыдущих вычислений: если Z — случайная переменная, принимающая только значения 0 и 1, то $E[Z] = \Pr[Z = 1]$.

Так как $E[X_i] = \frac{1}{n}$ для всех i, получаем

$$E[X] = \sum_{i=1}^{n} E[X_i] = n\left(\frac{1}{n}\right) = 1.$$

Итак, нам удалось доказать следующее:

(13.9) Ожидаемое количество правильных предсказаний в стратегии без запоминания равно 1 независимо от n.

Попытка вычислить $E[X]$ непосредственно из определения $\sum_{j=0}^{\infty} j \cdot \Pr[X = j]$ создаст куда больше проблем, потому что это потребует гораздо более сложного суммирования. За внешне безобидным утверждением (13.8) скрывается значительный уровень сложности.

Угадывание с запоминанием

Возьмем второй сценарий: с ясновидением ситуация не изменилась, но вы научились запоминать, какие карты уже были открыты. Следовательно, при предсказании следующей карты равномерный случайный выбор ограничивается только теми картами, *которые еще не открывались*. На сколько правильных предсказаний можно рассчитывать при такой стратегии?

И снова случайная переменная X_i принимает значение 1, если i-е предсказание верно, и 0 в противном случае. Чтобы i-е предсказание было правильным, необходимо угадать карту только из $n - i + 1$ оставшихся карт; следовательно,

$$E[X_i] = \Pr[X_i = 1] = \frac{1}{n-i+1}$$

и приходим к

$$\Pr[X] = \sum_{i=1}^{n} E[X_j] = \sum_{i=1}^{n} \frac{1}{n-i+1} = \sum_{i=1}^{n} \frac{1}{i}.$$

Последнее выражение $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ называется *гармоническим числом* $H(n)$; эти числа уже неоднократно встречались в двух предыдущих главах. В частности, в главе 11 было показано, что $H(n)$ как функция n близко воспроизводит значение $\int_1^{n+1} \frac{1}{x} dx = \ln(n+1)$. Для наших целей базовая граница $H(n)$ будет переформулирована следующим образом.

(13.10) $\ln(n + 1) < H(n) < 1 + \ln n$, или менее формально, $H(n) = \Theta(\log n)$.

Итак, когда вы получаете возможность запоминать уже открытые карты, ожидаемое количество правильных предсказаний значительно превышает 1.

(13.11) Ожидаемое число правильных предсказаний в стратегии предсказания с запоминанием составляет $H(n) = \Theta(\log n)$.

Пример: сбор купонов

Прежде чем переходить к более сложным примерам, рассмотрим еще один тривиальный пример, в котором линейность ожиданий приносит существенную пользу. Предположим, фирма–производитель овсяных хлопьев кладет в каждую коробку купон. Всего существуют n разных видов купонов. Сколько коробок придется купить, чтобы собрать купоны всех типов?

Очевидно, понадобится не менее n коробок; но будет весьма удивительно, если после покупки n коробок будут собраны все n типов купонов. По мере того как в коллекции будет появляться все больше разных типов, вероятность обнаружить отсутствующий купон становится все ниже. После того, как у вас появятся $n - 1$ из

n видов купонов вероятность того, что новая коробка содержит купон отсутствующего типа, равна всего $1/n$.

Следующий метод позволяет точно определить ожидаемое время. Пусть X — случайная переменная, равная количеству коробок, купленных до первого обнаружения купона каждого типа. Как и в наших предыдущих примерах, эта случайная переменная выражается достаточно сложно, и ее было бы удобнее записать в виде суммы более простых случайных переменных. Для этого рассмотрим следующую естественную идею: процесс сбора купонов продвигается вперед каждый раз, когда вы покупаете коробку с купоном, которого у вас еще нет. Таким образом, основная цель процесса — обеспечить продвижение n раз. Какова вероятность того, что в заданный момент времени вы добьетесь прогресса на следующем шаге? Это зависит от того, сколько разных типов купонов у вас уже есть. Если у вас уже есть j типов, то вероятность продвижения на следующем шаге составляет $(n - j)/n$: из n типов купонов $n - j$ позволяют продвинуться вперед. Так как вероятность зависит от того, сколько разных типов купонов у вас уже есть, отсюда следует естественная идея разбиения X на более простые случайные переменные.

Предположим, процесс сбора купонов находится в фазе j, вы уже собрали купоны j разных типов и ждете возможности получить новый тип. При обнаружении нового типа купона завершается фаза j и начинается фаза $j + 1$. Таким образом, процесс начинается в фазе 0 и завершается в фазе $n - 1$. Пусть X_j — случайная переменная, равная количеству шагов, выполненных в фазе j. Тогда $X = X_0 + X_1 + ... + X_{n-1}$, а значит, достаточно найти $E[X_j]$ для каждого j.

(13.12) $E[Xj] = n/(n - j)$.

Доказательство. В каждом шаге фазы j фаза завершается немедленно в том и только в том случае, если следующий купон принадлежит к числу $n - j$ типов, которые еще не встречались ранее. Таким образом, в фазе j в действительности ожидается событие с вероятностью $(n - j)/n$, и, согласно (13.7), ожидаемая длина фазы j равна $E[X_j] = n/(n - j)$. ■

Используя эту формулу, получаем общее ожидаемое время.

(13.13) Ожидаемое время до сбора всех n типов купонов составляет $E[X] = nH(n) = \Theta(n \log n)$.

Доказательство. Из линейности ожидания получаем

$$E[X] = \sum_{j=0}^{n-1} E\left[X_j\right] = \sum_{j=0}^{n-1} \frac{n}{n-j} = n\sum_{j=0}^{n-1} \frac{1}{n-j} = n\sum_{i=1}^{n} \frac{1}{i} = nH(n).$$

Из (13.10) следует, что эта величина асимптотически равна $\Theta(n \log n)$. ■

Динамику процесса интересно сравнить с нашим интуитивным представлением о ней. После того, как будут собраны $n - 1$ из n типов купонов, вы ожидаете, что до появления последнего типа придется купить еще n коробок хлопьев. Вам снова и снова попадаются купоны, которые у вас уже есть, и начинает казаться, что последний тип «самый редкий». Но на самом деле он встречается с такой же частотой, как и все остальные; просто на поиск последнего типа купона, каким бы он ни был, потребуется много времени.

Последнее определение: условное ожидание

Перейдем к последнему, очень полезному понятию, которое связано со случайными переменными и встречается при последующем анализе. По аналогии с тем, как мы определяем условную вероятность одного события при условии наступления другого, можно определить условное ожидание случайной переменной при условии наступления некоторого события. Предположим, имеется случайная переменная X и событие E с положительной вероятностью. *Условное ожидание* X для заданного E определяется как ожидаемое значение X, вычисленное только по части пространства выборки, соответствующей E. Обозначим эту величину $E[X \mid E]$. Для этого достаточно заменить вероятности $\Pr[X = j]$ в определении ожидания условными вероятностями:

$$E[X \mid \mathcal{E}] = \sum_{j=0}^{\infty} j \cdot \Pr[X = j \mid \mathcal{E}].$$

13.4. Рандомизированный аппроксимирующий алгоритм для задачи MAX 3-SAT

В предыдущем разделе были представлены некоторые способы использования линейности ожидания для анализа рандомизированного процесса. Посмотрим, как применить эту идею при разработке аппроксимирующего алгоритма. Задача будет рассматриваться на примере задачи 3-SAT; вы увидите, что одним из следствий рандомизированного аппроксимирующего алгоритма является неожиданно сильное общее утверждение по поводу 3-SAT, которое на первый взгляд не имеет никакого отношения ни к алгоритмам, ни к рандомизации.

Задача

При изучении *NP*-полноты одной из основных задач была задача 3-SAT: для заданного множества условий $C_1, \ldots, C_k$, каждое из которых имеет длину 3, по множеству переменных $X = \{x_1, \ldots, x_n\}$, существует ли выполняющее логическое присваивание?

Например, такая задача может встретиться в системе для проверки истинности или ложности системы утверждений об окружающем мире (переменные $\{x_i\}$) по нескольким условиям, связывающим их друг с другом (условия $\{C_j\}$). Но окружающий мир — весьма противоречивое место, и если наша система накопит достаточно информации, множество условий может не иметь выполняющих присваиваний. И что тогда?

Если найти логическое присваивание, выполняющее все условия, не удается, будет естественно преобразовать экземпляр 3-SAT в задачу оптимизации: для

заданного множества входных условий C_1, ..., C_k найти логическое присваивание, выполняющее *как можно больше* условий. Назовем эту задачу *максимальной задачей* 3-*SAT* (или сокращенно *MAX* 3-*SAT*). Конечно, это *NP*-сложная задача оптимизации, так как принятие решения о том, равно ли максимальное количество одновременно выполняемых условий *k*, является *NP*-сложным. Посмотрим, что можно сказать об аппроксимирующих алгоритмах с полиномиальным временем.

Разработка и анализ алгоритма

Неожиданно простой рандомизированный алгоритм дает сильные гарантии эффективности для этой задачи. Допустим, каждой переменной x_1, ..., x_n независимо присваивается 0 или 1 (каждое значение имеет вероятность $\frac{1}{2}$). Какое ожидаемое число условий будет выполнено таким случайным присваиванием?

Обозначим *Z* случайную переменную, равную количеству выполненных условий. Как и в разделе 13.3, разложим *Z* на сумму случайных переменных, каждая из которых принимает значение 0 или 1, а именно $Z_i = 1$, если условие C_i выполнено, или 0 в противном случае. Следовательно, $Z = Z_1 + Z_2 + ... + Z_k$. Теперь значение $E[Z_i]$ равно вероятности того, что условие C_i выполнено, а эта вероятность легко вычисляется: чтобы условие C_i не было выполнено, каждой из его трех переменных должно быть присвоено значение, которое мешает его истинности; так как переменные присваиваются независимо, вероятность равна $\left(\frac{1}{2}\right)^3 = \frac{1}{8}$. Следовательно, условие C_i выполняется с вероятностью $1-\frac{1}{8}=\frac{7}{8}$, и $E[Z_i]=\frac{7}{8}$.

Используя линейность ожидания, мы видим, что ожидаемое количество выполненных условий равно $E[Z]=E[Z_1]+E[Z_2]+...+E[Z_k]=\frac{7}{8}k$. Так как ни одно присваивание не может выполнить более *k* условий, мы приходим к следующей гарантии.

(13.14) Возьмем формулу 3-SAT, в которой каждое условие содержит три разные переменные. Ожидаемое количество условий, выполненных случайным присваиванием, лежит в пределах множителя $\frac{7}{8}$ от оптимума.

Но если разобраться, что же в действительности произошло в этом (откровенно говоря, простом) анализе случайного присваивания, становится ясно, что можно привести и более сильное утверждение. Для любой случайной переменной должна существовать некоторая точка, в которой она принимает значение, по крайней мере большее своего ожидания. Мы показали, что для любого экземпляра 3-SAT случайное логическое присваивание в ожидании выполняет $\frac{7}{8}$ всех условий;

следовательно, должно *существовать* логическое присваивание, выполняющее количество условий, по крайней мере не меньшее ожидания.

(13.15) Для каждого экземпляра 3-SAT существует логическое присваивание, выполняющее по крайней мере $\frac{7}{8}$ всех условий.

В утверждении (13.15) есть нечто по-настоящему удивительное. Мы прибыли к неочевидному факту, касающемуся 3-SAT, — существованию присваивания, выполняющего много условий, — в формулировке которого нет ничего связанного с рандомизацией; тем не менее мы пришли к нему посредством рандомизированного построения. Кроме того, рандомизированное построение предоставляет, пожалуй, простейшее доказательство (13.15). Этот принцип довольно часто встречается в комбинаторике: чтобы продемонстрировать возможность существования некоторой структуры, мы показываете, что случайное построение приводит к ней с положительной вероятностью. Подобные построения создаются в результате применения *вероятностного метода*.

Любопытное, хотя и второстепенное следствие (13.15): каждый экземпляр 3-SAT, содержащий не более 7 условий, является выполнимым. Почему? Если экземпляр содержит $k \leq 7$ условий, то из (13.15) следует, что существует присваивание, выполняющее не менее $\frac{7}{8}k$ из них. Но если $k \leq 7$, то $\frac{7}{8}k > k-1$; а поскольку количество условий, выполняемых присваиванием. должно быть целым, оно должно быть равно k, иначе говоря, выполняются все условия.

Дальнейший анализ: поиск хорошего присваивания

Предположим, нас не устраивает алгоритм, который выдает одно присваивание с ожидаемым большим количеством выполненных условий. Вместо этого требуется рандомизированный алгоритм, который имеет полиномиальное ожидаемое время выполнения и гарантированно выдает логическое присваивание, выполняющее как минимум $\frac{7}{8}$ всех условий.

Простейшее решение заключается в генерировании случайных логических присваиваний до тех пор, пока одно из них не выполнит по крайней мере $\frac{7}{8}k$ условий. Из (13.15) известно, что такое присваивание существует; но сколько времени потребуется для того, чтобы найти его методом случайных испытаний?

Здесь будет естественно применить границу времени ожидания, полученную в (13.7). Если мы сможем показать, что вероятность того, что случайное присваивание удовлетворяет по крайней мере $\frac{7}{8}k$ условий, не менее p, то ожидаемое ко-

личество испытаний, выполняемых алгоритмом, составляет $1/p$. Итак, нам хотелось бы показать, что величина p по крайней мере не более чем обратно-полиномиальна по n и k.

Для $j = 0, 1, 2, ..., k$ пусть p_j обозначает вероятность того, что случайное присваивание выполняет ровно j условий. Следовательно, ожидаемое количество выполненных условий по определению ожидания составляет $\sum_{j=0}^{k} jp_j$; из предшествующего анализа оно равно $\frac{7}{8}k$. Нас интересует величина $p = \sum_{j \geq 7k/8} p_j$. Как использовать нижнюю границу ожидаемого количества для получения нижней границы этой величины?

Начнем с равенства

$$\frac{7}{8}k = \sum_{j=0}^{k} jp_j = \sum_{j<7k/8} jp_j + \sum_{j \geq 7k/8} jp_j .$$

Обозначим k' наибольшее натуральное число, строго меньшее $\frac{7}{8}k$. Если заменить слагаемые в первой сумме на $k'p_j$, а слагаемые во второй сумме на kp_j, правая сторона приведенного равенства только возрастает. Также заметим, что $\sum_{j<7k/8} p_j = 1 - p$, а следовательно,

$$\frac{7}{8}k \leq \sum_{j<7k/8} k'p_j + \sum_{j \geq 7k/8} kp_j = k'(1-p) + kp \leq k' + kp .$$

Отсюда $kp \geq \frac{7}{8}k - k'$. Но $\frac{7}{8}k - k' \geq \frac{1}{8}$, так как k' — натуральное число, строго меньшее $\frac{7}{8}$ другого натурального числа, а следовательно,

$$p \geq \frac{\frac{7}{8}k - k'}{k} \geq \frac{1}{8k} .$$

Мы получили желаемое — нижнюю границу для p. С учетом границы времени ожидания (13.7) мы видим, что ожидаемое число испытаний, необходимых для нахождения выполняющего присваивания, не превышает $8k$.

(13.16) Существует рандомизированный алгоритм с полиномиальным ожиданием времени выполнения, который гарантированно выдает логическое присваивание, выполняющее по крайней мере $\frac{7}{8}$ всех условий.

13.5. Рандомизация принципа «разделяй и властвуй»: нахождение медианы и быстрая сортировка

Применение парадигмы «разделяй и властвуй» при разработке алгоритмов уже рассматривалось в одной из предыдущих глав. Принцип «разделяй и властвуй» хорошо работает в сочетании с рандомизацией; чтобы доказать это, мы приведем алгоритмы «разделяй и властвуй» для двух фундаментальных задач: вычисления медианы n чисел и сортировки. В каждом случае шаг «разделяй» выполняется с применением рандомизации, после чего ожидаемые значения случайных переменных используются для анализа времени, затраченного на рекурсивные вызовы.

Задача: нахождение медианы

Предположим, имеется множество из n чисел $S = \{a_1, a_2, \ldots a_n\}$. *Медианой* этого множества называется число, которое займет среднюю позицию в результате сортировки. При четном n появляется неприятное техническое осложнение, так как «средней позиции» не существует, поэтому формальное определение выглядит так: медиана $S = \{a_1, a_2, \ldots, a_n\}$ равна k-му по величине элементу S, где $k = (n + 1)/2$ при нечетном n и $k = n/2$ при четном n. В дальнейшем для простоты будем считать, что все числа различны. Без этого предположения формулировка задачи усложняется, но ничего принципиально нового в решении не появляется.

Очевидно, медиана легко вычисляется за время $O(n \log n)$, если числа будут предварительно отсортированы. Но если задуматься, становится совершенно не очевидно, почему для задачи нахождения медианы *необходима* сортировка, — как и то, почему для нее необходимо время $\Omega(n \log n)$. Мы покажем, что простой рандомизированный алгоритм, основанный на принципе «разделяй и властвуй», — ожидаемое время выполнения $O(n)$.

Разработка алгоритма

Обобщенный алгоритм с разделителями

Первым ключевым шагом к ожидаемому линейному времени выполнения станет переход от нахождения медианы к более общей задаче выбора. Для заданного множества S, состоящего из n чисел, и числа k от 1 до n рассмотрим функцию `Select`(S, k), которая возвращает k-й по величине элемент S. Частными случаями `Select` являются задачи нахождения медианы S через `Select`(S, $n/2$) или `Select`(S, $(n + 1)/2$); она также включает более простые задачи нахождения минимума (`Select`(S, 1)) и максимума (`Select`(S, n)). Требуется разработать алгоритм, реализующий `Select` с выполнением за ожидаемое время $O(n)$.

Базовая структура алгоритма, реализующего Select, выглядит так.

Мы берем элемент $a_i \in S$ — «разделитель» — и формируем множества $S^- = \{a_j : a_j < a_i\}$ и $S^+ = \{a_j : a_j > a_i\}$. Затем мы определяем, какое из множеств S^- или S^+ — содержит k-й по величине элемент, и выполняем итерацию только по нему. Не уточняя, как мы планируем выбирать разделитель, приведем более конкретное описание того, как формируются множества и проводятся итерации.

Select(S, k):
Выбрать разделитель $a_i \in S$
Для каждого элемента a_j множества S
  Включить a_j в S^-, если $a_j < a_i$
  Включить a_j в S^+, если $a_j > a_i$
Конец цикла
Если $|S^-| = k - 1$
  Разделитель a_i являлся искомым ответом
Иначе Если $|S^-| \geq k$
  k-й по величине элемент принадлежит S^-
  Рекурсивно вызвать *Select*(S^-, k)
Иначе $|S^-| = \ell < k - 1$
  k-й по величине элемент принадлежит S^+
  Рекурсивно вызвать *Select*$(S^+, k - 1 - \ell)$
Конец Если

Заметим, что алгоритм всегда рекурсивно вызывается для строго меньшего множества, поэтому он должен завершиться. Также заметим, что если $|S| = 1$, то и $k = 1$; единственный элемент S будет возвращен алгоритмом. Наконец, из выбора рекурсивного вызова по индукции очевидно, что правильный ответ также будет возвращен при $|S| > 1$. Мы приходим к следующему выводу:

(13.17) Независимо от выбора разделителя описанный алгоритм возвращает k-й по величине элемент S.

Выбор хорошего разделителя

Теперь разберемся, как время выполнения Select зависит от способа выполнения разделителя. Если предположить, что разделитель выбирается за линейное время, то остаток алгоритма выполняется за линейное время плюс время рекурсивного вызова. Но как выбор разделителя влияет на время выполнения рекурсивного вызова? На самом деле важно, что разделитель значительно сокращает размер рассматриваемого множества, чтобы нам не приходилось многократно перебирать большие множества чисел. Хороший выбор разделителя должен порождать множества S^- и S^+ приблизительно равного размера. Например, если бы мы могли всегда выбирать медиану в качестве разделителя, то для времени выполнения можно было бы установить линейную границу. Пусть cn — время выполнения Select, не считая времени рекурсивного вызова. Тогда, при выборе медиан в качестве разделителей, время выполнения $T(n)$ будет ограничиваться рекуррентным

отношением $T(n) \le T(n/2) + cn$. Это рекуррентное отношение уже встречалось нам в начале главы 5, где было показано, что у него имеется решение $T(n) = O(n)$.

Конечно, стремление использовать медиану в качестве разделителя выглядит своего рода «порочным кругом» — ведь именно медиану мы и хотели вычислить! Но на самом деле понятно, что хорошим разделителем может послужить любой «более или менее центральный» элемент: если бы мы могли выбрать медиану так, чтобы в множестве присутствовали как минимум εn элементов, больших и меньших a_i для любой фиксированной константы $\varepsilon > 0$, то размер множеств в рекурсивном вызове каждый раз будет уменьшаться как минимум с множителем $(1 - \varepsilon)$. Таким образом, время выполнения $T(n)$ будет ограничиваться рекуррентным отношением $T(n) \le T((1 - \varepsilon)n) + cn$. Здесь можно воспользоваться тем же аргументом, который показал, что предыдущее рекуррентное отношение имеет решение $T(n) = O(n)$: развертывая это рекуррентное отношение для любого $\varepsilon > 0$, получаем

$$T(n) \le cn + (1-\varepsilon)cn + (1-\varepsilon)^2 cn + \cdots = \left[1 + (1-\varepsilon) + (1-\varepsilon)^2 + \cdots\right]$$

$$cn \le \frac{1}{\varepsilon} cn,$$

так как мы имеем дело со сходящейся геометрической прогрессией.

Единственное, чего следует действительно остерегаться, — это разделитель, «смещенный от центра». Например, если всегда выбирать в качестве разделителя минимальный элемент, то множество, используемое для рекурсивного вызова, может содержать всего на один элемент меньше предыдущего. В этом случае время выполнения $T(n)$ будет ограничено рекуррентным отношением $T(n) \le T(n - 1) + cn$. При раскрутке этого рекуррентного отношения возникает проблема:

$$T(n) \le cn + c(n-1) + c(n-2) + \cdots = \frac{cn(n+1)}{2} = \Theta(n^2).$$

Случайные разделители

Выбор разделителя «из центра» (в только что определенном смысле) безусловно напоминает исходную задачу выбора медианы; но на самом деле все не так плохо, так как подойдет *любой* разделитель «из центра».

По этой причине еще не определенный шаг выбора разделителя будет реализован по следующему простому правилу:

Выбрать разделитель $a_i \in S$ случайно с равномерным распределением.

Правило основано на очень естественном представлении: так как достаточно большая часть элементов находится «недалеко от центра», случайный выбор с большой вероятностью обеспечит хорошее разбиение.

Эта идея заложена в основу анализа времени выполнения со случайным разделителем; ожидается, что размер рассматриваемого множества будет уменьшаться с фиксированным постоянным коэффициентом при каждой итерации, поэтому мы должны получить сходящуюся прогрессию, а следовательно, линейную границу, как в предыдущем случае.

Анализ алгоритма

Будем считать, что алгоритм находится *в фазе j*, когда размер рассматриваемого множества не превышает $n\left(\frac{3}{4}\right)^j$, но при этом он больше $n\left(\frac{3}{4}\right)^{j+1}$. Попробуем найти границу для ожидаемого времени, проводимого алгоритмом в фазе *j*. В заданной итерации алгоритма элемент рассматриваемого множества называется *центральным*, если по крайней мере четверть элементов меньше и по крайней мере четверть элементов больше него.

Теперь заметим, что если в качестве разделителя выбирается центральный элемент, то по крайней мере четверть множества будет отброшена, множество уменьшится с коэффициентом $\frac{3}{4}$ или лучше, и текущая фаза подойдет к концу. Кроме того, к категории центральных относится половина всех элементов в множестве, поэтому вероятность того, что случайный выбор разделителя даст центральный элемент, равна $\frac{1}{2}$. Следовательно, из простой границы времени ожидания (13.7) ожидаемое количество итераций перед обнаружением центрального элемента равно 2; а значит, ожидаемое количество итераций, проведенных в фазе *j*, для любого *j* не превышает 2.

Собственно, это все, что необходимо для анализа. Пусть X — случайная переменная, равная количеству шагов, предпринятых алгоритмом. Ее можно записать в виде суммы $X = X_0 + X_1 + X_2 + ...$, где X_j — ожидаемое количество шагов, проведенных алгоритмом в фазе *j*. Когда алгоритм находится в фазе *j*, размер множества не превышает $n\left(\frac{3}{4}\right)^j$, поэтому количество шагов, необходимых для одной итерации в фазе *j*, не превышает $cn\left(\frac{3}{4}\right)^j$ для некоторой константы *c*. Мы только что обосновали, что ожидаемое количество итераций, проводимых в фазе *j*, не превышает 2, а следовательно, $E[X_j] \le 2cn\left(\frac{3}{4}\right)^j$. Это позволяет ограничить общее ожидаемое время выполнения с использованием линейности ожидания:

$$E[X] = \sum_j E[X_j] \le \sum_j 2cn\left(\frac{3}{4}\right)^j = 2cn\sum_j \left(\frac{3}{4}\right)^j \le 8cn,$$

так как сумма $\sum_j \left(\frac{3}{4}\right)^j$ представляет собой сходящуюся геометрическую прогрессию. Приходим к следующему результату:

(13.18) Ожидаемое время выполнения `Select`(*n*,*k*) равно $O(n)$.

Второй пример: быстрая сортировка

Рандомизированный метод «разделяй и властвуй», который использовался для нахождения медианы, также заложен в основу *алгоритма быстрой сортировки*. Как и прежде, мы выбираем разделитель для входного множества S и разбиваем S по элементам со значениями ниже и выше разделителя. Различие в том, что вместо поиска медианы только с одной стороны от разделителя мы рекурсивно сортируем обе стороны и соединяем два отсортированных фрагмента (с промежуточным разделителем) для получения общего результата. Кроме того, необходимо явно включить базовый случай для рекурсивного кода: рекурсия используется только для множеств с размером 4 и более. Полное описание алгоритма быстрой сортировки приведено ниже.

```
Quicksort(S):
Если |S| ≤ 3
   Отсортировать S
   Вывести отсортированный список
Иначе
   Случайно выбрать разделитель a_i ∈ S с равномерным распределением
   Для каждого элемента a_j множества S
      Включить a_j в множество S⁻, если a_j < a_i
      Включить a_j в множество S⁺, если a_j > a_i
   Конец цикла
   Рекурсивно вызвать Quicksort(S⁻) и Quicksort(S⁺)
   Вывести отсортированное множество S⁻,
      затем a_i и отсортированное множество S⁺
Конец Если
```

Как и в случае с нахождением медианы, время выполнения этого метода в худшем случае не впечатляет. Если всегда выбирать в качестве разделителя наименьший элемент, но время выполнения $T(n)$ для n-элементных множеств определяется уже знакомым рекуррентным отношением: $T(n) \leq T(n-1) + cn$, так что в итоге приходим к границе $T(n) = \Theta(n^2)$ (худшее время быстрой сортировки из всех возможных).

С другой стороны, если при каждой итерации в качестве разделителя выбирается медиана каждого множества, то получается рекуррентное отношение $T(n) \leq 2T(n/2) + cn$, часто встречавшееся при анализе алгоритмов «разделяй и властвуй» главы 5; время выполнения в этом случае составляет $O(n \log n)$.

Сейчас нас интересует ожидаемое время выполнения; мы покажем, что оно может быть ограничено величиной $O(n \log n)$ — почти такой же, как в лучшем случае при выборе разделителей идеально по центру. Наш анализ быстрой сортировки достаточно близко воспроизводит анализ нахождения медианы. Как и в процедуре *Select*, использованной при поиске медианы, ключевую роль играет определение *центрального разделителя* — разделяющего множество так, чтобы каждая сторона содержала не менее четверти элементов. (Как упоминалось ранее, для анализа достаточно, чтобы каждая сторона содержала некоторую фиксированную часть элементов; четверть выбрана для удобства.) Идея заключается в том, что случайный

выбор с большой вероятностью приведет к центральному разделителю, а центральные разделители работают эффективно. В случае сортировки центральный разделитель разбивает задачу на две существенно меньшие подзадачи.

Чтобы упростить формулировку, мы слегка изменим алгоритм так, чтобы он выдавал рекурсивные вызовы только при обнаружении центрального разделителя. По сути, измененный алгоритм отличается от `Quicksort` тем, что он отбрасывает «нецентральные» разделители и делает повторную попытку; `Quicksort`, напротив, выдает рекурсивные вызовы даже с нецентральным разделителем. Дело в том, что ожидаемое время выполнения измененного алгоритма очень просто анализируется по прямой аналогии с нашим анализом нахождения медианы. Похожий, хотя и более сложный анализ также можно провести и для исходной реализации `Quicksort`; мы этот анализ рассматривать не будем.

```
Измененная версия Quicksort(S):
Если |S|≤3
   Отсортировать S
   Вывести отсортированный список
Конец Если
Иначе
   Пока не будет найден центральный разделитель
      Случайно выбрать разделитель a_i ∈ S с равномерным распределением
      Для каждого элемента a_j множества S
         Включить a_j в множество S⁻, если a_j < a_i
         Включить a_j в множество S⁺, если a_j > a_i
      Конец цикла
      Если |S⁻| ≥ |S|/4 и |S⁺| ≥ |S|/4
         a_i является центральным разделителем
      Конец Если
   Конец Пока
   Рекурсивно вызвать Quicksort(S⁻) и Quicksort(S⁺)
   Вывести отсортированное множество S⁻,
      затем a_i и отсортированное множество S⁺
Конец Если
```

Рассмотрим подзадачу для некоторого множества S. Каждая итерация цикла *Пока* выбирает возможный разделитель a_i и тратит время $O(|S|)$ на разбиение множества и принятие решения о том, является ли a_i центральным элементом. Ранее мы обосновали, что количество необходимых итераций до нахождения центрального разделителя не превышает 2. Таким образом, мы получаем следующее утверждение:

(13.19) Ожидаемое время выполнения алгоритма для множества S без учета времени рекурсивных вызовов составляет $O(|S|)$.

Алгоритм рекурсивно вызывается для нескольких подзадач, которые будут группироваться по размеру. Задача будет называться относящейся к типу j, если размер рассматриваемого множества не превышает $n\left(\frac{3}{4}\right)^j$, но при этом больше

$n\left(\frac{3}{4}\right)^{j+1}$. Согласно (13.19), ожидаемое время, потраченное на подзадачу типа j без учета рекурсивных вызовов, равно $O\left(n\left(\frac{3}{4}\right)^{j+1}\right)$. Чтобы ограничить общее время выполнения, необходимо найти границу количества подзадач каждого типа j. Разбиение подзадачи типа j по центральному разделителю создает две подзадачи более высокого типа. Таким образом, подзадачи заданного типа j не пересекаются, и мы получаем границу для количества подзадач.

(13.20) Количество подзадач типа j, созданных алгоритмом, не превышает $\left(\frac{3}{4}\right)^{j+1}$.

Всего существует не более $\left(\frac{3}{4}\right)^{j+1}$ подзадач типа j, и ожидаемое время, потраченное на каждую, равно $O\left(n\left(\frac{3}{4}\right)^{j}\right)$ согласно (13.19). Тогда, вследствие линейности ожидания, ожидаемое время, потраченное на подзадачи типа j, составляет $O(n)$. Количество разных типов ограничивается величиной $\log_{\frac{3}{4}} n = O(\log n)$, которая и дает желаемую границу.

(13.21) Ожидаемое время выполнения измененного алгоритма `Quicksort` равно $O(n \log n)$.

Мы рассмотрели измененную версию `Quicksort` для упрощения анализа. Что касается исходной реализации, интуиция подсказывает, что ожидаемое время выполнения у нее не хуже, чем у измененного алгоритма, так как принятие нецентральных разделителей немного помогает с сортировкой (хотя и не так, как при выборе центрального разделителя). Как упоминалось ранее, формализация этого интуитивного представления приводит к границе ожидаемого времени $O(n \log n)$ для исходного алгоритма `Quicksort`, но подробный анализ здесь не приводится.

13.6. Хеширование: рандомизированная реализация словарей

Рандомизация также доказала свою эффективность при разработке структур данных. В этом разделе рассматривается, пожалуй, самое фундаментальное применение рандомизации в этом контексте — метод *хеширования*, который может быть применен для хранения динамически изменяющегося множества элементов. В следующем разделе будет показано, как этот метод помогает создать очень простой алгоритм для задачи, которая встречалась в главе 5, — задачи нахождения ближайшей пары точек на плоскости.

Задача

Одна из основных задач структур данных — хранение множества элементов, изменяющегося со временем. Например, большая компания может вести базу данных о своих работниках и подрядчиках, служба новостей — сохранять первые абзацы статей, поступающих от информагентств; алгоритм поиска — отслеживать небольшую часть экспоненциального пространства поиска, которая уже была исследована, и т. д.

Во всех перечисленных примерах существует очень большое *универсальное множество* U возможных элементов: множество всех возможных людей, всех возможных абзацев (допустим, до определенной длины в символах) или всех возможных решений вычислительно сложной задачи. Структура данных организует хранение информации о множестве $S \subseteq U$, размер которого обычно составляет ничтожную часть U; при этом структура данных обеспечивает возможность вставки и удаления элементов из S, а также быстрой проверки того, принадлежит ли заданный элемент S.

Словарь представляет собой структуру данных, которая поддерживает следующие операции:

- `MakeDictionary` инициализирует словарь, способный хранить подмножество S универсального множества U; только что созданный словарь пуст.
- `Insert`(u) добавляет элемент $u \in U$ в множество S. Во многих случаях с u также связывается дополнительная информация (например, u может быть именем или табельным номером работника, а в структуре данных также сохраняется личная информация о работнике); будем считать, что эта информация сохраняется в словаре как часть записи вместе с u. (Таким образом, в общем случае под «элементом u» будет пониматься u вместе с дополнительной информацией).
- `Delete`(u) удаляет элемент u из множества S, если он там хранится.
- `Lookup`(u) проверяет, принадлежит ли элемент u множеству S (и если принадлежит читает дополнительную информацию, сохраненную вместе с u).

Многие реализации, рассмотренные ранее в книге, используют эти операции (или часть из них): например, в реализациях алгоритмов обхода графа BFS и DFS велись множества S уже посещенных узлов. Тем не менее между теми задачами и текущей ситуацией существует принципиальное различие — размер U. Универсальное множество U в BFS и DFS представляет собой множество узлов V, уже явно заданное как часть входных данных. В таких случаях вполне допустимо хранить множество $S \subseteq U$ так, как это делалось: определение массива с позициями $|U|$, по одной для каждого возможного элемента, и присваивание элементу в позиции, соответствующей u, 1 для $u \in S$ или 0 для $u \notin S$. Такая реализация позволяет выполнять операции вставки, удаления и проверки с постоянным временем простым обращением к элементу массива.

С другой стороны, сейчас рассматривается ситуация, в которой универсальное множество U огромно. Это означает, что мы не сможем использовать массив, размер которого хотя бы сопоставим с U. Принципиальный вопрос заключается в том,

возможно ли в данном случае реализовать словарь, поддерживающий основные операции почти с такой же быстротой, как при относительно малых U.

Ответом на этот вопрос станет рандомизированный метод *хеширования*, описанный в следующем разделе. Нам не удастся достичь таких же характеристик, как в ситуации, в которой возможно определить массив по всему универсальному множеству U, но хеширование позволит подойти достаточно близко к ним.

Разработка структуры данных

Для примера рассмотрим задачу, с которой сталкивается автоматизированный сервис обработки новостей. Предположим, вы получаете постоянный поток коротких статей от разных информагентств, публикаций в блогах и т. д. и сохраняете начальный абзац каждой статьи (который усекается до 1000 символов). Ради полноты охвата используется множество источников информации, поэтому возникает значительная избыточность: одна статья может встречаться многократно.

При появлении новой статьи нужно быстро проверить, не встречался ли такой начальный абзац ранее. Таким образом, словарь — именно та структура данных, которая нужна в этой задаче: универсальное множество U представляет собой множество всех строк длины до 1000 символов (или ровно 1000, если дополнить их пробелами); структура данных поддерживает множество $S \subseteq U$, состоящее из строк (то есть начальных абзацев), встречавшихся ранее.

Одно из возможных решений — ведение связанного списка всех абзацев с просмотром его при каждом появлении нового абзаца. Однако операция `Lookup` в этом случае занимает время, пропорциональное $|S|$. Как вернуться к характеристикам, близким к решению на базе массива?

Хеш-функции

Основная идея хеширования заключается в том, чтобы работать с массивом размера $|S|$ (вместо размера, сравнимого с астрономическим размером U).

Предположим, требуется организовать хранение множества S с размером до n. Для хранения информации создается массив H размера n, а функция $h : U \to \{0, 1, ..., n-1\}$ отображает элементы U на позиции массива. Такая функция называется *хеш-функцией*, а массив H называется *хеш-таблицей*. Если потребуется добавить элемент u в множество S, u просто сохраняется в позиции $h(u)$ массива H. В случае сортировки абзацев текста $h(\cdot)$ может рассматриваться как вычисление некоторой числовой сигнатуры или «контрольной суммы» абзаца u; результат определяет позицию массива для хранения u.

Хеширование превосходно работает, если для всех разных u и v в множестве S выполняется условие $h(u) \neq h(v)$. В таком случае проверка u выполняется за постоянное время: вы проверяете позицию массива $H[h(u)]$, которая либо пуста, либо содержит только u.

Однако в общем случае на такое везение рассчитывать не приходится: могут встретиться два элемента $u, v \in S$, для которых $h(u) = h(v)$. Возникает *коллизия*: два элемента отображаются на одну позицию в H. Существуют разные методы раз-

решения коллизий. Мы будем считать, что в каждой позиции $H[i]$ хеш-таблицы хранится связанный список всех элементов $u \in S$ с $h(u) = i$. Операция `Lookup`(u) работает по следующей схеме:

- вычислить хеш-функцию $h(u)$;
- проверить связанный список в позиции $H[h(u)]$ и проверить, присутствует ли u в списке.

Отсюда следует, что время, необходимое для `Lookup`(u), пропорционально сумме времени вычисления $h(u)$ и длине связанного списка в позиции $H[h(u)]$. В свою очередь, последняя величина представляет собой количество элементов в S, находящихся в коллизии с u. Операции `Insert` и `Delete` работают аналогично: `Insert` добавляет u в связанный список в позиции $H[h(u)]$, а `Delete` просматривает этот список и удаляет элемент u, если он присутствует.

Теперь цель ясна: нужно найти хеш-функцию, которая «равномерно распределяет» добавляемые элементы, чтобы ни один элемент хеш-таблицы H не содержал слишком много позиций. В задачах такого рода анализ худшего случая не слишком информативен. В самом деле, допустим, что $|U| \geq n^2$ (а в некоторых случаях *намного* больше). Тогда для любой выбранной хеш-функции h существует некоторое множество S из n элементов, которые отображаются на одну позицию. В худшем случае будут вставлены все элементы множества, и тогда в операции `Lookup` будет задействован перебор связанного списка длины n. Мы стремимся показать, что рандомизация способна оказать существенную помощь в таких задачах. Как обычно, мы не делаем никаких предположений относительно случайности множества элементов S, а просто применяем рандомизацию при планировании хеш-функции. Коллизии не предотвращаются полностью, но становятся относительно редкими, так что списки получаются достаточно короткими.

Выбор хорошей хеш-функции

Итак, эффективность словаря зависит от выбора хеш-функции h. Как правило, мы будем рассматривать U как большое множество чисел, а затем применять легко вычислимую функцию h, которая отображает каждое число $u \in U$ на некоторое значение из меньшего целочисленного диапазона $\{0, 1, ..., n - 1\}$. Это можно сделать многими простыми способами: например, использовать несколько первых или последних цифр u или просто вычислить остаток от деления u на n. Хотя эти простые решения хорошо работают во многих ситуациях, они также могут привести к многочисленным коллизиям. В самом деле, фиксированный выбор хеш-функции может создать проблемы из-за особенностей элементов u в приложении: может быть, в конкретных цифрах, используемых в определении хеш-функции, кодируется некоторое свойство u, поэтому количество возможных вариантов невелико. Вычисление остатка от деления u на n может привести к аналогичной проблеме, особенно если n является степенью 2. Рассмотрим конкретный пример: допустим, хеш-функция берет абзац английского текста, использует стандартную кодировку символов (например, ASCII) для преобразования его в последовательность битов, а затем оставляет только несколько начальных битов этой последователь-

ности. Можно предполагать большое количество коллизий в элементах массивов, соответствующих битовым строкам для кодирования часто встречающихся слов (например, «The»), тогда как большие части массива могут быть заняты только абзацами, начинающимися со строк вида «qxf», а следовательно, останутся пустыми.

На практике лучше использовать величину ($u \bmod p$) для простого числа p, близкого к n. Хотя в некоторых случаях этот метод порождает хорошие хеш-функции, он не универсален, а некоторые простые числа работают намного лучше других (например, простые числа, очень близкие к степеням 2, работают не так хорошо).

Так как хеширование уже давно применяется на практике, в области выбора хорошей хеш-функции накоплен значительный опыт, и были предложены многие хеш-функции, которые обычно хорошо работают эмпирически. Нам хотелось бы разработать схему хеширования, для которой бы доказывалась эффективность выполнения операций со словарем с высокой вероятностью.

Основная идея, как говорилось ранее, заключается в использовании рандомизации при построении h. Начнем с крайнего случая: для каждого элемента $u \in U$ при вставке в S значение $h(u)$ выбирается случайно с равномерным распределением из множества $\{0,1, ..., n-1\}$, независимо от всех предшествующих вариантов выбора. В этом случае вероятность того, что два случайно выбранных значения $h(u)$ и $h(v)$ совпадут (а следовательно, вызовут коллизию), достаточно мала.

(13.22) В схеме случайного равномерного хеширования вероятность того, что два случайно выбранных значения $h(u)$ и $h(v)$ образуют конфликт (то есть $h(u) = h(v)$), равна точно $1/n$.

Доказательство. Все n^2 возможных вариантов выбора пар $(h(u), h(v))$ имеют одинаковую вероятность, и ровно n из этих вариантов приводят к коллизии. ■

Тем не менее использовать хеш-функцию с независимыми случайными равномерными значениями не получится. Чтобы понять почему, предположим, что элемент u вставляется в S, а потом потребовалось выполнить операцию `Delete`(u) или `Lookup`(u). Спрашивается, где его искать? Нужно знать случайное значение $h(u)$, использованное при вставке, поэтому значение $h(u)$ необходимо где-то сохранить для быстрой выборки. Но ведь это именно та задача, которую мы пытались изначально решить!

Из (13.22) следуют два факта. Во-первых, утверждение закладывает конкретную основу для интуитивных представлений о том, что хеш-функции со «случайным» распределением могут эффективно работать на сокращение числа коллизий. Во-вторых (что важнее для наших целей), мы сможем показать, что планомерное применение рандомизации обеспечит производительность не худшую, чем предлагает (13.22), но при этом приведет к эффективной реализации словаря.

Универсальные классы хеш-функций

Ключевая идея заключается в том, что хеш-функция случайно выбирается не из набора всех возможных функций со значениями $[0,n-1]$, а из особого семейства функций. Каждая функция h в классе функций H отображает универсальное мно-

жество U на множество $\{0,1, ..., n-1\}$, а включаемые функции должны обладать двумя свойствами. Во-первых, они должны предоставлять гарантии из (13.22):

- для любой пары элементов $u, v \in U$ вероятность того, что для случайно выбранной функции $h \in H$ выполняется $h(u) = h(v)$, не более $1/n$.

Класс функций H называется *универсальным*, если для него выполняется первое свойство. Таким образом, (13.22) можно рассматривать как утверждение о том, что класс всех возможных функций, отображающих U на $\{0,1, ...,n-1\}$, является универсальным.

Однако для класса H также должно выполняться второе свойство. Сейчас мы его сформулируем неформально, а позднее приведем более точное определение.

- каждая функция $h \in H$ имеет компактное представление, и для заданных $h \in H$ и $u \in U$ значение $h(u)$ может быть вычислено эффективно.

Класс всех возможных функций этим свойством не обладает: фактически единственным способом представления произвольной функции из U в $\{0, 1, ..., n-1\}$ является запись ее значений для каждого элемента U.

В оставшейся части этого раздела мы продемонстрируем удивительный факт: существуют классы H, обладающие обоими свойствами. Но перед этим необходимо сначала точно сформулировать базовое свойство, которым должен обладать универсальный класс хеш-функций. Если функция h выбирается случайным образом из универсального класса хеш-функций, то для любого множества $S \subset U$, размер которого не превышает n, и любого $u \in U$, ожидаемое количество элементов S, создающих коллизию с u, является константой.

(13.23) Пусть H — универсальный класс хеш-функций, отображающих универсальное множество U на множество $\{0, 1, ..., n-1\}$, S — произвольное подмножество U размера не более n, а u — любой элемент U. Определим X как случайную переменную, равную количеству элементов $s \in S$, для которых $h(s) = h(u)$, для случайно выбранной хеш-функции $h \in H$. (Здесь S и u фиксированны, а случайность проявляется в выборе $h \in H$). Тогда $E[X] \le 1$.

Доказательство. Для элемента $s \in S$ определяется случайная переменная X_s, которая равна 1, если $h(s) = h(u)$, или 0 в противном случае $E[X_s] = \Pr[X_s = 1] \le 1/n$, так как класс функций универсален.

Затем $X = \sum_{s \in S} X_s$, и вследствие линейности ожидания имеем $E[X] = \sum_{s \in S} E[X_s] \le |S| \cdot \frac{1}{n} \le 1$. ■

Разработка универсального класса хеш-функций

Переходим к созданию универсального класса хеш-функций. В качестве размера хеш-таблицы H будет использоваться простое число $p \approx n$. Чтобы иметь возможность использовать целочисленную арифметику при создании хеш-функций, универсальное множество будет определяться векторами вида $x = (x_1, x_2, ... x_r)$ для некоторого целого r, где $0 \le x_i < p$ для каждого i. Например, можно сначала идентифицировать U целыми числами из диапазона $[0, N-1]$ для некоторого N, а затем

воспользоваться смежными блоками из $\lfloor \log p \rfloor$ битов u для определения соответствующих координат x_i. Если $U \subseteq [0, N-1]$, то необходимое количество координат составит $r \approx \log N / \backslash \log n$.

Пусть A — множество всех векторов вида $a = (a_1, ..., a_r)$, где a_i — целое число из диапазона $[0, p-1]$ для каждого $i = 1, ..., r$. Для каждого $a \in A$ определяется линейная функция $h_a(x) = \left(\sum_{i=1}^{r} a_i x_i \right) \bmod p$.

Случайная реализация словарей завершена. Семейство хеш-функций определяется в виде $H = \{h_a : a \in A\}$. Чтобы выполнить операцию `MakeDictionary`, мы выбираем случайную хеш-функцию из H; иначе говоря, выбирается случайный вектор из A (случайным равномерным выбором каждой координаты), и формируется функция h_a. Заметим, что для определения A необходимо найти простое число $p \geq n$. Методы быстрого генерирования целых чисел известны, но мы здесь в эту тему не углубляемся. (На практике можно воспользоваться таблицами известных простых чисел даже для относительно больших n.)

Затем результат используется как хеш-функция для реализации операций *Insert*, `Delete` и `Lookup`. Семейство $H = \{h_a : a \in A\}$ удовлетворяет формальному определению второго свойства: оно имеет компактное представление, так как простым выбором и сохранением случайного $a \in A$ мы можем вычислить $h_a(u)$ для всех элементов $u \in U$. Следовательно, чтобы показать, что H ведет к эффективной реализации словарей на базе хеширования, достаточно показать, что H — универсальное семейство хеш-функций.

Анализ структуры данных

При использовании хеш-функции h_a из определенного ранее класса H коллизия $h_a(x) = h_a(y)$ определяет линейное уравнение с вычислением остатка от деления на простое число p. Для анализа таких уравнений полезно знать следующий «закон сокращения».

(13.24) Для любого простого числа p и любого целого $z \neq 0 \bmod p$ и любых двух целых α, β, если $\alpha z = \beta z \bmod p$, то $\alpha = \beta \bmod p$.

Доказательство. Предположим, $\alpha z = \beta z \bmod p$. Тогда при переносе составляющих получаем $z(\alpha - \beta) = 0 \bmod p$, а следовательно, $z(\alpha - \beta)$ делится на p. Но $z \neq 0 \bmod p$, поэтому z не делится на p. Так как p является простым числом, из этого следует, что $\alpha - \beta$ должно делиться на p; то есть $\alpha = \beta \bmod p$, как и утверждалось. ■

Воспользуемся этим утверждением для доказательства основного результата анализа.

(13.25) Класс линейных функций H, определенный выше, универсален.

Доказательство. Пусть $x = (x_1, x_2, ... x_r)$ и $y = (y_1, y_2, ... y_r)$ — два разных элемента U. Необходимо показать, что вероятность $h_a(x) = h_a(y)$ для случайно выбранного $a \in A$ не превышает $1/p$.

Так как $x = y$, должен существовать индекс j, при котором $x_j \neq y_j$. Рассмотрим следующий способ выбора случайного вектора $a \in A$. Сначала выбираются все координаты a_i, для которых $i \neq j$, после чего выбираются координаты a_j. Покажем, что независимо от выбора всех остальных координат a_i вероятность $h_a(x) = h_a(y)$, обусловленная завершающим выбором a_j, равна $1/p$. Из этого следует, что вероятность $h_a(x) = h_a(y)$ при случайном выборе полного вектора a также должна быть равна $1/p$.

Этот вывод интуитивно понятен: если вероятность равна $1/p$ независимо от того, как выбираются остальные a_i, то она равна $1/p$ в целом. Этот факт также можно доказать напрямую при помощи условных вероятностей. Пусть $\mathcal{E}$ — событие $h_a(x) = h_a(y)$, а событие $\mathcal{F}_b$ — событие получения всеми координатами a_i (для $i \neq j$) последовательности значений b. Ниже будет показано, что $\Pr[\mathcal{E} \mid \mathcal{F}_b] = 1/p$ для всех b. Из этого следует, что $\Pr[\mathcal{E}] = \sum_b \Pr[\mathcal{E} \mid \mathcal{F}_b] \cdot \Pr[\mathcal{F}_b] = (1/p)\sum_b \Pr[\mathcal{F}_b] = 1/p$.

Итак, чтобы завершить доказательство, мы предположим, что значения всех остальных координат a_i были выбраны произвольно, и рассмотрим вероятность такого выбора a_j, при котором $h_a(x) = h_a(y)$. Перенося слагаемые, мы видим, что $h_a(x) = h_a(y)$ в том и только в том случае, если

$$a_j(y_j - x_j) = \sum_{i \neq j} a_i(y_i - x_i) \bmod p .$$

Так как варианты выбора для всех a_i ($i \neq j$) были фиксированы, правая часть может рассматриваться как фиксированная величина m. Кроме того, определим $z = y_j - x_j$.

Теперь достаточно показать, что существует ровно одно значение $0 \leq a_j < p$, при котором выполняется $a_j z = m \bmod p$; действительно, если это так, то вероятность выбора этого значения a_j составляет ровно $1/p$. Предположим, что существуют два таких значения a_j и a'_j. Тогда $a_j z = a'_j z \bmod p$, и, согласно (13.24), должно быть $a_j = a'_j \bmod p$. Но мы предположили, что $a_j, a'_j < p$, поэтому a_j и a'_j должны совпадать. Следовательно, в этом диапазоне существует только одно a_j, при котором $a_j z = m \bmod p$.

В частности, это означает, что вероятность выбора a_j, при котором $h_a(x) = h_a(y)$, равна $1/p$ при любых назначениях других координат a_i в a; следовательно, вероятность коллизии между x и y равна $1/p$. Таким образом, нам удалось показать, что H является универсальным классом хеш-функций. ■

13.7. Нахождение ближайшей пары точек: рандомизированный метод

В главе 5 метод «разделяй и властвуй» использовался для разработки алгоритма нахождения ближайшей пары точек на плоскости за время $O(n \log n)$. Сейчас вы

увидите, как на основе рандомизации создать другой алгоритм для этой задачи на базе структуры данных словаря. Мы покажем, что этот алгоритм выполняется за ожидаемое время $O(n)$ плюс $O(n)$ ожидаемых операций со словарем.

Есть несколько причин, по которым время выполнения алгоритма удобно представлять именно так — с отдельным учетом операций со словарем. В разделе 13.6 было показано, что словари имеют очень эффективную реализацию на базе хеширования, поэтому абстрагирование операций со словарем позволяет рассматривать хеширование как своего рода «черный ящик», с которым общее время выполнения алгоритма определяется гарантиями быстродействия, предоставленными процедурой хеширования. Было показано, что при правильном выборе процедуры хеширования (более мощной и более сложной, чем описанная в разделе 13.6) можно обеспечить выполнение операций со словарем за линейное время, в результате чего общее время выполнения также станет равно $O(n)$. Таким образом, рандомизированное решение приводит к улучшению по сравнению с временем выполнения алгоритма «разделяй и властвуй», приведенного выше. Идеи, приводящие к этой границе $O(n)$, кратко описаны в конце раздела.

Напоследок стоит заметить, что рандомизация проявляется в двух независимых аспектах этого алгоритма: способ обработки входных точек алгоритмом имеет случайную составляющую независимо от реализации структуры данных словаря; и при реализации словаря на базе хеширования дополнительный случайный фактор является частью операций с хеш-таблицей. Выражение времени выполнения через количество операций со словарем позволяет четко разделить эти два применения случайности.

Задача

Для начала вспомним (очень простую) формулировку этой задачи. Даны n точек на плоскости; требуется найти пару точек, находящихся ближе всего друг к другу. Как упоминалось в главе 5, это одна из основных геометрических задач, находящая множество практических применений.

Воспользуемся той же записью, которая уже применялась в предшествующем обсуждении задачи о ближайшей паре. Множество точек обозначается $P = \{p_1, ..., p_n\}$, точка p_i имеет координаты (x_i, y_i); для двух точек $p_i, p_j \in P$ метрика $d(p_i, p_j)$ обозначает стандартное евклидово расстояние между ними. Требуется найти пару точек p_i, p_j, минимизирующую $d(p_i, p_j)$.

Для упрощения анализа будем считать, что все точки лежат в границах единичного квадрата: $0 \le x_i y_i < 1$ для всех $i = 1, ..., n$. Никакой потери общности при этом нет: за линейное время координаты x и y всех точек можно масштабировать так, чтобы они размещались в единичном квадрате, а затем сместить их так, чтобы левый нижний угол квадрата совпадал с началом координат.

Разработка алгоритма

Основная идея алгоритма очень проста: он рассматривает точки в случайном порядке и поддерживает текущее значение δ для ближайшей пары в процессе обработки точек в этом порядке. Добравшись до новой точки p, алгоритм проверяет ее «окрестности»: не находятся ли какие-либо из ранее рассмотренных точек на расстоянии менее δ от p? Если таких точек нет, значит, ближайшая пара не изменилась, и алгоритм переходит к следующей точке в случайном порядке. Если существует точка, находящаяся от p на расстоянии менее δ, то ближайшая пара изменилась и информацию о ней нужно обновить.

Чтобы преобразовать эту схему в эффективный алгоритм, необходимо как-то реализовать задачу нахождения точек в окрестности p. Здесь-то и пригодится структура данных словаря.

Для простоты будем считать, что точки в случайном порядке обозначаются $p_1, ..., p_n$. Работа алгоритма разделена на этапы; на каждом этапе ближайшая пара остается неизменной. Первый этап начинается с присваивания $\delta = d(p_1, p_2)$ — расстояния между первыми двумя точками. На этом этапе алгоритм либо убеждается в том, что δ действительно является расстоянием между ближайшей парой точек, либо находит пару точек p_i, p_j на расстоянии $d(p_i, p_j) < \delta$. Во время этого этапа в порядок $p_1, p_2, ..., p_n$ последовательно добавляются точки. Этап завершается при достижении такой точки p_i, что для некоторого $j < i$ выполняется $d(p_i, p_j) < \delta$. Тогда δ становится ближайшим расстоянием, найденным к текущему моменту, для следующего этапа: $\delta = \min_{j,j<i} d(p_i, p_j)$.

Количество этапов зависит от случайного порядка. Если нам повезет и p_1, p_2 образуют ближайшую пару точек, то хватит одного этапа. Также количество этапов может достигнуть $n - 2$, если добавление новой точки всегда приводит к увеличению минимального расстояния. Мы покажем, что ожидаемое время выполнения алгоритма лежит в пределах постоянного множителя от времени, необходимого в первом (счастливом) случае, когда исходное значение δ оказывается наименьшим расстоянием.

Проверка предлагаемого расстояния

Основная часть алгоритма — метод, который проверяет, остается ли текущая пара точек с расстоянием δ ближайшей парой при добавлении новой точки, и если нет — находит новую ближайшую пару.

Проверка реализуется делением единичного квадрата (область, в пределах которой лежат точки) на подквадраты, стороны которых имеют длину $\delta/2$, как показано на рис. 13.2. Всего будет N^2 подквадратов, где $N = \lceil 1/(2\delta) \rceil$: для $0 \le s \le N-1$ и $1 \le t \le N-1$ подквадрат S_{st} определяется по формуле

$$S_{st} = \{(x, y) : s\delta/2 \le x < (s+1)\delta/2 \le y < (t+1)\delta/2\}.$$

Такое разбиение на подквадраты обладает двумя полезными свойствами. Во-первых, расстояние между любыми двумя точками, принадлежащими одному подквадрату, меньше δ. Во-вторых, любые две точки, находящиеся на расстоянии

менее δ друг от друга, должны находиться либо в одном подквадрате, либо в очень близких подквадратах.

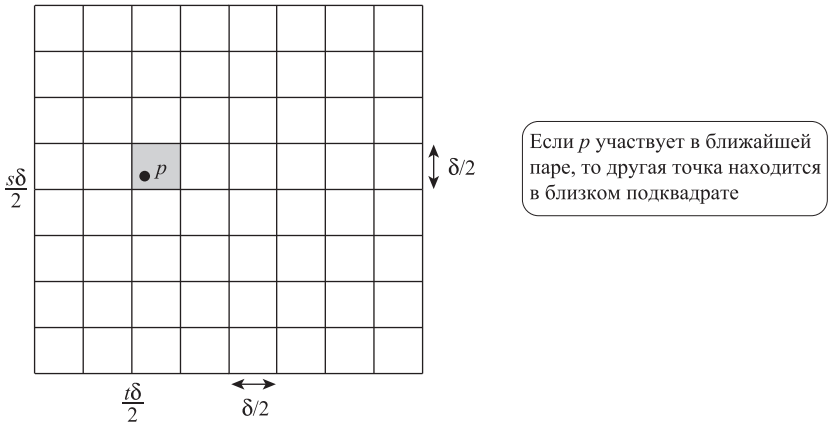


Рис. 13.2. Разбиение квадрата на подквадраты δ/2. Точка p находится в подквадрате S_{st}

(13.26) Если две точки p и q принадлежат одному подквадрату S_{st}, то $d(p, q) < \delta$.

Доказательство. Если точки p и q находятся в одном подквадрате, то обе координаты двух точек отличаются не более чем на δ/2, а следовательно, $d(p,q) \le \sqrt{(\delta/2)^2 + (\delta/2)^2} = \delta/\sqrt{2} < \delta$, как и требуется. ■

Подквадраты S_{st} и $S_{s't'}$ называются *близкими*, если $|s - s'| \le 2$ и $|t - t'| \le 2$. (Обратите внимание: каждый подквадрат близок к самому себе.)

(13.27) Если для двух точек $p, q \in P$ выполняется $d(p, q) < \delta$, то подквадраты, содержащие эти точки, являются близкими.

Доказательство. Рассмотрим две точки $p, q \in P$, принадлежащие подквадратам, которые не являются близкими; будем считать, что $p \in S_{st}$ и $q \in S_{s't'}$, где одна из пар s, s' или t, t' отличается более чем на 2. Следовательно, в одной из соответствующих координат x или y значения p и q отличаются не менее чем на δ, поэтому условие $d(p, q) < \delta$ выполняться не может. ■

Обратите внимание: для любого подквадрата S_{st} множество подквадратов, близких к нему, образует вокруг него сетку 5 × 5. Из этого следует, что у S_{st} может быть не более 25 близких подквадратов, включая сам подквадрат S_{st}. (Если S_{st} прилегает к ребру единичного квадрата, содержащему входные точки, их будет меньше 25.)

Утверждения (13.26) и (13.27) подсказывают базовую структуру алгоритма. Предположим, в какой-то точке алгоритма случайная последовательность точек была частично обработана вплоть до $P' \subseteq P$ и минимальное расстояние между точками в P' равно δ. Для каждой из точек в P' сохраняется содержащий ее подквадрат.

Затем при рассмотрении следующей точки p мы определяем, какому из подквадратов S_{st} она принадлежит. Если p приведет к изменению минимального рас-

стояния, должна существовать некоторая более ранняя точка $p' \in P'$ на расстоянии менее δ от нее; следовательно, из (13.27) следует, что точка p' должна принадлежать одному из 25 квадратов вокруг квадрата S_{st}, содержащего p. Соответственно мы просто проверяем каждый из 25 квадратов, содержит ли он точку в P'; для каждой точки из P', найденной при этом, вычисляется расстояние от нее до p. Согласно (13.26), каждый из этих подквадратов содержит не более одной точки из P', поэтому это потребует не более чем постоянного количества вычислений расстояния. (Кстати, похожая идея использовалась в критической точке алгоритма «разделяй и властвуй» (5.10) из главы 5.)

Структура данных для хранения подквадратов

Высокоуровневое описание алгоритма зависит от возможности назвать подквадрат S_{st} и быстро определить, какие точки P содержатся в нем (если они есть). Словарь — наиболее естественная структура данных для реализации таких операций. Универсальное множество U возможных элементов представляет собой множество всех подквадратов, а множество S, хранимое в структуре данных, состоит из подквадратов с точками из множества P', встречавшимися до настоящего момента. А конкретно, для каждой точки $p' \in P'$ в словаре хранится подквадрат, содержащий ее, вместе с индексом p'. Заметим, что $N^2 = \lceil 1/(2\delta) \rceil^2$ в общем случае намного больше количества точек n. Таким образом, данная ситуация напоминает ту, что была описана в разделе 13.6 для хеширования: универсальное множество возможных элементов (множество всех подквадратов) намного больше количества индексируемых элементов (подквадраты, содержащие входные точки, которые уже встречались).

Затем, при рассмотрении следующей точки p в случайном порядке, определяется содержащий ее подквадрат S_{st} и выполняется операция `Lookup` для каждого из 25 подквадратов, близких к S_{st}. Для всех точек, обнаруженных этими операциями *Lookup*, вычисляется расстояние до p. Если ни одно из этих расстояний не меньше δ, то ближайшее расстояние не изменяется; мы вставляем S_{st} (вместе с p) в словарь и переходим к следующей точке.

Но если будет найдена точка p', для которой $\delta' = d(p, p') < \delta$, то ближайшую пару необходимо обновить. Обновление приводит к весьма масштабным последствиям: так как расстояние ближайшей пары уменьшилось с δ до δ', вся коллекция подквадратов вместе с поддерживающим ее словарем становится бесполезной — ведь она приносит пользу только в том случае, если минимальное расстояние равно δ. Соответственно мы вызываем `MakeDictionary` для создания нового, пустого словаря, в котором хранятся подквадраты с длиной стороны $\delta'/2$. Для каждой точки, встреченной до настоящего момента, мы определяем содержащий ее подквадрат (в новом наборе подквадратов), после чего вставляем этот подквадрат в словарь. После того как это будет сделано, можно переходить к вставке следующей точки в случайном порядке.

Описание алгоритма

Итак, мы полностью разобрали принцип работы алгоритма.

```
Расположить точки в случайной последовательности p1, p2, ..., pn
Пусть δ — минимальное расстояние, обнаруженное на данный момент
Инициализировать δ = d(p1, p2)
Вызвать MakeDictionary для сохранения подквадратов с длиной стороны δ/2
Для i = 1,2, ..., n:
   Определить подквадрат Sst, содержащий pi
   Проверить 25 подквадратов, близких к pi
   Вычислить расстояние от pi до всех точек, находящихся в этих подквадратах
   Если существует точка pj (j < i), для которой δ′ = d(pj,pi) < δ
      Удалить текущий словарь
      Вызвать MakeDictionary для сохранения подквадратов со стороной δ′/2
      Для каждой из точек p1, p2, ..., pi:
         Определить содержащий ее подквадрат с длиной стороны δ′/2
         Вставить этот подквадрат в новый словарь
      Конец цикла
   Иначе
      Вставить pi в текущий каталог
   Конец Если
Конец цикла
```

Анализ алгоритма

Даже сейчас можно кое-что сказать об общем времени выполнения алгоритма. Чтобы рассмотреть новую точку p_i, необходимо выполнить постоянное количество операций `Lookup` и постоянное количество вычислений расстояния. Более того, даже если бы ближайшую пару пришлось обновлять при каждой итерации, всего пришлось бы выполнить n операций `MakeDictionary`.

Недостающим ингредиентом является общая ожидаемая стоимость выполнения алгоритма, обусловленная повторными вставками в новые словари при обновлении ближайшей пары. Этот вопрос будет рассмотрен ниже, а пока сформулируем ту информацию, которой владеем на данный момент.

(13.28) Алгоритм корректно поддерживает информацию ближайшей пары и выполняет не более $O(n)$ вычислений расстояния, $O(n)$ операций `Lookup` и $O(n)$ операций `MakeDictionary`.

В завершение анализа определим границу ожидаемого количества операций `Insert`. Попытка найти хорошую границу для общего ожидаемого количества операций `Insert` на первый взгляд создает немало проблем: обновление ближайшей пары в итерации i приведет к i вставкам, и каждое обновление будет обходиться недешево при больших i. Несмотря на это, мы продемонстрируем удивительный факт: ожидаемое количество вставок равно всего $O(n)$. Это объясняется тем, что наряду с ростом стоимости обновлений сама вероятность таких обновлений снижается соответствующим образом.

Пусть X — случайная переменная, определяющая количество выполняемых операций `Insert`; значение этой случайной переменной определяется случайным порядком, выбранным изначально. Нас интересует граница $E[X]$, и как обычно в подобных ситуациях, X будет полезно разбить на сумму более простых случайных переменных. При этом случайная переменная X_i равна 1, если i-я точка в случайном порядке приводит к изменению минимального расстояния, и 0 в противном случае.

Используя случайные переменные X_i, можно записать простую формулу для общего количества операций `Insert`. Каждая точка вставляется один раз при первом обнаружении; i точек придется вставить заново, если минимальное расстояние изменяется в итерации i. Это позволяет сделать следующее утверждение:

(13.29) Общее количество операций `Insert`, выполняемых алгоритмом, равно $n+\sum_i iX_i$.

Ограничим вероятность $\Pr[X_i = 1]$ того, что рассмотрение i-й точки приводит к изменению минимального расстояния.

(13.30) $\Pr[X_i = 1] \le 2/i$.

Доказательство. Рассмотрим первые i точек $p_1, p_2, ..., p_i$ в случайном порядке. Предположим, минимальное расстояние между этими точками достигается для p и q. Теперь точка p_i может привести к уменьшению минимального расстояния только в том случае, если $p_i = p$ или $p_i = q$. Так как первые i точек следуют в случайном порядке, все они с равной вероятностью могут оказаться последними, так что вероятность того, что последней точкой окажется p или q, равна $2/i$. ■

Заметим, что $2/i$ в (13.30) является только верхней границей, потому что среди первых i точек может встретиться несколько пар, определяющих одно наименьшее расстояние. Из (13.29) и (13.30) следует, что общее количество операций *Insert* ограничивается выражением

$$E[X] = n + \sum_i i \cdot E[X_i] \le n + 2n = 3n.$$

Объединяя это выражение с (13.28), получаем следующую границу для времени выполнения алгоритма:

(13.31) Ожидаемое время выполнения рандомизированного алгоритма нахождения ближайшей пары составляет $O(n)$ плюс $O(n)$ операций со словарем.

Достижение линейного ожидаемого времени выполнения

До настоящего момента структура данных словаря рассматривалась как «черный ящик», а (13.31) выражает границу времени выполнения алгоритма как сумму времени вычислений и операций со словарем. Теперь нам хотелось бы получить границу фактического ожидаемого времени выполнения, но для этого необходимо проанализировать работу, выполняемую при осуществлении этих операций со словарем.

Для реализации словаря будет использоваться универсальная схема хеширования наподобие описанной в разделе 13.6. После того как алгоритм применит

схему хеширования, случайность используется в нем двумя разными способами: во-первых, добавляемые точки располагаются в случайном порядке; во-вторых, для каждого нового минимального расстояния δ рандомизация применяется для создания новой хеш-таблицы с использованием универсальной схемы хеширования.

При вставке новой точки p_i алгоритм использует операцию хеш-таблицы Lookup для нахождения всех узлов в 25 подквадратах, близких к p_i. Однако если в хеш-таблице присутствуют коллизии, то эти 25 операций могут потребовать проверки много более чем 25 узлов. Утверждение (13.23) из раздела 13.6 показывает, что каждая операция Lookup требует проверки $O(1)$ ранее вставленных точек (в ожидании). Интуитивно понятно, что выполнение в ожидании $O(n)$ операций с хеш-таблицей, в каждой из которых задействована проверка в ожидании $O(1)$ элементов, приведет к ожидаемому общему времени выполнения $O(n)$. Чтобы точно выразить это интуитивное представление, необходимо внимательно проследить за тем, как взаимодействуют эти два источника случайности.

(13.32) Допустим, рандомизированный алгоритм нахождения ближайшей пары был реализован с применением универсальной схемы хеширования. В ожидании общее количество точек, рассматриваемых во время операций Lookup, имеет границу $O(n)$.

Доказательство. Из (13.31) мы знаем, что ожидаемое количество операций Lookup равно $O(n)$, а из (13.23) — что каждая из этих операций требует рассмотрения в ожидании всего $O(1)$ точек. Чтобы сделать вывод о том, что ожидаемое количество рассматриваемых точек равно $O(n)$, мы проанализируем связь между этими двумя источниками случайности.

Пусть X — случайная переменная для обозначения количества операций Lookup, выполненных алгоритмом. Случайный порядок σ, выбранный алгоритмом, полностью определяет последовательность значений минимального расстояния, рассматриваемую алгоритмом, и последовательность выполняемых операций со словарем. В результате выбор σ определяет значение X; обозначим это значение $X(\sigma)$, а $\mathcal{E}_\sigma$ — событие выбора алгоритмом случайного порядка σ. Заметим, что условное ожидание $E[X \mid \mathcal{E}_\sigma]$ равно $X(\sigma)$. Кроме того, из (13.31) известно, что $E[X] \le c_0 n$ для некоторой константы c_0.

Теперь рассмотрим эту последовательность операций Lookup для фиксированного порядка σ. Для $i = 1, \ldots, X(\sigma)$ обозначим Y_i количество точек, которые должны быть проверены при выполнении i-х операций Lookup, а именно количество ранее вставленных точек, образующих коллизию с элементом словаря, задействованным в текущей операции Lookup. Нам хотелось бы ограничить ожидаемое значение $\sum_{i=1}^{X(\sigma)} Y_i$ с ожиданием как по случайному выбору σ, так и по случайному выбору хеш-функции.

Из (13.23) известно, что $E[Y_i \mid \mathcal{E}_\sigma] = O(1)$ для всех σ и всех значений i. Полезно иметь возможность ссылаться на константу в выражении O(1), поэтому мы будем говорить, что $E[Y_i \mid \mathcal{E}_\sigma] \le c_1$ для всех σ и всех значений i. Суммируя по всем i и используя линейность ожидания, получаем $E\left[\sum_i Y_i \mid \mathcal{E}_\sigma\right] \le c_1 X(\sigma)$. В итоге получаем:

$$E\left[\sum_{i=1}^{X(\sigma)} Y_i\right] = \sum_{\sigma} \Pr[\mathcal{E}_\sigma] E\left[\sum_i Y_i \mid \mathcal{E}_\sigma\right] \le \sum_{\sigma} \Pr[\mathcal{E}_\sigma] \cdot c_1 X(\sigma) = c_1 \sum_{\sigma} E[X \mid \mathcal{E}_\sigma] \cdot \Pr[\mathcal{E}_\sigma] = c_1 E[X].$$

Так как мы знаем, что $E[X]$ не более $c_0 n$, общее ожидаемое количество рассматриваемых точек не превышает $c_0 c_1 n = O(n)$, что доказывает утверждение. ■

Вооружившись этим утверждением, мы сможем использовать универсальные хеш-функции из раздела 13.6 в алгоритме нахождения ближайшей пары. В ожидании алгоритм будет рассматривать $O(n)$ точек в ходе операций `Lookup`. Потребуется создать несколько хеш-таблиц (новая таблица создается при каждом изменении минимального расстояния) и вычислить $O(n)$ значений хеш-функции. Все хеш-таблицы создаются с одним размером, простым числом $p \ge n$. Мы можем выбрать одно простое число и использовать одну таблицу на протяжении всей работы алгоритма. В этом случае для времени выполнения будет получена следующая граница.

(13.33) В ожидании алгоритм использует для нахождения ближайшей пары точек $O(n)$ вычислений хеш-функции и дополнительное время $O(n)$.

Обратите внимание на отличие этого утверждения от (13.31). Тогда каждая операция со словарем считалась одним атомарным шагом; здесь, напротив, операции со словарем концептуально открыты, чтобы учесть время, связанное с коллизиями в хеш-таблицах и вычислениями хеш-функции.

Наконец, рассмотрим время, необходимое для $O(n)$ вычислений хеш-функции. Сколько времени займет вычисление значения универсальной хеш-функции h? Класс универсальных хеш-функций, разработанных в разделе 13.6, разбивает числа универсального множества U на $r \approx \log N / \log n$ меньших чисел с размером $O(\log n)$ каждое и затем использует $O(r)$ арифметических операций с меньшими числами для вычисления значения хеш-функции. Итак, в вычислении хеша для одной точки задействовано $O(\log N / \log n)$ умножений чисел с размером $\log n$. В сумме получаем $O(n \log N / \log n)$ арифметических операций в ходе выполнения алгоритма — больше величины $O(n)$, на которую мы рассчитывали.

На самом деле количество арифметических операций можно уменьшить до $O(n)$ при использовании более сложного класса хеш-функций. Существуют другие классы универсальных хеш-функций, для которых вычисление значения хеш-функции выполняется за $O(1)$ арифметических операций (хотя эти операции выполняются с большими числами — целыми с размером приблизительно $\log N$). Этот класс улучшенных хеш-функций также создает одну дополнительную сложность в данном случае: схеме хеширования необходимо простое число, превышающее размер универсального множества (вместо размера множества точек). А в этом приложении универсальное множество растет в обратной зависимости от минимального расстояния δ, и в частности возрастает каждый раз, когда обнаруживается новое, меньшее минимальное расстояние. В таких точках приходится искать новое простое число и создавать новую хеш-таблицу. И хотя мы не будем вдаваться в подробности, с этими трудностями можно справиться и привести алгоритм к ожидаемому времени выполнения $O(n)$.

13.8. Рандомизация кэширования

Этот раздел посвящен применению рандомизации в задаче кэширования, представленной в главе 4. Сначала мы разработаем класс алгоритмов *маркировки*, включающих как детерминированные, так и рандомизированные методы. После получения общей гарантии эффективности, распространяющейся на все алгоритмы маркировки, мы покажем, как получить усиленную гарантию для конкретного алгоритма маркировки, использующего рандомизацию.

Задача

Для начала вспомним задачу поддержания кэша из главы 4. В простейшей конфигурации рассматривается процессор, основная память которого состоит из n адресов; процессор также оснащен кэшем на k ячеек памяти, к которым можно обращаться очень быстро. В ячейках кэша могут храниться копии k элементов из основной памяти, и при обращении к адресу памяти процессор сначала проверяет кэш, чтобы узнать, нельзя ли быстро получить необходимые данные. Если нужный элемент присутствует в кэше, обращение называется *попаданием*. Если же элемент в кэше отсутствует, то обращение называется *кэш-промахом*. В случае промаха обращение занимает намного больше времени; более того, один из элементов, находящихся в кэше, необходимо *вытеснить*, чтобы освободить место для нового элемента. (Будем считать, что кэш постоянно заполнен.)

Целью алгоритма поддержания кэша является минимизация количества кэш-промахов, которые являются действительно затратной частью процесса. Последовательность обращений к памяти неподконтрольна алгоритму (она определяется работающим приложением), поэтому алгоритмам остается только выбрать политику вытеснения: какой элемент, находящийся в кэше, должен вытесняться при очередном промахе?

В главе 4 был представлен жадный алгоритм, оптимальный для этой задачи: всегда вытеснять элемент, который понадобится в наиболее отдаленном будущем. Хотя этот алгоритм полезно иметь как эталон производительности кэширования, очевидно, что его невозможно реализовать в условиях реальной работы, потому что заранее неизвестно, когда потребуется тот или иной элемент. Вместо этого необходимо продумать политики вытеснения, которые работают оперативно, руководствуясь информацией только о прошлых запросах.

На практике чаще всего применяется политика, при которой вытесняется элемент, использовавшийся ранее других (то есть элемент с самой ранней временной меткой последнего обращения); эта политика обозначается аббревиатурой *LRU* (Least-Recently-Used). Эмпирическое обоснование LRU заключается в том, что для алгоритмов характерна *локальность* обращений: обычно они многократно используют одни и те же данные в течение некоторого времени. Если к элементу данных давно не было ни одного обращения, весьма вероятно, что к нему в ближайшем будущем обращений не будет.

Здесь мы оценим быстродействие разных политик вытеснения без каких-либо предположений (например, локальности) относительно последовательности запросов. Для этого мы сравним число кэш-промахов, характерных для политики вытеснения σ, с минимальным количеством кэш-промахов, возможных для σ. Последняя величина будет обозначаться $f(\sigma)$; это количество кэш-промахов достигается с оптимальной политикой отдаленного использования. Сравнение политик вытеснения с оптимумом идейно близко к предоставлению гарантий быстродействия для аппроксимирующих алгоритмов, как было сделано в главе 11. Однако обратите внимание на одно интересное различие: причина, по которой оптимум не достигался в аппроксимационном анализе из этой главы (в предположении $P \neq NP$), заключалась в том, что алгоритмы ограничивались требованием о полиномиальном времени выполнения; с другой стороны, сейчас политики вытеснения ограничиваются фактом неизвестности обращений, которые могут поступить в будущем.

Казалось бы, если политики вытеснения действуют в условиях оперативности решений, бесполезно даже пытаться давать какие-то прогнозы по поводу их эффективности: что мешает создать последовательность обращений, которая приведет в полное замешательство любую политику оперативного вытеснения? Как ни странно, мы действительно можем дать абсолютные гарантии по производительности разных оперативных политик относительно оптимума.

Сначала мы покажем, что количество кэш-промахов при политике LRU для любой последовательности обращений ограничивается примерно оптимумом, умноженным на k. Затем при помощи рандомизации будет разработана версия LRU, предоставляющая экспоненциально усиленную границу своего быстродействия: количество промахов в ней никогда не превышает оптимум более чем в $O(\log k)$ раз.

Разработка класса алгоритмом маркировки

Границы как политики LRU, так и ее рандомизированного варианта следуют из общего шаблона построения оперативных политик вытеснения — класса политик, называемого *алгоритмами маркировки*. В их основе лежит следующая идея: чтобы обеспечить хорошие показатели по сравнению с эталоном $f(\sigma)$, необходима политика вытеснения, которая учитывает различия между двумя возможностями: (a) в недавнем прошлом последовательность обращений содержала более k разных элементов; или (b) в недавнем прошлом последовательность обращений поступила исключительно из множества, содержащего не более k элементов. В первом случае мы знаем, что значение $f(\sigma)$ должно увеличиться, так как ни один алгоритм не может обратить более k разных элементов без кэш-промаха. Но во втором случае может оказаться, что σ проходит через длинную серию, в которой оптимальный алгоритм вообще не приводит к промахам. В этом случае наша политика должна принять меры к тому, чтобы количество промахов было минимальным.

Руководствуясь этими соображениями, мы опишем базовую структуру алгоритма маркировки, которая отдает предпочтение вытеснению элементов, не использо-

вавшихся в течение длительного времени. Процесс выполнения такого алгоритма делится на *фазы*; ниже приведено описание одной фазы.

```
Каждый элемент памяти либо помечен маркером, либо не помечен.
В начале фазы все элементы не помечены маркером.
При обращении к элементу s:
   Пометить s
   Если s находится в кэше, ничего не вытеснять.
   Иначе s отсутствует в кэше:
      Если все элементы, находящиеся в кэше, помечены
         Объявить фазу завершенной
         Обработка s откладывается до начала следующей фазы
      Иначе вытеснить из кэша непомеченный элемент
      Конец Если
   Конец Если
```

Обратите внимание: этот фрагмент описывает не один конкретный алгоритм, а класс алгоритмов, потому что ключевой шаг — вытеснение непомеченного элемента из кэша — не указывает, какой непомеченный элемент должен быть выбран. Вы увидите, что в зависимости от способа разрешения этой неоднозначности образуются политики вытеснения с разными свойствами и гарантиями производительности.

Для начала заметим, что в начале фазы все элементы не помечены, а пометка осуществляется только при обращении, поэтому обращения к непомеченным элементам приходятся на более ранний момент, чем обращения к помеченным элементам. Именно в этом смысле алгоритм маркировки пытается вытеснять элементы, к которым не было обращений за последнее время. Кроме того, если в любой момент фазы в кэше присутствуют непомеченные элементы, элемент с самым ранним обращением помечен быть не может. Отсюда следует, что политика LRU вытесняет непомеченный элемент, когда он доступен, и мы приходим к следующему утверждению:

(13.34) Политика LRU является алгоритмом маркировки.

Анализ алгоритмов маркировки

В этом разделе мы рассмотрим метод анализа алгоритмов маркировки, а в конце будет приведена оценка сложности, применимая ко всем алгоритмам маркировки. После этого при введении рандомизации этот анализ нужно будет усилить.

Рассмотрим произвольный алгоритм маркировки, работающий с последовательностью обращений σ. Для анализа представим оптимальный алгоритм кэширования, который работает с σ параллельно с этим алгоритмом разметки и обеспечивает общую стоимость $f(\sigma)$. Предположим, последовательность σ состоит из r фаз, определяемых алгоритмом маркировки. Чтобы упростить рассмотрение анализа, мы «дополним» последовательность σ в начале и в конце дополнительными обращениями: никаких лишних кэш-промахов в оптимальном алгоритме

они не создадут (то есть не приведут к увеличению $f(\sigma)$), поэтому любая граница, доказанная для эффективности алгоритма маркировки относительно оптимума дополненной последовательности, также будет применима к σ. А конкретно перед первой фазой вставляется «фаза 0», в которой запрашиваются все элементы, изначально находящиеся в кэше. Это никак не влияет на стоимость алгоритма маркировки или оптимального алгоритма. Также можно представить, что за завершающей фазой r следует эпилог, в котором каждый элемент, находящийся в кэше оптимального алгоритма, запрашивается дважды по кругу. Значение $f(\sigma)$ при этом не увеличивается, а к концу второго прохода в алгоритме маркировки каждый элемент будет находиться в кэше и каждый будет помечен.

Для границы эффективности необходимы две составляющие: верхняя граница для количества промахов, порожденных алгоритмом маркировки, и нижняя граница, которая означает, что оптимум должен породить по крайней мере какое-то количество кэш-промахов.

Деление последовательности обращений σ на фазы играет ключевую роль для получения границ. Прежде всего, история фазы с точки зрения алгоритма маркировки выглядит так: в начале фазы все элементы не помечены. Любой элемент, к которому происходит обращение в фазе, помечается маркером и затем остается в кэше в оставшейся части фазы. В ходе фазы количество помеченных элементов растет с 0 до k, а следующая фаза начинается с обращения к $(k + 1)$-му элементу, отличному от всех помеченных. Некоторые выводы из этого описания представлены в следующем утверждении:

(13.35) В каждой фазе σ содержит обращения ровно к k разным элементам. Последующая фаза начинается с обращения к другому $(k + 1)$-му элементу.

Так как элемент после пометки остается в кэше до конца фазы, алгоритм маркировки не может породить кэш-промах для элемента чаще, чем раз в фазу. В сочетании с (13.35) это дает верхнюю границу для количества промахов, вызванных алгоритмом маркировки.

(13.36) Алгоритм маркировки порождает не более k промахов за фазу и не более kr промахов за все r фаз.

Для нижней границы оптимума имеется следующий факт:

(13.37) Оптимум порождает не менее $r - 1$ промахов. Другими словами, $f(\sigma) \geq r - 1$.

Доказательство. Рассмотрим любую фазу, кроме последней, в ситуации непосредственно после первого обращения (к элементу s) в этой фазе. В настоящий момент s находится в кэше, поддерживаемом оптимальным алгоритмом, и из (13.35) следует, что в оставшейся части этой фазы произойдут обращения к $k - 1$ другим элементам, а в первом обращении следующей фазы также произойдет обращение к k-му другому элементу. Обозначим S это множество из k элементов, отличных от s. Заметим, что по крайней мере один из элементов S в настоящее время не находится в кэше, поддерживаемом оптимальным алгоритмом (так как после s остается место только для $k - 1$ других элементов), и в оптимальном алгоритме при первом обращении к этому элементу произойдет кэш-промах.

Итак, мы показали, что для каждой фазы $j < r$ последовательность от второго обращения в фазе j до первого обращения в фазе $j + 1$ приводит по крайней мере к одному промаху в оптимуме. В сумме это дает по крайней мере $r - 1$ промахов. ■

Объединяя (13.36) и (13.37), мы получаем следующие гарантии эффективности:

(13.38) Для любого алгоритма маркировки количество кэш-промахов для любой последовательности σ не превышает $k \cdot f(\sigma) + k$.

Доказательство. Количество кэш-промахов, порожденных алгоритмом маркировки, не превышает

$$kr = k(r - 1) + k \leq k \cdot f(\sigma) + k,$$

где последнее неравенство следует из (13.37). ■

Обратите внимание: «$+k$» в границе (13.38) представляет собой аддитивную константу, не зависящую от длины последовательности обращений σ, поэтому ключевым аспектом границы является множитель k относительно оптимума. Чтобы понять, что этот множитель k является лучшей возможной границей для некоторых алгоритмов маркировки (и для LRU в частности), рассмотрим поведение LRU для последовательности обращений, в которой $k + 1$ элементов многократно запрашиваются по кругу. LRU каждый раз будет вытеснять элемент, который понадобится только на следующем шаге, а следовательно, породит промах при каждом обращении. (Такая чудовищная неэффективность кэширования может проявиться на практике ровно по той же причине: программа выполняет цикл, который чуть-чуть велик для кэша.) С другой стороны, оптимальная политика — вытеснение элемента, который потребуется в самом отдаленном будущем, — порождает промах только через каждые k шагов, так что LRU порождает в k раз больше промахов, чем оптимальная политика.

Разработка рандомизированного алгоритма маркировки

Только что рассмотренный неудачный пример для LRU показывает, что если вы хотите получить лучшую границу для алгоритма оперативного кэширования, рассуждать на уровне полностью обобщенных алгоритмов маркировки не удастся. Вместо этого мы определим простой рандомизированный алгоритм маркировки и покажем, что он никогда не порождает более чем в $O(\log k)$ раз больше промахов, чем оптимальный алгоритм — экспоненциально улучшенная граница.

Рандомизация является естественным выбором для предотвращения нежелательных серий «ошибочных» выборов из плохого примера для LRU. Чтобы получить эту плохую последовательность, нужно было определить последовательность, которая всегда вытесняет ошибочный элемент. Рандомизация позволит политике «в среднем» вытеснять непомеченный элемент, который по крайней мере не понадобится немедленно. А конкретно, строка

```
Иначе вытеснить из кэша непомеченный элемент
```

в общем описании алгоритма, не уточняющая, как именно должен выбираться непомеченный элемент, в рандомизированном алгоритме маркировки заменяется следующим правилом:

```
Иначе вытеснить из кэша непомеченный элемент, выбранный случайным образом
   с равномерным распределением
```

Пожалуй, это самый простой способ включения рандомизации в механизм маркировки[1].

Анализ рандомизированного алгоритма маркировки

Теперь нам хотелось бы получить для рандомизированного алгоритма маркировки границу более сильную, чем (13.38); но для этого придется немного усложнить анализ из (13.36) и (13.37). Дело в том, что существуют последовательности σ с r фазами, для которых рандомизированный алгоритм маркировки действительно можно заставить порождать kr промахов — представьте последовательность, в которой никогда не повторяются элементы. Однако для таких последовательностей оптимум будет порождать много более $r - 1$ промахов. Необходимо каким-то образом сблизить верхнюю и нижнюю границы на основании структуры последовательности.

Последовательность с неповторяющимися элементами является крайним выражением того различия, которое нам хотелось бы подчеркнуть: непомеченные элементы в середине фазы полезно разделить на две категории. Непомеченный элемент называется *свежим*, если в предыдущей фазе он также не помечался; элементы, помечавшиеся в предыдущей фазе, будут называться *устаревшими*.

Вспомните описание одной фазы, которое привело к (13.35): в начале фазы ни один элемент не помечен, и фаза содержит обращения к k разным элементам, каждый из которых переходит из непомеченного состояние в помеченное при первом обращении. Пусть для этих k обращений к непомеченным элементам в фазе j величина c_j обозначает количество обращений к свежим элементам.

Чтобы усилить результат по сравнению с утверждением (13.37), которое фактически утверждало, что оптимум порождает по крайней мере один промах на фазу, мы сформулируем границу в контексте количества свежих элементов в фазе:

(13.39) $$f(\sigma) \geq \frac{1}{2}\sum_{j=1}^{r} c_j.$$

[1] Однако это не самый простой способ включения рандомизации в алгоритм кэширования. Можно рассмотреть *чисто случайный алгоритм*, который отказывается от самой концепции маркировки и при каждом кэш-промахе выбирает один из k текущих элементов для вытеснения случайным образом. (Обратите внимание на различие: рандомизированный алгоритм маркировки выполняет рандомизацию только среди непомеченных элементов.) Хотя здесь это не доказывается, чисто случайный алгоритм может породить по крайней мере в c раз больше промахов, чем оптимум, для любой константы $c < k$, поэтому он не обеспечивает выигрыша по сравнению с LRU.

Доказательство. Пусть $f_j(\sigma)$ обозначает количество промахов, порожденных оптимальным алгоритмом в фазе j, и $f(\sigma) = \frac{1}{2}\sum_{j=1}^{r} f_j(\sigma)$. Из (13.35) известно, что в любой фазе j существуют обращения к k разным элементам. Более того, по нашему определению *свежих* элементов, в фазе $j + 1$ выдаются обращения к $c_j + 1$ дальнейшим элементам; таким образом, между фазами j и $j + 1$ происходят обращения к минимум $k + c_{j+1}$ разных элементов. Отсюда следует, что оптимальный алгоритм должен порождать не менее c_{j+1} промахов во время фаз j и $j + 1$ и $f_j(\sigma) + f_{j+1}(\sigma) \geq c_{j+1}$. Условие выполняется даже для $j = 0$, так как оптимальный алгоритм порождает c_1 промахов в фазе 1. Следовательно, имеем

$$\sum_{j=0}^{r-1}(f_j(\sigma) + f_{j+1}(\sigma)) \geq \sum_{j=0}^{r-1} c_{j+1}.$$

Но левая сторона не превышает $2\sum_{j=1}^{r} f_j(\sigma) = 2f(\sigma)$, а правая сторона равна $\sum_{j=1}^{r} c_j$. ■

Теперь определим верхнюю границу ожидаемого количества промахов, порожденных рандомизированным алгоритмом маркировки, также выраженную в контексте количества свежих элементов в каждой фазе. Объединение верхней и нижней границ дает искомые гарантии эффективности. В следующем утверждении M_σ обозначает случайную переменную, равную количеству кэш-промахов, порожденных рандомизированным алгоритмом маркировки для последовательности обращений σ.

(13.40) Для каждой последовательности обращений σ выполняется $E[M_\sigma] \leq H(k)\sum_{j=1}^{r} c_j$.

Доказательство. Вспомните, что c_j обозначает количество обращений в фазе j к свежим элементам. В этой фазе выдаются k обращений к непомеченным элементам, и каждый непомеченный элемент является либо свежим, либо устаревшим, поэтому в фазе j должно быть $k - c_j$ обращений к непомеченным устаревшим элементам.

Обозначим X_j количество промахов, порожденных рандомизированным алгоритмом маркировки в фазе j. Каждое обращение к свежему элементу приводит к гарантированному промаху для рандомизированного алгоритма маркировки; так как свежий элемент не был помечен в предыдущей фазе, он не может находиться в кэше, когда он запрашивается в фазе j. Следовательно, рандомизированный алгоритм маркировки порождает минимум c_j промахов в фазе j из-за обращений к свежим элементам.

Напротив, с устаревшими элементами дело обстоит сложнее. Фаза начинается с k устаревших элементов в кэше; это элементы, с которых в начале фазы была массово снята пометка. При обращении к устаревшему элементу s возникает проблема: не был ли он вытеснен ранее в этой фазе рандомизированным алгоритмом маркировки и не вызовет ли он теперь промах для возвращения? Какова вероятность того, что i-е обращение к устаревшему элементу (допустим, s) приведет

к промаху? Предположим, в этой фазе было $c \le c_j$ обращений к свежим элементам. Тогда кэш содержит c ранее свежих элементов, которые сейчас помечены, $i - 1$ ранее устаревших элементов, которые теперь помечены, и $k - c - i + 1$ элементов, устаревших и не помеченных в этой фазе. Но всего существует $k - i + 1$ элементов, которые все еще остаются устаревшими; и поскольку ровно $k - c - i + 1$ из них находятся в кэше, остальные c там находиться не могут. Каждый из $k - i + 1$ устаревших элементов с равной вероятностью может отсутствовать в кэше, поэтому s отсутствует в кэше на данный момент с вероятностью $\frac{c}{k-i+1} \le \frac{c_j}{k-i+1}$.

Это вероятность промаха при обращении к s. Суммируя по всем обращениям к непомеченным элементам, получаем

$$E[X_j] \le c_j + \sum_{i=1}^{k-c_j} \frac{c_j}{k-i+1} \le c_j \left[1 + \sum_{\ell=c_j+1}^{k} \frac{1}{\ell}\right] = c_j(1 + H(k) - H(c_j)) \le c_j H(k).$$

Таким образом, общее ожидаемое количество кэш-промахов, порожденных рандомизированным алгоритмом маркировки, равно

$$E[M_\sigma] = \sum_{j=1}^{r} E[X_j] \le H(k) \sum_{j=1}^{r} c_j \,. \blacksquare$$

Объединяя (13.39) и (13.40), мы немедленно приходим к следующей гарантии эффективности:

(13.41) Ожидаемое количество промахов, порожденных рандомизированным алгоритмом маркировки, не превышает $2H(k) \cdot f(\sigma) = O(\log k) \cdot f(\sigma)$.

13.9. Границы Чернова

В разделе 13.3 было приведено формальное определение случайной переменной; с тех пор мы работали с этим определением и следствиями из него. Интуитивно понятно, что значение случайной переменной должно быть «близко» к его ожиданию с достаточно высокой вероятностью, но мы еще не исследовали, насколько обоснованны эти ожидания. В этом разделе рассматриваются некоторые результаты, которые позволяют делать подобные оценки, и приведены некоторые практические примеры.

Две случайные переменные X и Y называются *независимыми*, если для всех значений i и j события $\Pr[X = i]$ и $\Pr[Y = j]$ являются независимыми. Это определение естественным образом расширяется для больших множеств случайных переменных. Теперь рассмотрим случайную переменную X, которая является суммой нескольких независимых случайных переменных, принимающих значения 0 или 1: $X = X_1 + X_2 + \ldots + X_n$, где X_i принимает значение 1 с вероятностью p_i, и значение 0 в противном случае. Из линейности ожидания следует $E[X] = \sum_{i=1}^{n} p_i$. Интуитивно понятно, что вследствие независимости случайных переменных X_1, X_2, ..., X_n их отклонения должны «компенсироваться», а значение их суммы X с высокой веро-

ятностью будет близко к своему ожиданию. Это действительно так, и мы сформулируем две конкретные версии этого результата: одна ограничивает вероятность того, что X отклоняется выше $E[X]$, а другая — вероятность того, что X отклоняется ниже $E[X]$. Эти результаты называются *границами Чернова* по фамилии ученого, впервые установившего границы в этой форме.

(13.42) Пусть $X, X_1, X_2, ..., X_n$ определяются так, как показано выше, а $\mu \geq E[X]$. Тогда для любого $\delta > 0$

$$\Pr[X > (1+\delta)\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}.$$

Доказательство. Чтобы ограничить вероятность того, что X превышает $(1 + \delta)\mu$, выполним серию простых преобразований. Сначала заметим, что для любых $t > 0$ выполняется $\Pr[X > (1+\delta)\mu] = \Pr\left[e^{tX} > e^{t(1+\delta)\mu}\right]$, так как функция $f(x) = e^{tx}$ монотонна по x. Этот факт будет использоваться со значением t, которое мы выберем позднее.

Затем мы воспользуемся некоторыми простыми свойствами ожидания. Для случайной переменной Y по определению ожидания выполняется $\gamma\Pr[Y > \gamma] \leq E[Y]$. Это позволяет ограничить вероятность того, что Y превысит γ, в контексте $E[Y]$. Объединяя эти две идеи, приходим к следующим неравенствам:

$$\Pr[X > (1+\delta)\mu] = \Pr\left[e^{tX} > e^{t(1+\delta)\mu}\right] \leq e^{-t(1+\delta)\mu} E\left[e^{tX}\right].$$

Затем необходимо ограничить ожидание $E\left[e^{tX}\right]$. Если записать X в виде $X = \sum_i X_i$, ожидание принимает вид $E\left[e^{tX}\right] = E\left[e^{t\sum_i X_i}\right] = E\left[\prod_i e^{tX_i}\right]$. Для независимых переменных Y и Z ожидание произведения YZ равно $E[YZ] = E[Y]E[Z]$. Переменные X_i независимы, поэтому $E\left[\prod_i e^{tX_i}\right] = \prod_i E\left[e^{tX_i}\right]$.

Тогда e^{tX_i} равно e^t с вероятностью p_i и $e^0 = 1$ в противном случае, поэтому ожидание может быть ограничено в следующем виде:

$$E\left[e^{tX_i}\right] = p_i e^t + (1 - p_i) = 1 + p_i(e^t - 1) \leq e^{p_i(e^t - 1)},$$

где последнее неравенство вытекает из того факта, что $1 + \alpha \leq e\alpha$ для любых $\alpha \geq 0$. Объединяя эти неравенства, получаем следующую границу.

$$\Pr[X > (1+\delta)\mu] \leq e^{-t(1+\delta)\mu} E\left[e^{tX}\right] = e^{-t(1+\delta)\mu} \prod_i E\left[e^{tX_i}\right]$$
$$\leq e^{-t(1+\delta)\mu} \prod_i e^{p_i(e^t - 1)} \leq e^{-t(1+\delta)\mu} e^{\mu(e^t - 1)}.$$

Чтобы получить границу, заявленную в утверждении, подставляем $t = \ln(1 + \delta)$. ■

Если (13.42) определяет верхнюю границу и показывает, что большие отклонения X от ожидания в сторону увеличения маловероятны, следующее утверждение (13.43) определяет нижнюю границу и показывает, что отклонения X от ожидания

в сторону уменьшения тоже маловероятны. Обратите внимание: формулировки результатов не симметричны, и это логично: для верхней границы представляют интерес значения δ, много большие 1, тогда как для нижней границы это не имеет смысла.

(13.43) Пусть X, X_1, X_2, ..., X_n и μ определяются так, как показано выше, и $\mu \le E[X]$. Тогда для любого $1 > \delta > 0$

$$\Pr[X < (1-\delta)\mu] < e^{-\frac{1}{2}\mu\delta^2}.$$

Доказательство (13.43) аналогично доказательству (13.42), и мы его не приводим. В рассматриваемых далее примерах важно помнить сами утверждения (13.42) и (13.43), а не технические тонкости их доказательств.

13.10. Распределение нагрузки

В разделе 13.1 рассматривалась распределенная система, в которой передача информации между процессами была затруднена и рандомизация в некоторой степени заменяла явную координацию и синхронизацию. Вернемся к этой теме и рассмотрим еще один стилизованный пример рандомизации в распределенной среде.

Задача

В некоторой системе m заданий поступают в потоковом режиме для немедленной обработки. Имеется группа из n идентичных процессоров, способных выполнять задания; требуется связать каждое задание с некоторым процессором так, чтобы равномерно распределить нагрузку между процессорами. Если в системе имеется центральный контроллер, который получает задания и передает их процессорам по кругу, он позволяет тривиально гарантировать, что за каждым процессором будет закреплено не более $\lceil m/n \rceil$ заданий, — самое равномерное возможное распределение.

Но предположим, в системе отсутствуют средства координации или централизации. Намного более простая схема просто закрепляет каждое задание за одним из случайно выбранных процессоров с равной вероятностью. Интуитивно понятно, что задания в этой схеме тоже должны распределяться равномерно, потому что вероятность получения задания каждым процессором одинакова. В то же время, поскольку назначение заданий полностью случайно, идеального распределения ждать не приходится. Спросим себя: насколько хорошо работает этот простой рандомизированный метод?

Хотя мы будем придерживаться терминологии с заданиями и процессорами, стоит заметить, что похожие проблемы возникают при анализе хеш-функций (см. раздел 13.6), где вместо распределения заданий между процессорами про-

исходит распределение элементов между ячейками хеш-таблицы. Стремление к равномерности распределения в случае хеш-таблиц основано на желании по возможности минимизировать количество коллизий для каждой конкретной ячейки. Как следствие, приведенный в этом разделе анализ также актуален для изучения схем хеширования.

Анализ случайного распределения заданий

Как вы вскоре увидите, анализ процесса случайного распределения нагрузки зависит от относительных размеров m (количество заданий) и n (количество процессоров). Начнем с самого простого случая: $m = n$. На каждый процессор может быть назначено ровно одно задание, хотя это и маловероятно. Скорее всего, на некоторых процессорах заданий вообще не будет, а на других их будет несколько. Для оценки качества этой эвристики рандомизированного распределения нагрузки стоит проанализировать, сколько заданий может быть закреплено за процессором.

Обозначим X_i случайную переменную, равную количеству заданий, назначенных на процессор i (для $i = 1, 2, ..., n$). Ожидаемое значение X_i определяется легко; пусть Y_{ij} — случайная переменная, которая равна 1, если задание j назначено на процессор i, и 0 в противном случае. Тогда $X_i = \sum_{j=1}^{n} Y_{ij}$ и $E[Y_{ij}] = 1/n$, а следовательно, $E[X_i] = \sum_{j=1}^{n} E[Y_{ij}] = 1$. Но нас интересует, насколько X_i может отклониться выше своего ожидания: какова вероятность того, что $X_i > c$? Чтобы определить верхнюю границу, можно применить (13.42): X_i представляет собой сумму независимых случайных переменных $\{Y_{ij}\}$, принимающих значения 0 и 1; имеем $\mu = 1$ и $1+\delta = c$. Следовательно, следующее утверждение истинно.

(13.44) $$\Pr[X_i > c] < \left(\frac{e^{c-1}}{c^c}\right).$$

Чтобы вероятность того, что *какие-либо* X_i превысят c, мы вычислим границу объединения по $i = 1, 2, ..., n$; следовательно, c необходимо выбрать достаточно большим, чтобы опустить $\Pr[X_i > c]$ ниже $1/n$ для каждого i. Для этого необходимо рассмотреть знаменатель c^c в (13.44). Чтобы знаменатель был достаточно большим, необходимо понять, как эта величина растет с c; и для этого мы сначала зададимся вопросом: каким должно быть x, чтобы выполнялось $x^x = n$?

Предположим, такое число x будет обозначаться $\gamma(n)$. Для $\gamma(n)$ не существует выражения в замкнутой форме, но его асимптотическое значение можно определить следующим образом. Если $x^x = n$, то при взятии логарифма получаем $x \log x = \log n$; повторное взятие логарифма дает $\log x + \log\log x = \log\log n$. Следовательно, имеем

$$2 \log x > \log x + \log\log x = \log n > \log x,$$

Используя это выражение для деления равенства $x \log x = \log n$, получаем

$$\frac{1}{2}x \le \frac{\log n}{\log\log n} \le x = \gamma(n).$$

Таким образом, $\gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$.

Если теперь выбрать $c = e\,\gamma(n)$, то из (13.44) следует

$$\Pr[X_i > c] < \left(\frac{e^{c-1}}{c^c}\right) < \left(\frac{e}{c}\right)^c = \left(\frac{1}{\gamma(n)}\right)^{e\gamma(n)} < \left(\frac{1}{\gamma(n)}\right)^{2\gamma(n)} = \frac{1}{n^2}.$$

Итак, применение границы объединения к этой верхней границе для $X_1, X_2, ..., X_n$ дает следующее утверждение:

(13.45) С вероятностью не менее $1 - n^{-1}$ ни один процессор не получит более $e\gamma(n) = \Theta\left(\frac{\log n}{\log\log n}\right)$ заданий.

При помощи более сложного анализа можно показать, что эта граница является асимптотически точной: с высокой вероятностью некоторый процессор действительно получает $\Omega\left(\frac{\log n}{\log\log n}\right)$ заданий.

Итак, хотя нагрузка на некоторые процессоры, вероятно, превысит ожидание, это отклонение только логарифмически зависит от количества процессоров.

Увеличение количества заданий

Границы Чернова помогут обосновать, что с введением в систему новых заданий нагрузка быстро «сглаживается»: количество заданий на разных процессорах быстро выравнивается в пределах постоянных множителей.

А конкретно, при $m = 16n \ln n$ заданий ожидаемая нагрузка на процессор составит $\mu = 16 \ln n$. Используя (13.42), мы видим, что вероятность превышения нагрузки любого процессора величины $32 \ln n$ не превышает

$$\Pr[X_i > 2\mu] < \left(\frac{e}{4}\right)^{16\ln n} < \left(\frac{1}{e^2}\right)^{\ln n} = \frac{1}{n^2}.$$

Кроме того, вероятность того, что нагрузка любого процессора окажется ниже $8 \ln n$, не превышает

$$\Pr\left[X_i < \frac{1}{2}\mu\right] < e^{-\frac{1}{2}\left(\frac{1}{2}\right)^2(16\ln n)} = e^{-2\ln n} = \frac{1}{n^2}.$$

Применяя границу объединения, приходим к следующему результату.

(13.46) В системе с n процессорами и $\Omega(n \log n)$ заданиями нагрузка на каждый процессор с высокой вероятностью находится в диапазоне от половины до удвоенной средней.

13.11. Маршрутизация пакетов

Рассмотрим более сложный пример применения рандомизации для снижения конкуренции в распределенных системах, а именно в контексте *маршрутизации пакетов*.

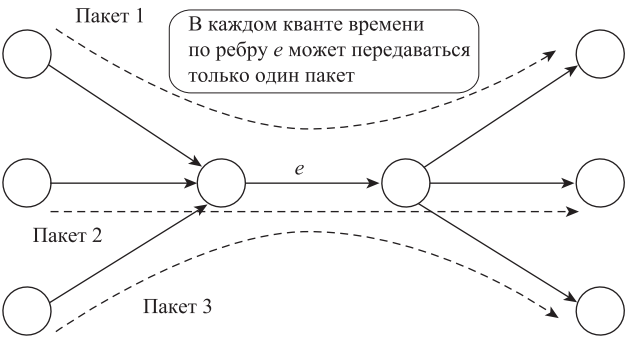


Рис. 13.3. Три пакета, в пути которых входит общее ребро e

Задача

Механизм *маршрутизации пакетов* обеспечивает передачу данных между узлами большой сети, которая может моделироваться направленным графом $G = (V, E)$. Если узел s захочет передать данные узлу t, эти данные разбиваются на *пакеты*, каждый из которых передается по сетевому пути s–t P. В любой момент времени в сети может передаваться множество пакетов, связанных с разными источниками и получателями и передаваемых по разным путям. При это действует ключевое ограничение: по каждому ребру e в каждом кванте времени может передаваться только один пакет. Таким образом, когда пакет p прибывает к ребру e, может оказаться, что несколько других пакетов уже ожидают передачи по e; в этом случае p встает в очередь, связанную с ребром e, и ожидает, пока ребро e будет готово передать его. На рис. 13.3, например, три разных пакета с разными источниками и получателями требуют передачи по ребру e; если они поступят к e одновременно, то некоторым из них придется ждать в очереди.

Предположим, имеется сеть G с множеством пакетов, которые должны быть переданы по заданным путям. Нужно понять, сколько шагов понадобится для того, чтобы все пакеты достигли своих получателей. И хотя все пути пакетов полностью известны заранее, мы сталкиваемся с алгоритмической задачей хронометража

перемещений пакетов по ребрам. В частности, необходимо решить, когда следует отправить каждый пакет из источника, а также выбрать *политику управления очередью* для каждого ребра e (то есть как выбирать следующий пакет для передачи из очереди e на каждом шаге).

Важно понимать, что решения по планированию пакетов могут сильно повлиять на время, необходимое для того, чтобы все пакеты добрались до своих получателей. Например, рассмотрим древовидную сеть на рис. 13.4, в которой девять пакетов должны быть переданы по соответствующим пунктирным путям в дереве. Предположим, все пакеты выходят из источников немедленно, и каждое ребро e всегда передает пакет, ближайший к точке получения. В этом случае пакету 1 придется ожидать пакетов 2 и 3 на втором уровне дерева, а позднее — пакетов 6 и 9 на четвертом уровне дерева. Таким образом, для достижения конечной точки пакету потребуется девять шагов. С другой стороны, представим, что каждое ребро e всегда передает пакет, самый дальний от точки получения. Пакету 1 ждать вообще не придется, и он достигнет конечной точки за пять шагов; более того, вы можете убедиться в том, что каждый пакет достигнет своей конечной точки за шесть шагов.

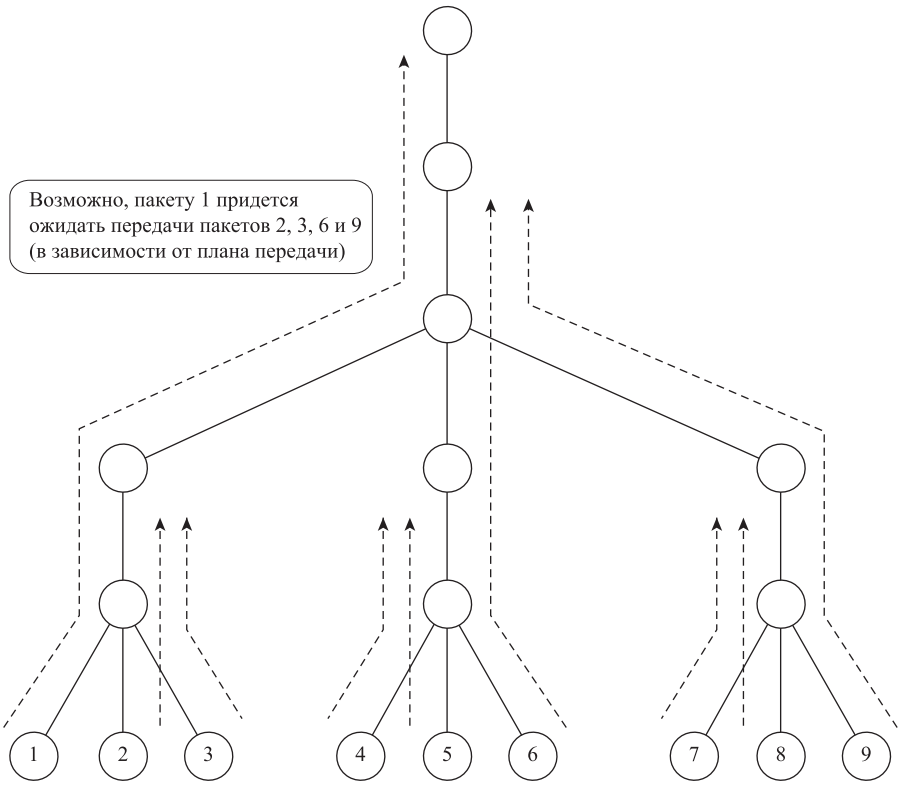


Рис. 13.4. Ситуация, в которой политика управления очередью влияет на время передачи

У древовидной сети на рис. 13.4 существует естественное обобщение, в котором дерево имеет высоту h, а узлы на каждом втором уровне имеют k дочерних узлов. В этом случае с политикой управления очередью, которая всегда передает пакет, ближайший к конечной точке, некоторому пакету потребуется $\Omega(hk)$ для завершения передачи (так как пакет, передаваемый на самое дальнее расстояние, задерживается на $\Omega(k)$ шагов на каждом из $\Omega(h)$ уровней). С другой стороны, с политикой, которая всегда передает пакет, самый дальний от конечной точки, все пакеты достигнут своих конечных точек за $O(h + k)$ шагов. С ростом h и k различия становятся весьма заметными.

Планы и их продолжительность

Перейдем от примеров к теме планирования пакетов и управления очередями в произвольной сети G. Для заданных пакетов 1, 2, ..., N и связанных с ними путей P_1, P_2, ..., P_N план передачи пакетов определяет для каждого ребра e и каждого кванта времени t, какой пакет будет передан по ребру e на шаге t. Конечно, план должен удовлетворять некоторым базовым критериям целостности: за один шаг по любому ребру e передается не более одного пакета, и если пакет i запланирован для передачи по ребру e на шаге t, то ребро e должно входить в путь P_i, и более ранние части плана должны обеспечить достижение e пакетом i. *Продолжительностью* плана называется количество шагов, необходимых для того, чтобы каждый пакет достиг своей конечной точки; требуется найти план с минимальной продолжительностью.

Что может помешать построению плана с низкой продолжительностью? Одним препятствием может стать очень длинный путь, по которому должен пройти некоторый пакет; очевидно, продолжительность не может быть меньше длины этого пути. Другое потенциальное препятствие — одно ребро e, по которому должны быть переданы многие пакеты; так как каждый из пакетов передается по e на отдельном шаге, это тоже устанавливает нижнюю границу для продолжительности. Итак, для максимальной длины d по множеству путей $\{P_1, P_2, ..., P_N\}$ и максимальной загрузки c множества путей (максимальное количество путей, имеющих одно общее ребро) продолжительность передачи не может быть меньше $\max(c,d) = \Omega(c+d)$.

В 1988 году Лейтон, Маггс и Рао (Leighton, Maggs, Rao) доказали следующий неожиданный результат: максимальная длина и максимальная загрузка являются единственными препятствиями для поиска быстрых планов — в том смысле, что всегда существует план с продолжительностью $O(c + d)$. Формулировка утверждения очень проста, но доказать его невероятно сложно; и результатом является только очень сложный метод *построения* такого плана. Вместо того чтобы доказывать это утверждение, мы проанализируем простой алгоритм (также предложенный Лейтоном, Маггсом и Рао), который легко реализуется в распределенной среде и обеспечивает продолжительность, которая отличается в худшую сторону только логарифмическим множителем: $O(c + d \log(mN))$, где m — количество ребер, а N — количество пакетов.

Разработка алгоритма

Простой рандомизированный план

Если каждое ребро просто передает на каждом шаге случайный ожидающий пакет, очевидно, полученный план имеет продолжительность $O(cd)$: в худшем случае пакет может быть заблокирован $c - 1$ другими пакетами на каждом из d ребер своего пути. Для снижения этой границы необходимо принять меры к тому, чтобы каждый пакет мог проводить в ожидании передачи существенно меньшее количество шагов.

Такая большая граница, как $O(cd)$, может возникнуть из-за очень плохой синхронизации пакетов по отношению друг к другу: блоки из c пакетов одновременно встречаются у одного ребра, а когда «затор» устраняется, то же самое происходит у следующего ребра. Ситуация может показаться аномальной, но следует помнить, что на рис. 13.4 она возникла из-за вполне естественной политики управления очередью. Тем не менее такое нежелательное поведение обусловлено крайне неудачным стечением обстоятельств при синхронизации перемещения пакетов; если ввести в управление пакетами рандомизацию, такое поведение становится маловероятным. Проще всего включить случайный сдвиг по времени выхода пакетов из источника. Даже если через одно ребро должно пройти много пакетов, вряд ли они подойдут одновременно, так как конкуренция за ребра была «сглажена». Сейчас мы покажем, что такая рандомизация при правильной реализации работает достаточно эффективно.

Для начала рассмотрим следующий алгоритм, который на самом деле не работает. В нем задействован параметр r, значение которого будет определено позднее.

```
Каждый пакет i ведет себя следующим образом:
   i выбирает случайную задержку s от 1 до r
   i ожидает у источника s шагов
   i движется вперед на полной скорости, по одному ребру за шаг,
       пока не достигнет конечной точки
```

Если случайные задержки действительно будут выбираться так, что никакие два пакета никогда не «столкнутся» (не подойдут к одному ребру одновременно), то этот план будет работать так, как предполагалось; его продолжительность не превысит суммы r (максимальная исходная задержка) и d (максимальное количество ребер по всем путям). Но если r не будет очень большим, где-то в сети может произойти конфликт и в алгоритме произойдет сбой: два пакета одновременно подойдут к ребру e на одном шаге t, и оба должны будут пересечь e на следующем шаге.

Объединение времени в блоки

Чтобы обойти эту проблему, рассмотрим следующее обобщение стратегии: вместо того, чтобы реализовать план «на полной скорости» на уровне отдельных квантов времени, мы реализуем его на уровне смежных *блоков* квантов.

```
Для параметра b сгруппировать интервалы из b смежных квантов времени в блоки
Каждый пакет i ведет себя следующим образом:
   i выбирает случайную задержку s от 1 до r
   i ожидает у источника s блоков
   i движется вперед по одному ребру за блок,
       пока не достигнет конечной точки
```

Этот план будет работать, если удастся избежать более масштабных конфликтов: не должно быть ситуаций, когда более b пакетов должны подойти к одному ребру e в начале одного блока. Если это произойдет, то по крайней мере один из пакетов не сможет быть переданным по e в следующем блоке. Но если исходные задержки обеспечивают распределение, достаточное для того, чтобы к любому ребру в одном блоке прибывали не более b пакетов, то план будет работать так, как задумано. В этом случае продолжительность не превысит $b(r + d)$–максимального количества блоков $r + d$, умноженного на длину каждого блока b.

(13.47) Обозначим $\mathcal{E}$ событие «более b пакетов должны находиться у одного ребра e в начале одного блока». Если событие $\mathcal{E}$ не происходит, то продолжительность плана не превышает $b(r + d)$.

Наша цель — выбрать значения r и b так, чтобы и вероятность $\Pr[\mathcal{E}]$, и продолжительность $b(r + d)$ были малыми величинами. Это ключевой момент анализа — если нам удастся продемонстрировать это, то (13.47) даст границу продолжительности.

Анализ алгоритма

Чтобы предоставить границу для $\Pr[\mathcal{E}]$, будет полезно разложить это событие на объединение более простых плохих событий для применения границы объединения. Естественное множество плохих событий образуется из раздельного рассмотрения каждого ребра и каждого временного блока: если e — ребро, а t — блок от 1 до $r + d$, мы обозначим Φ_{et} событие «более b пакетов должны быть у ребра e в начале блока t». Очевидно, $E = \cup_{e,t} F_{et}$. Кроме того, если N_{et} — случайная переменная, равная количеству пакетов, запланированных находиться у ребра e в начале блока t, то событие Φ_{et} эквивалентно событию $[N_{et} > b]$.

На следующем шаге анализа случайная переменная N_{et} раскладывается в сумму независимых случайных переменных, принимающих значения 0 и 1, для применения границы Чернова. Задача естественно решается определением X_{eti} равным 1, если пакет i должен быть у ребра e в начале блока t, и 0 в противном случае. Тогда $N_{et} = \sum_i X_{eti}$; и для разных значений i случайные переменные X_{eti} независимы, так как задержки пакетов выбираются независимо. (Разумеется, X_{eti} и $X'_{e\,t\,i}$ с одинаковыми значениями i независимыми *не будут*; но наш анализ не требует суммирования переменных в этой форме.) Заметим, что из r возможных задержек, которые может выбрать пакет i, не более чем одна потребует его нахождения у ребра e в блоке t; следовательно, $E[X_{eti}] \le 1/r$. Кроме того, максимум c пакетов имеют пути, включающие e; и если i не входит в их число, то очевидно, $E[X_{eti}] = 0$. Следовательно,

$$E[N_{et}] = \sum_i E[X_{eti}] \le \frac{c}{r}.$$

Теперь все готово для применения границы Чернова (13.42), так как N_{et} является суммой независимых случайных переменных X_{eti}, принимающих значения 0 и 1. Эти величины похожи на те, которые встречались при анализе задачи со случайным распределением m заданий по n процессорам: в том случае каждая составляющая случайная переменная имела ожидание $1/n$, общее ожидание было равно m/n и значение m должно было быть равно $\Omega(n \log n)$, чтобы нагрузка каждого процессора была с высокой вероятностью близка к ожиданию. Подходящая аналогия для текущего случая — r играет роль n, а c играет роль m; во-первых, это выглядит разумно на символическом уровне, с учетом параметров, а во-вторых, соответствует аналогии пакетов с заданиями, а разных временных блоков одного ребра — с разными процессорами, которые могут получать задания. Это наводит на мысль о том, что если мы хотим, чтобы количество пакетов для конкретного ребра в конкретном блоке было близким к ожиданию, то должно выполняться $c = \Omega(r \log r)$.

Такая аналогия работает, разве что придется немного увеличить логарифмическую составляющую, чтобы в конечном итоге работала граница объединения по всем e и всем t. Итак, назначим

$$r = \frac{c}{q \log(mN)},$$

где q — константа, которая будет определена позднее.

Зафиксируем выбор e и t и попробуем ограничить вероятность того, что N_{et} превышает произведение c/r на константу. Определим $\mu = c/r$ и заметим, что $E[N_{et}] \le \mu$, что позволяет применить границу Чернова (13.42). Выберем $\delta = 2$, чтобы $(1+\delta)\mu = \frac{3c}{r} = 3q$, и используем как верхнюю границу в выражении $\Pr\left[N_{et} > \frac{3c}{r}\right] = \Pr\left[N_{et} > (1+\delta)\mu\right]$. Теперь, применяя (13.42), имеем

$$\Pr\left[N_{et} > \frac{3c}{r}\right] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu} = \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu} = \left(\frac{e}{3}\right)^{(1+\delta)\mu} = \left(\frac{e}{3}\right)^{3c/r} = \left(\frac{e}{3}\right)^{3q\log(mN)} = \frac{1}{(mN)^z}.$$

где z — константа, которую можно сделать сколь угодно большой соответствующим выбором константы q.

Из этих вычислений видно, что будет безопасно присвоить $b = 3c/r$, потому что в этом случае событие $\mathcal{F}_{et}$ «$N_{et} > b$» будет иметь очень малую вероятность для всех вариантов выбора e и t. Всего есть m разных вариантов для e и $d + r$ разных вариантов для t; заметим, что $d + r \le d + c - 1 \le N$. Получаем

$$\Pr[\mathcal{E}] = \Pr\left[\bigcup_{e,t} \mathcal{F}_{et}\right] \le \sum_{e,t} \Pr[\mathcal{F}_{et}] \le mN \cdot \frac{1}{(mN)^z} = \frac{1}{(mN)^{z-1}}.$$

Из формулы видно, что значение можно сделать сколь угодно малым за счет выбора достаточно большого z.

Из нашего выбора параметров b и r, в сочетании с (13.44), можно сделать следующий вывод:

(13.48) С высокой вероятностью продолжительность плана для пакетов составляет $O(c + d \log (mN))$.

Доказательство. Только что было показано, что вероятность плохого события E очень мала: она не превышает $(mN)^{-(z-1)}$ для сколь угодно большой константы z. А при условии, что E не происходит, (13.47) сообщает, что продолжительность плана ограничивается величиной

$$b(r+d) = \frac{3c}{r}(r+d) = 3c + d \cdot \frac{3c}{r} = 3c + d(3q \log(mN)) = O(c + d \log(mN)) . \blacksquare$$

13.12. Основные вероятностные определения

Для многих (хотя, безусловно, не всех) областей применения рандомизированных алгоритмов достаточно работать с вероятностями, определяемыми по конечным множествам; анализировать такие ситуации оказывается намного проще, чем размышлять о вероятностях по произвольным множествам. Итак, для начала мы ограничимся только этим частным случаем, а в конце раздела рассмотрим заново все концепции на более общем уровне.

Конечные вероятностные пространства

Всем нам интуитивно понятны выражения типа «Если подбросить монетку, то «орел» выпадет с вероятностью 1/2» или «При броске кубика вероятность выпадения 6 составляет 1/6». Начнем с описания математической инфраструктуры, которая позволяет точно анализировать подобные утверждения. Эта инфраструктура хорошо подходит для систем с четко определенными наборами результатов, таких как броски монеток и кубиков; в то же время мы избежим длинных и непростых философских аспектов, возникающих при попытке моделирования утверждений вида «Вероятность того, что завтра пойдет дождь, равна 20 %». К счастью, в большинстве алгоритмических задач ситуация определяется так же четко, как при бросках кубиков и монеток, — разве что масштабнее и сложнее.

Чтобы иметь возможность вычислять вероятности, введем понятие конечного вероятностного пространства. (Не забывайте, что пока речь идет только о конечных множествах.) Конечное вероятностное пространство определяется нижележащим *пространством выборки*, которое состоит из всех возможных результатов рассматриваемого процесса. Каждой точке пространства выборки ставится в соответствие неотрицательная *вероятностная мера* $p(i) \geq 0$; единственное ограничение для вероятностных мер заключается в том, что их сумма должна быть равна 1,

то есть $\sum_{i\in\Omega} p(i) = 1$. *Событие* $\mathcal{E}$ определяется как произвольное подмножество Ω (то есть просто как множество результатов, образующих это событие), а *вероятность* события определяется как сумма вероятностных мер всех точек в $\mathcal{E}$, то есть

$$\Pr[\mathcal{E}] = \sum_{i\in\mathcal{E}} p(i).$$

Во многих ситуациях, которые мы будем рассматривать, все точки пространства выборки имеют одинаковую вероятностную меру, а вероятность события $\mathcal{E}$ вычисляется как отношение его размера к размеру Ω; то есть в этом частном случае $\Pr[\mathcal{E}] = |\mathcal{E}|/|\Omega|$. Дополняющее событие $\overline{\mathcal{E}}$ будет обозначаться $\Omega - \mathcal{E}$; заметим, что $\Pr\left[\overline{\mathcal{E}}\right] = 1 - \Pr[\mathcal{E}]$.

Таким образом, точки в пространстве выборки и соответствующие им вероятности образуют полное описание рассматриваемой системы; мы собираемся вычислять вероятности событий — подмножеств пространства выборки. Скажем, для представления одного броска «правильной» монетки можно определить пространство выборки Ω = {`heads`, `tails`} и присвоить p(`heads`) = p(`tails`) = 1/2. Чтобы смоделировать несимметричную монетку, в которой «орел» выпадает вдвое чаще «решки», достаточно определить вероятностные меры p(`heads`) = 2/3 и p(`tails`) = 1/3. Даже в этом простом примере важно заметить, что определение вероятностных мер является частью задачи; мы указываем, симметрична монетка или нет, при определении условий, а не выводим это из неких дополнительных данных.

Рассмотрим более сложный пример, который можно было бы назвать *задачей выбора идентификаторов*. Допустим, в распределенной системе существуют n процессов $p_1, p_2, ..., p_n$, каждый из которых выбирает для себя случайный идентификатор из пространства всех k-разрядных строк. Каждый процесс выбирает свой идентификатор одновременно с другими процессами, поэтому решения принимаются независимо. Если рассматривать каждый идентификатор как элемент, выбираемый из множества {0, 1, 2, ..., $2k - 1$} (рассматривая числовое значение идентификатора как число в двоичной записи), пространство выборки Ω можно представить как множество всех n-кортежей целых чисел из диапазона от 0 до $2^k - 1$. В этом случае пространство выборки состоит из $(2k)n = 2kn$ точек, каждая из которых имеет вероятностную меру 2^{-kn}.

Предположим, нас интересует вероятность того, что процессы p_1 и p_2 выберут одинаковые идентификаторы. Это событие $\mathcal{E}$ представляется подмножеством, состоящим из всех n-кортежей из Ω с совпадающими первыми двумя координатами. Количество таких n-кортежей равно $2k(n - 1)$; для координат с 3 по n можно выбрать любое значение, затем для координаты 2 тоже выбирается любое значение, но при выборе координаты 1 свобода полностью отсутствует. Таким образом, получаем

$$\Pr[\mathcal{E}] = \sum_{i\in\mathcal{E}} p(i) = 2^{k(n-1)} \cdot 2^{-kn} = 2^{-k}.$$

Конечно, это соответствует нашим интуитивным представлениям о вероятности: для процесса p_2 можно выбрать любой идентификатор, после чего для процесса p_1 останется всего 1 вариант с совпадением имен из 2^k. Стоит заметить, что эти рассуждения всего лишь компактно описывают приведенные выше вычисления.

Условная вероятность и независимость

Если рассматривать вероятность события $\mathcal{E}$ как шанс на то, что событие E произойдет, также может возникнуть вопрос о его вероятности при наличии дополнительной информации. Для другого события $\mathcal{F}$ с положительной вероятностью *условная вероятность* $\mathcal{E}$ для заданного $\mathcal{F}$ определяется как

$$\Pr[\mathcal{E} \mid \mathcal{F}] = \frac{\Pr[\mathcal{E} \cap \mathcal{F}]}{\Pr[\mathcal{F}]}.$$

Это определение также интуитивно понятно, потому что оно ищет ответ на следующий вопрос: какую часть пространства выборки, состоящего из $\mathcal{F}$ (событие, о наступлении которого нам «известно»), занимает $\mathcal{E}$?

Условные вероятности часто используются при анализе $\Pr[\mathcal{E}]$ для некоторого сложного события $\mathcal{E}$. Предположим, каждое из событий $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_k$ имеет положительную вероятность, и все эти события образуют разбиение пространства выборки; другими словами, каждый результат в пространстве выборки принадлежит ровно одному из них, так что $\sum_{j=1}^{k} \Pr[\mathcal{F}_j] = 1$. Теперь предположим, что значения $\Pr[\mathcal{F}_j]$ известны и мы также можем определить $\Pr[\mathcal{E} \mid \mathcal{F}_j]$ для всех $j = 1, 2$, ..., k. Следовательно, вероятность события $\mathcal{E}$ можно определить в предположении о том, что произошло любое из событий Φ_j. Тогда $\Pr[\mathcal{E}]$ вычисляется по следующей простой формуле:

$$\Pr[\mathcal{E}] = \sum_{j=1}^{k} \Pr[\mathcal{E} \mid \mathcal{F}_j] \cdot \Pr[\mathcal{F}_j].$$

Чтобы обосновать эту формулу, развернем ее правую часть:

$$\sum_{j=1}^{k} \Pr[\mathcal{E} \mid \mathcal{F}_j] \cdot \Pr[\mathcal{F}_j] = \sum_{j=1}^{k} \frac{\Pr[\mathcal{E} \cap \mathcal{F}_j]}{\Pr[\mathcal{F}_j]} \cdot \Pr[\mathcal{F}_j] = \sum_{j=1}^{k} \Pr[\mathcal{E} \cap \mathcal{F}_j] = \Pr[\mathcal{E}].$$

Независимые события

Два события называются *независимыми*, если информация об исходе одного из них не влияет на оценку правдоподобия другого. Например, одно из конкретных определений может выглядеть так: события $\mathcal{E}$ и $\mathcal{F}$ объявляются независимыми, если $\Pr[\mathcal{E} \mid \mathcal{F}] = \Pr[\mathcal{E}]$ и $\Pr[\mathcal{F} \mid \mathcal{E}] = \Pr[\mathcal{F}]$. (Будем считать, что оба события имеют положительную вероятность; в противном случае понятие независимости интереса

не представляет.) Если выполняется одно из этих двух равенств, то должно выполняться и второе, по следующей причине: если $\Pr[\mathcal{E} \mid \mathcal{F}] = \Pr[\mathcal{E}]$, то

$$\frac{\Pr[\mathcal{E} \cap \mathcal{F}]}{\Pr[\mathcal{F}]} = \Pr[\mathcal{E}],$$

а значит, $\Pr[\mathcal{E} \cap \mathcal{F}] = \Pr[\mathcal{E}] \cdot \Pr[\mathcal{F}]$, из чего также следует другое равенство.

Как выясняется, немного проще принять эту эквивалентную формулировку в качестве рабочего определения независимости. Формально события $\mathcal{E}$ и $\mathcal{F}$ называются независимыми, если $\Pr[\mathcal{E} \cap \mathcal{F}] = \Pr[\mathcal{E}] \cdot \Pr[\mathcal{F}]$.

Формулировка с произведением приводит к следующему естественному обобщению. Совокупность событий $E_1, E_2, ..., E_n$ называется независимой, если для каждого набора индексов $I \subseteq \{1, 2, ..., n\}$

$$\Pr\left[\bigcap_{i \in I} \mathcal{E}_i\right] = \prod_{i \in I} \Pr[\mathcal{E}_i].$$

Важно заметить следующее: чтобы проверить большое множество событий на независимость, недостаточно убедиться в том, что каждая пара независима. Предположим, мы бросаем три независимые симметричные монетки: если E_i — событие выпадения «орла» на i-й монетке, то события E_1, E_2, E_3 независимы и каждое из них имеет вероятность 1/2. Теперь обозначим A событие «на монетках 1 и 2 выпали одинаковые значения»; B — событие «на монетках 2 и 3 выпали одинаковые значения»; C — событие «на монетках 1 и 3 выпали разные значения». Легко убедиться в том, что каждое из этих событий имеет вероятность 1/2, а пересечение любых двух событий имеет вероятность 1/4. Таким образом, любая пара событий, выбранная из A, B, C, независима. При этом множество все трех событий A, B, C независимым не является, так как $Pr[A \cap B \cap C] = 0$.

Граница объединения

Предположим, имеется множество событий $E_1, E_2, ..., E_n$ и нас интересует вероятность наступления *каких-либо* из этих событий; другими словами, нас интересует вероятность $\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right]$. Если все события являются попарно непересекающимися, то вероятностная мера их объединения попросту формируется из отдельных вкладов каждого события.

(13.49) Допустим, имеются события $E_1, E_2, ..., E_n$, такие что $\mathcal{E}_i \cap \mathcal{E}_j = \phi$ для каждой пары. Тогда

$$\Pr\left[\bigcup_{i=1}^{n} \mathcal{E}_i\right] = \sum_{i=1}^{n} \Pr[\mathcal{E}_i].$$

В общем случае множество событий $E_1, E_2, ..., E_n$ может содержать достаточно сложные перекрытия. В таком случае равенство (13.49) уже не выполняется; из-за

перекрытий между событиями вероятностная мера точки, которая учитывается один раз в левой части, будет учтена один *или более* раз в правой части (рис. 13.5). Это означает, что для общего множества событий равенство в (13.49) ослабляется до неравенства; и этот факт является содержанием границы объединения. Формулировка границы объединения уже приводилась в виде (13.2), но здесь мы приведем ее снова для сравнения с (13.49).

(13.50) (Граница объединения) Для заданных событий E_1, E_2, ..., E_n имеем $\Pr\left[\bigcup_{i=1}^{n}\mathcal{E}_i\right] \le \sum_{i=1}^{n}\Pr[\mathcal{E}_i]$.

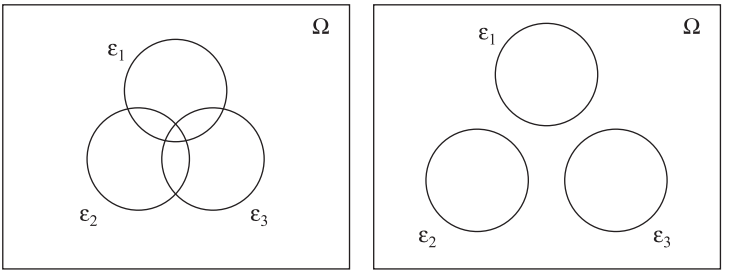


Рис. 13.5. Граница объединения: вероятность объединения максимизируется для неперекрывающихся событий

Несмотря на безобидный внешний вид, граница объединения оказывается неожиданно мощным инструментом для анализа рандомизированных алгоритмов. Ее мощь в основном обусловлена одним часто используемым приемом анализа рандомизированных алгоритмов. Если рандомизированный алгоритм должен выдавать правильный результат с высокой вероятностью, мы сначала определяем множество «плохих событий» E_1, E_2,..., E_n со следующим свойством: если ни одно из плохих событий не произошло, то алгоритм выдает правильный ответ. Другими словами, если обозначить $\mathcal{F}$ событие неудачи при выполнении алгоритма, то

$$\Pr[\mathcal{F}] \le \Pr\left[\bigcup_{i=1}^{n}\mathcal{E}_i\right].$$

Точно вычислить вероятность объединения сложно, поэтому мы применяем границу объединения и приходим к следующему выводу:

$$\Pr[\mathcal{F}] \le \Pr\left[\bigcup_{i=1}^{n}\mathcal{E}_i\right] \le \sum_{i=1}^{n}\Pr[\mathcal{E}_i].$$

Итак, если алгоритм действительно приводит к успеху с очень высокой вероятностью и плохие события были тщательно выбраны, то каждая из вероятностей $\Pr[\mathcal{E}_i]$ будет настолько мала, что малой будет даже их сумма (а следовательно, и наша завышенная оценка вероятности неудачи). В этом заключается вся суть происходящего: сложное событие (неудача при выполнении алгоритма) раскладывается на множество простых событий, вероятности которых легко вычисляются.

Простой пример поможет лучше понять рассматриваемую стратегию. Вспомните задачу выбора идентификаторов, описанную ранее в этом разделе, в которой группа процессов пытается выбрать случайный идентификатор. Предположим, в системе существует 1000 процессов, каждый из которых выбирает 32-разрядный идентификатор, и нас интересует вероятность того, что двум процессам будут назначены одинаковые идентификаторы. Можно ли утверждать, что такое стечение обстоятельств маловероятно? Для начала обозначим это событие $\mathcal{F}$. Хотя точно вычислить $Pr[\mathcal{F}]$ не так уж и сложно, проще определить границу. Событие $\mathcal{F}$ в действительности представляет собой объединение $\binom{1000}{2}$ «атомарных» событий; это события E_{ij}, в которых процессам p_i и p_j назначаются одинаковые идентификаторы. Легко убедиться в том, что $\mathcal{F} = \cup_{i<j} \mathcal{E}_{ij}$. Для любого $i \neq j$ $\Pr[\mathcal{E}_{ij}] = 2^{-32}$, что было обосновано в одном из предыдущих примеров. Применяя границу объединения, получаем

$$\Pr[\mathcal{F}] \leq \sum_{i,j} \Pr[\mathcal{E}_{ij}] = \binom{1000}{2} \cdot 2^{-32}.$$

Значение $\binom{1000}{2}$ не больше полумиллиона, а 2^{32} (немногим более) 4 миллиардов, так что вероятность не превышает $\frac{0,5}{4000} = 0,000125$.

Бесконечные пространства выборки

До сих пор мы имели дело только с конечными вероятностными пространствами. Тем не менее в некоторых разделах этой главы рассматриваются ситуации, в которых случайный процесс может выполняться сколь угодно долго и не может быть нормально описан пространством выборки конечного размера. Для таких ситуаций необходимо определить более общую концепцию вероятностного пространства. Описание содержит много технических подробностей, и отчасти мы приводим его ради полноты: хотя некоторые из рассматриваемых примеров требуют использования бесконечных пространств выборки, ни в одном из них не задействована вся мощь формальных конструкций, описанных в этом разделе.

При переходе к бесконечным пространствам выборки приходится проявлять большую осторожность при определении вероятностной функции. Мы не можем просто связать с каждой точкой пространства выборки Ω вероятностную меру, а затем вычислить вероятность нужной совокупности суммированием. По причинам, в которые мы не будем углубляться, даже если просто разрешить рассматривать каждое подмножество Ω как событие, вероятность которого можно вычислить, это может привести к неприятностям. Обобщенное вероятностное пространство состоит из трех компонентов:

(i) Пространство выборки Ω.

(ii) Набор *S* подмножеств Ω; только для этих событий разрешено вычисление вероятностей.

(iii) Вероятностная функция *Pr*, отображающая события из *S* на вещественные числа из диапазона [0,1].

Совокупность *S* допустимых событий может быть любым семейством множеств, удовлетворяющим следующим основным свойствам замыканий: пустое множество и полное пространство выборки Ω принадлежат *S*; если $\mathcal{E} \in S$, то $\overline{\mathcal{E}} \in S$ (замыкание в отношении дополнения), и если $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots \in S$, то $\bigcup_{i=1}^{\infty} \mathcal{E}_i \in S$ (замыкание в отношении счетного объединения). Вероятностной функцией *Pr* может быть любая функция из *S* в [0,1], удовлетворяющая следующим базовым свойствам непротиворечивости: $\Pr[\mathcal{F}] = 0$, $\Pr[\Omega] = 1$, $\Pr[\mathcal{E}] = 1 - \Pr\left[\overline{\mathcal{E}}\right]$ и граница объединения для непересекающихся событий (13.49) должна выполняться даже для счетных объединений — если $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots \in S$ являются попарно непересекающимися, то

$$\Pr\left[\bigcup_{i=1}^{\infty} \mathcal{E}_i\right] = \sum_{i=1}^{\infty} \Pr[\mathcal{E}_i].$$

Так как Pr уже не строится на базе более общего понятия вероятностной меры, (13.49) из теоремы превращается в обязательное свойство *Pr*.

Как правило, бесконечные пространства выборки возникают в нашем контексте по следующей причине: имеется алгоритм, который принимает серию случайных решений из фиксированного конечного множества возможностей; а поскольку алгоритм может выполняться сколь угодно долго, он может принять произвольно большое количество решений. По этой причине мы рассматриваем пространства выборки Ω, которые строятся следующим образом: мы начинаем с конечного множества символов $X = \{1,2, ..., n\}$ и назначаем вес $w(i)$ каждому символу $i \in X$. Затем Ω определяется как множество всех бесконечных последовательностей символов из *X* (с возможными повторениями). Итак, типичный элемент Ω имеет вид x_1, x_2, x_3, ..., где все $x_i \in X$.

Простейший тип событий, интересующих нас, определяется так: точка $\omega \in \Omega$ начинается с заданной конечной последовательности символов. Таким образом, для конечной последовательности $\sigma = x_1 x_2 \ldots x_s$ длины *s* *префиксное событие*, связанное с σ, определяется как множество всех точек Ω, для которых первые *s* символов образуют последовательность σ. Обозначим это событие E_σ и определим его вероятность $\Pr[E_\sigma] = \omega(x_1)\omega(x_2)\cdots\omega(x_s)$.

Доказать следующий факт определенно нелегко.

(13.51) Существует вероятностное пространство (Ω, *S*, Pr), удовлетворяющее необходимым условиям замыкания и непротиворечивости, для которого Ω — пространство выборки, определенное выше, $\mathcal{E}_\sigma \in S$ для каждой конечной последовательности σ, и $\Pr[E_\sigma] = \omega(x_1)\omega(x_2)\cdots\omega(x_s)$.

Руководствуясь этим фактом, замыканием S в отношении дополнения и счетного объединения, а также непротиворечивостью Pr в отношении этих операций, мы можем вычислить вероятности практически любого «разумного» подмножества Ω.

В бесконечном пространстве Ω с событиями и вероятностями, определенными так, как показано выше, встречается феномен, не типичный для конечных пространств выборки. Предположим, множество X, используемое для генерирования Ω, равно {0,1}, а $w(0) = w(1) = 1/2$. Пусть E — множество, состоящее из всех последовательностей, содержащих как минимум один элемент, равный 1. (Будем считать, что в E не входит последовательность из одних нулей.) Заметим, что E является событием в S, так как σ_i можно определить как последовательность из $i - 1$ нулей, за которым следует 1, и что $\mathcal{E} = \bigcup_{i=1}^{\infty} \mathcal{E}_{\sigma_i}$. Кроме того, все события $\mathcal{E}_{\sigma_i}$ являются попарно непересекающимися, поэтому

$$\Pr[\mathcal{E}] = \sum_{i=1}^{\infty} \Pr[\mathcal{E}_{\sigma_i}] = \sum_{i=1}^{\infty} 2^{-i} = 1.$$

А теперь феномен, о котором упоминалось выше: событие может иметь вероятность 1 даже в том случае, если оно не равно всему пространству выборки Ω. Аналогичным образом $\Pr[\overline{\mathcal{E}}] = 1 - \Pr[\mathcal{E}] = 0$, то есть событие может иметь вероятность 0 даже в том случае, если оно не равно пустому множеству. В этих результатах нет ничего странного; в некотором смысле они необходимы, если мы хотим, чтобы вероятности, определяемые на бесконечных множествах, имели смысл. Просто в таких ситуациях необходимо отделять концепцию события с вероятностью 0 от интуитивного представления о том, что событие «невозможно».

Упражнения с решениями

Упражнение с решением 1

Допустим, в здании установлено несколько компактных маломощных устройств, способных обмениваться данными на небольших расстояниях по беспроводной связи. Для простоты будем считать, что радиус действия каждого устройства достаточен для взаимодействия с d другими устройствами. Тогда систему беспроводных подключений между устройствами можно смоделировать ненаправленным графом $G = (V, E)$, в котором каждый узел инцидентен ровно d ребрам.

Некоторые узлы требуется оснастить более сильным передатчиком, который может использоваться для отправки данных на базовую станцию. Установка такого передатчика на всех узлах гарантирует, что данные смогут передаваться всеми узлами, но того же эффекта можно достичь и при меньшем количестве передатчиков. Допустим, нам удастся найти подмножество S узлов, обладающих тем свойством, что каждый узел в $V–S$ имеет смежный узел в S. Такое множество

будет называться *доминирующим*, потому что оно «доминирует» над всеми остальными узлами в графе. Если установить передатчики только на узлах доминирующего множества, данные все равно можно будет получать со всех узлов: любой узел $u \notin S$ сможет выбрать соседа $v \in S$ и отправить свои данные v, чтобы узел v передал данные базовой станции.

Вопрос в том, как найти доминирующее множество S с минимально возможным размером, потому что это сведет к минимуму количество необходимых передатчиков. Эта задача является *NP*-сложной; доказательство ее составляет суть упражнения 29 в главе 8. (Также обратите внимание на различия между доминирующими множествами и вершинными покрытиями: в доминирующем множестве может присутствовать ребро (u, v), в котором ни u, ни v не входят в множество S — при условии, что у u и v есть соседи в S. Таким образом, например, граф из трех узлов, соединенных ребрами, имеет доминирующее множество с размером 1, но ни одного вершинного покрытия с размером 1.)

Несмотря на *NP*-полноту задачи, в таких ситуациях важно найти доминирующее множество как можно меньшего размера, даже если оно не оптимально. Оказывается, простая стратегия рандомизации может быть достаточно эффективной. Вспомните, что в графе G каждый узел инцидентен ровно d ребрам. Очевидно, что любое доминирующее множество должно иметь размер не менее $\frac{n}{d+1}$, потому что каждый узел, включенный в доминирующее множество, может «обслуживать» только себя самого и d своих соседей. Требуется показать, что случайный выбор узлов действительно подведет нас достаточно близко к этой простой нижней границе.

Покажите, что для некоторой константы c множество из $\frac{cn\log n}{d+1}$ узлов, случайно и равномерно выбираемых из G, с высокой вероятностью образует доминирующее множество. (Иначе говоря, это полностью случайное множество с большой вероятностью образует доминирующее множество, которое всего в $O(\log n)$ раз больше простой нижней границы $\frac{n}{d+1}$.

Решение

Пусть $k = \frac{cn\log n}{d}$, где константа c будет выбрана позднее, когда мы получим более ясное представление о происходящем. Обозначим E событие «случайно выбранные k узлов образуют доминирующее множество для G». Чтобы упростить анализ, мы рассмотрим модель, в которой узлы выбираются по одному и один узел может встречаться дважды (если он будет выбран дважды в результате серии случайных решений).

Теперь нужно показать, что при достаточно больших c (а следовательно, и k) $\Pr[\mathcal{E}]$ близко к 1. Но E — исключительно сложное событие, поэтому для начала мы разобьем его на более простые события, вероятности которых проще анализировать.

Начнем с определения некоторых терминов. Узел *w доминирует* над узлом *v*, если w является соседом *v*, или *w* = *v*. Множество *S* доминирует над узлом *v*, если некоторый элемент *S* доминирует над *v*. (Эти определения позволяют сказать, что доминирующее множество — просто множество узлов, доминирующее над каждым узлом в графе.) Обозначим $D[v, t]$ событие «выбранный *t*-й случайный узел доминирует над узлом *v*». Вероятность этого события вычисляется очень просто: из *n* узлов графа должен быть выбран узел *v* или один из *d* его соседей, поэтому

$$\Pr[D[v, t]] = \frac{d+1}{n}.$$

Пусть D_v — событие «случайное множество состоящее из всех *k* выбранных узлов, доминирует над *v*. Тогда

$$D_v = \bigcup_{t=1}^{k} D[v, t].$$

Как уже говорилось, для независимых событий проще работать с пересечениями (где вероятности можно просто перемножить), чем с объединениями. Итак, вместо того, чтобы анализировать Δ_v, мы рассмотрим дополняющее «событие неудачи» $\overline{D_v}$ «ни один узел в случайном множестве не доминирует над *v*». Чтобы ни один узел не доминировал над *v*, ни один вариант выбора не должен доминировать, поэтому мы имеем

$$\overline{D_v} = \bigcap_{t=1}^{k} \overline{D[v, t]}.$$

Так как события $\overline{D[v,t]}$ независимы, вероятность в правой части можно вычислить умножением всех отдельных вероятностей; следовательно

$$\Pr\left[\overline{D_v}\right] = \prod_{t=1}^{k} \Pr\left[\overline{D[v, t]}\right] = \left(1 - \frac{d+1}{n}\right)^k.$$

Так как $k = \frac{cn\log n}{d+1}$, последнее выражение можно записать в виде

$$\left(1 - \frac{d+1}{n}\right)^k = \left[\left(1 - \frac{d+1}{n}\right)^{n/(d+1)}\right]^{c\log n} \le \left(\frac{1}{e}\right)^{c\log n},$$

где неравенство следует из утверждения (13.1), приведенного ранее в этой главе.

Мы еще не указали основание для логарифма, используемого для определения *k*, но похоже, основание *e* будет хорошим вариантом. С ним последнее выражение упрощается до

$$\Pr\left[\overline{D_v}\right] \le \left(\frac{1}{e}\right)^{c \ln n} = \frac{1}{n^c}.$$

Работа почти закончена. Мы показали, что для каждого узла v вероятность того, что случайное множество не будет доминирующим, не превышает значение n^{-c}, которое можно сократить до очень малой величины, сделав c достаточно большим. Теперь вспомним исходное событие E (случайное множество является доминирующим). Оно не происходит в том, и только в том случае, если не происходит одно из событий D_v, так что $\overline{\mathcal{E}} = \bigcup_v \overline{D_v}$. Следовательно, из границы объединения (13.2) получаем

$$\Pr\left[\overline{\mathcal{E}}\right] \le \sum_{v \in V} \Pr\left[\overline{D_v}\right] \le n \cdot \frac{1}{n^c} = \frac{1}{n^{c-1}}.$$

Если просто выбрать $c = 2$, вероятность становится равной $\frac{1}{n}$, что намного меньше 1. Таким образом, с высокой вероятностью событие E выполняется, а случайный выбор узлов действительно является доминирующим множеством.

Интересно заметить, что вероятность успеха как функция k проявляет поведение, очень сходное с тем, которое мы видели в примере с разрешением конфликтов из раздела 13.1. Выбора $k = \Theta(n/k)$ достаточно для того, чтобы гарантировать доминирование над каждым отдельным узлом с постоянной вероятностью. Впрочем, для получения полезного результата из границы объединения этого недостаточно. Увеличение k с логарифмическим множителем позволило повысить вероятность доминирования над каждым узлом до величины, очень близкой к 1, когда можно воспользоваться границей объединения.

Упражнение с решением 2

Предположим, имеется множество из n переменных $x_1, x_2, ..., x_n$, принимающих значения из множества {0, 1}. Также имеется множество из k равенств; r-е равенство имеет форму

$$(x_i + x_j) \bmod 2 = b_r$$

для некоторого выбора двух разных переменных x_i, x_j и для некоторого значения b_r, равного либо 0, либо 1.

Таким образом, каждое равенство определяет, четна или нечетна сумма двух переменных.

Рассмотрим задачу нахождения присваивания значений переменным, которая максимизирует количество выполняемых равенств (то есть тех, для которых условие действительно истинно). Задача является *NP*-сложной, хотя доказывать это не нужно.

Предположим, даны следующие равенства:

$$(x_1 + x_2) \bmod 2 = 0$$

$$(x_1 + x_3) \bmod 2 = 0$$

$$(x_2 + x_4) \bmod 2 = 1$$

$$(x_3 + x_4) \bmod 2 = 0$$

для четырех переменных $x_1, ..., x_4$. Тогда возможно показать, что никакое присваивание значений переменным не выполняет все равенства одновременно, но присваивание всем переменным 0 выполняет три равенства из четырех.

(a) Пусть c^* — максимально возможное количество равенств, которые могут быть выполнены присваиванием значений переменным. Предложите алгоритм с полиномиальным временем для получения присваивания, выполняющего по крайней мере $\frac{1}{2}c^*$ равенств. При желании алгоритм можно рандомизировать; в этом случае ожидаемое количество выполняемых равенств должно быть не менее $\frac{1}{2}c^*$. В любом случае следует доказать, что алгоритм обеспечивает заявленные гарантии эффективности.

(b) Допустим, ограничение, согласно которому каждое равенство должно содержать ровно две переменные, отменяется; другими словами, теперь каждое равенство просто указывает, что остаток от деления суммы произвольного подмножества переменных на 2 равен заданному значению b_r.

И снова обозначим c^* максимально возможное количество равенств, которые могут быть выполнены присваиванием значений переменным. Предложите алгоритм с полиномиальным временем для получения присваивания, выполняющего по крайней мере $\frac{1}{2}c^*$ равенств. (Как и в предыдущем пункте, алгоритм может быть рандомизированным.) Если вы полагаете, что ваш алгоритм из пункта (a) тоже предоставляет такую гарантию, укажите это и обоснуйте, приведя доказательство гарантий эффективности для более общего случая.

Решение

Вспомним основной вывод простого рандомизированного алгоритма для задачи MAX 3-SAT, приведенный ранее в этой главе: в задаче на выполнение условий случайное присваивание значений переменным может быть неожиданно эффективным способом выполнения постоянной части всех ограничений.

Попробуем применить этот принцип в текущей задаче, начиная с части (a). Рассмотрим алгоритм, который присваивает всем переменным случайные значения с равномерным распределением. Насколько хорошо работает это случайное присваивание в ожидании?

Как обычно, для получения ответа будет использована линейность ожидания: возьмем X — случайную переменную, обозначающую количество выполненных равенств, и разобьем ее на сумму более простых случайных переменных.

Пусть для некоторого r в диапазоне от 1 до k r-е равенство имеет вид

$$(x_i + x_j) \bmod 2 = b_r.$$

Случайная переменная X_r равна 1, если это равенство выполняется, и 0 в противном случае. $E[X_r]$ — вероятность того, что равенство r выполняется. Из четырех возможных вариантов присваивания равенству i в двух левая часть дает результат 0 mod 2 ($x_i = x_i = 0$ и $x_i = x_j = 1$), а в двух других 1 mod 2 ($x_i = 0$; $x_j = 1$ и $x_i = i$; $x_j = 0$). Следовательно, $E[X_r] = 2/4 = 1/2$.

Из линейности ожидания имеем $E[X] = \sum_r E[X_r] = k/2$. Так как максимальное количество выполняемых равенств c^* не должно превышать k, в ожидании выполняются не менее $c^*/2$ равенств. Следовательно, как и в случае задачи MAX 3-SAT, при простом случайном присваивании выполняется постоянная часть всех ограничений.

В части (b) попробуем рискнуть и воспользоваться тем же алгоритмом. И снова пусть X_r — случайная переменная, которая равна 1, если r-е уравнение выполняется, и 0 в противном случае; X — общее количество выполненных равенств и c^* — оптимум.

Хотелось бы заявить, что $E[X_r] = 1/2$, как и прежде, даже при произвольном количестве переменных в r-м равенстве; другими словами, вероятность того, что равенству будет присвоено правильное значение mod 2, составляет ровно 1/2. Просто записать все случаи, как это было сделано для равенств с двумя переменными, не удастся, поэтому будет использоваться альтернативное обоснование.

Доказать, что $E[X_r] = 1/2$, можно двумя естественными способами. В первом используется прием, встречавшийся в доказательстве (13.25) из раздела 13.6 для хеширования: мы рассматриваем присваивание произвольных значений всем переменным, кроме последней, а затем присваиваем случайное значение последней переменной x. Независимо от того, как были присвоены значения других переменных, присвоить значение x можно двумя способами; нетрудно проверить, что с одним из них равенство выполняется, а с другим нет. Таким образом, независимо от присваивания значений переменным, отличным от x, вероятность присваивания x с выполнением равенства составляет ровно 1/2. Итак, вероятность того, что равенство будет выполнено при случайном присваивании, равна 1/2.

(Как и в доказательстве (13.25), это обоснование можно записать в контексте условных вероятностей. Если E — событие выполнения равенства, а $\mathcal{F}_b$ — событие присваивания переменным, отличным от x, последовательности значений b, то $\Pr[\mathcal{E} \mid \mathcal{F}_b] = 1/2$ для всех b, а следовательно, $\Pr[\mathcal{E}] = \sum_b \Pr[\mathcal{E} \mid \mathcal{F}_b] \cdot \Pr[\mathcal{F}_b] = (1/2)\sum_b \Pr[\mathcal{F}_b] = 1/2$.)

В другом доказательстве просто подсчитывается количество способов, которыми в r-м равенстве может быть достигнута четная или нечетная сумма. Если удастся показать, что эти два числа равны, то вероятность того, что случайное

присваивание выполнит r-е равенство, совпадает с вероятностью получения правильной четности/нечетности суммы, которая равна 1/2.

Собственно, на высоком уровне это доказательство практически эквивалентно предыдущему, не считая того, что базовую счетную задачу удается сформулировать явно. Предположим, r-е равенство состоит из t слагаемых; тогда существуют 2^t возможных вариантов присваивания переменным в этом равенстве. Утверждается, что с 2^{t-1} присваиваний сумма будет четной, а с другими 2^{t-1} — нечетной; это покажет, что $E[X_r]=1/2$. Мы докажем это утверждение индукцией по t. Для $t=1$ используются всего два присваивания, по одному для каждого варианта четности; для $t=2$ доказательство уже приводилось выше для всех $2^2=4$ возможных присваиваний. Теперь предположим, что утверждение выполняется для произвольного значения $t-1$. Тогда существуют ровно 2^{t-1} способов получить четную сумму с t переменными:

- 2^{t-2} способов получения четной суммы для первых $t-1$ переменных (по индукции) с приваиванием 0 t-й переменной, плюс
- 2^{t-2} способов получения нечетной суммы для первых $t-1$ переменных (по индукции) с приваиванием 1 t-й переменной.

Остальные 2^{t-1} присваиваний дают нечетную сумму, и на этом шаг индукции завершается.

Зная, что $E[X_r]=1/2$, мы можем завершить доказательство по аналогии с частью (а): из линейности ожидания следует $E[X]=\sum_r E[X_r]=k/2\geq c^*/2$.

Упражнения

1. Формулировка задачи 3-раскраски предполагает ответ «да/нет», но задачу также можно переформулировать в виде оптимизационной задачи.

   Допустим, имеется граф $G=(V,E)$, каждый узел которого должен быть окрашен в один из трех цветов, даже если не удастся назначить разные цвета каждой паре смежных узлов. Ребро (u, v) будет называться *реализованным*, если u и v назначены разные цвета.

   Рассмотрим 3-раскраску, максимизирующую количество реализованных ребер, и обозначим это количество c^*. Предложите алгоритм с полиномиальным временем, который дает 3-раскраску, реализующую не менее $\frac{2}{3}c^*$ ребер. При желании алгоритм можно рандомизировать; в этом случае ожидаемое количество реализуемых ребер должно быть не менее $\frac{2}{3}c^*$.

2. Рассмотрим округ, в котором проживают 100 000 избирателей. В списке для голосования всего два кандидата: от демократической партии (D) и от республиканской партии (R). Так уж сложилось, что округ населен в основном

сторонниками демократов, так что 80 000 избирателей идут на выборы с намерением голосовать за D, а 20 000 избирателей собираются голосовать за R. Тем не менее правила голосования достаточно сложны, поэтому каждый избиратель независимо от других с вероятностью $\frac{1}{100}$ голосует за другого кандидата — то есть за того, за которого он голосовать не собирался. (Напомним, что в этих выборах участвуют всего два кандидата.)

Случайная переменная X равна количеству голосов за демократического кандидата D при проведении голосования с возможными ошибками. Определите ожидаемое значение X и объясните ход своих рассуждений при выводе этого значения.

3. В разделе 13.1 описан простой распределенный протокол для решения конкретной задачи разрешения конфликтов. Здесь рассматривается другая ситуация с применением рандомизации для разрешении конфликтов, в которой осуществляется распределенное построение независимого множества.

Предположим, имеется система с n процессами. Между некоторыми парами процессов возникает *конфликт* (предполагается, что обоим процессам необходим доступ к общему ресурсу). Для заданного интервала времени требуется спланировать выполнение большого подмножества S процессов (остальные процессы при этом не работают), чтобы никакие два конфликтующих процесса не входили в запланированное множество S. Назовем такое множество S *бесконфликтным*.

Ситуацию можно смоделировать графом $G = (V, E)$, в котором узлы представляют процессы, а ребра соединяют пары процессов, находящихся в конфликте. Легко убедиться в том, что множество процессов S является бесконфликтным в том, и только в том случае, если оно образует независимое множество в G. Это наводит на мысль, что найти бесконфликтное множество S максимального размера для произвольного конфликта G будет сложно (так как обобщенная задача о независимом множестве сводится к этой задаче). Тем не менее мы попробуем найти эвристику для нахождения бесконфликтного множества достаточно большого размера. Также хотелось бы найти простой метод достижения этой цели без централизованного управления: каждый процесс должен обменяться информацией только с небольшим количеством других процессов и затем решить, должен ли он принадлежать множеству S.

В контексте этого вопроса предположим, что у каждого узла в графе G ровно d соседей. (То есть каждый процесс участвует в конфликтах ровно с d другими процессами.)

(а) Рассмотрим следующий простой протокол.

Каждый процесс P_i независимо выбирает случайное значение x_i; с вероятностью $\frac{1}{2}$ переменной x_i присваивается значение 1, и с вероятностью $\frac{1}{2}$ x_i присваивается значение 0. Затем процесс решает войти в множество S в том,

и только в том случае, если он выбирает значение 1, и каждый из процессов, с которыми он участвует в конфликте, выбирает значение 0.

Докажите, что множество S, полученное в результате выполнения этого протокола, является бесконфликтным. Также приведите формулу для ожидаемого размера S в отношении к n (количество процессов) и к d (количество конфликтов на один процесс).

(b) Выбор вероятности $\frac{1}{2}$ в приведенном выше протоколе был достаточно произвольным, и не так уж очевидно, что он должен обеспечить лучшую эффективность системы. В более общей спецификации протокола вероятность $\frac{1}{2}$ заменяется параметром p в диапазоне от 0 до 1 так, как описано ниже.

Каждый процесс P_i независимо выбирает случайное значение x_i; с вероятностью p переменной x_i присваивается значение 1, и с вероятностью $1 - p$ x_i присваивается значение 0. Затем процесс решает войти в множество S в том, и только в том случае, если он выбирает значение 1, и каждый из процессов, с которыми он участвует в конфликте, выбирает значение 0.

В контексте параметров графа G приведите значение p, максимизирующее ожидаемый размер полученного множества S. Предложите формулу для ожидаемого размера S, когда p присваивается это оптимальное значение.

4. Многие одноранговые системы в Интернете строятся на базе *оверлейных сетей*. Вместо использования физической топологии Интернета и выполнения с ней необходимых вычислений такие системы используют протоколы, в которых узлы выбирают совокупности виртуальных «соседей» для определения высокоуровневого графа, структура которого имеет минимальное отношение к структуре используемой физической сети. Затем такая оверлейная сеть используется для обмена данными и сервисами и может быть исключительно гибкой по сравнению с физической сетью, которую трудно изменить в реальном времени для адаптации к изменяющимся условиям.

   Одноранговые сети обычно растут за счет новых участников, которые присоединяются к существующей структуре. Процесс развития сети оказывает специфическое влияние на характеристики общей сети. За последнее время были исследованы простые абстрактные модели развития сетей, которые дают представление о поведении таких процессов в реальных сетях на качественном уровне.

   Рассмотрим простой пример такой модели. Система начинается с одиночного узла v_1, после чего узлы присоединяются к ней один за одним; при присоединении узла выполняется протокол с формированием направленного канала связи с другим узлом, выбранным случайно и равномерно из узлов, уже присутствующих в системе. Конкретнее, если система уже содержит узлы $v_1, v_2, ..., v_{k-1}$ и узел v_k желает присоединиться, он случайным образом выбирает один из узлов $v_1, v_2, ..., v_{k-1}$ и устанавливает связь с ним.

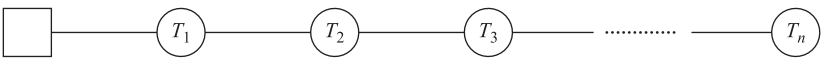
Предположим, процесс повторяется вплоть до формирования системы, состоящей из узлов v_1, v_2, ..., v_n; описанная выше случайная процедура создает направленную сеть, в которой каждый узел, кроме v_1, имеет ровно один выходной канал. С другой стороны, узел может иметь несколько входных каналов, а может не иметь ни одного. Входные каналы v_j представляют все остальные узлы, которые получают доступ к системе через v_j; таким образом, если v_j имеет много входных каналов, нагрузка на него может оказаться достаточно высокой. Чтобы система была сбалансирована, нам хотелось бы, чтобы узлы имели примерно одинаковое количество входных каналов. Однако в описанной ситуации это маловероятно, так как узлы, присоединенные на более ранней стадии процесса, с большей вероятностью будут иметь больше входных каналов, чем узлы, присоединившиеся позднее. Попробуем дать численную оценку этому дисбалансу.

(a) Для заданного случайного процесса, описанного выше, каково ожидаемое количество входных каналов к узлу v_j в такой сети? Приведите точную формулу для n и j и попробуйте выразить эту величину асимптотически (выражением без громоздкого суммирования) с использованием записи $\Theta(\cdot)$.

(b) Часть (a) точно выражает смысл, в котором узлы, присоединяемые ранее, получают «несправедливую» долю соединений в сети. Другой способ численного выражения дисбаланса основан на том факте, что при выполнении этого случайного процесса можно ожидать, что многие узлы останутся без входных каналов.

Предложите формулу для ожидаемого количества узлов без входных каналов в сети, развивающейся по условиям описанной выше модели.

5. Где-то в сельской местности n небольших городков решили подключиться к Интернету по оптоволоконному кабелю с высокой пропускной способностью. Городки обозначаются T_1, T_2, ..., T_n и располагаются вдоль одной длинной долроги, так что город T_i удален на i километров от точки подключения (рис. 13.6).



Точка подключения

Рис. 13.6. Городки T_1, ..., T_n должны решить, как им разделить расходы на покупку кабеля

Кабель стоит довольно дорого — k долларов за километр, то есть общая стоимость всего кабеля составляет kn долларов. Представители городков собираются вместе и обсуждают, как им следует разделить расходы на покупку кабеля.

Наконец, один из городков на дальнем конце дороги вносит следующее предложение.

Предложение A. Разделить затраты поровну между всеми городками, чтобы каждый платил k долларов.

В каком-то смысле предложение честное: все выглядит так, словно каждый городок платит за километр кабеля, который ведет непосредственно к нему.

Но один из представителей городка, расположенного очень близко к точке подключения, резонно указывает, что от более длинного кабеля выигрывают в основном дальние городки, а ближние вполне обошлись бы и коротким кабелем. Поэтому он выдвигает встречное предложение:

> **Предложение Б**. Разделить стоимость так, чтобы вклад городка T_i был пропорционален i, то есть расстоянию до точки подключения.

Представитель другого городка, также очень близкого к точке подключения, указывает, что существует другой способ непропорционального деления, который тоже естествен. Он основан на концептуальном делении кабеля на n «ребер» равной длины e_1, ..., e_n; первое ребро e_1 ведет от точки подключения к T_1, а i-е ($i > 1$) ведет от T_{i-1} к T_i. Заметим, что хотя e_1 используется всеми городками, e_n приносит пользу только последнему городку. Так появляется

> **Предложение В**. Разделить стоимость для каждого ребра e_i по отдельности. Стоимость e_i должна оплачиваться в равной мере городками T_i, T_{i+1}, ..., T_n, так как они находятся «после» e_i.

Итак, вариантов много; какой из них самый справедливый? За ответом на этот вопрос собрание обращается к работе Ллойда Шепли, одного из самых знаменитых экономистов-математиков XX века. Он предложил общий механизм разделения стоимости или прибылей между общими сторонами, который теперь называется *распределением Шепли*. Этот механизм может рассматриваться как определение «предельного вклада» каждой стороны *в предположении, что стороны прибывают в случайном порядке*.

В условиях нашей задачи произойдет следующее: возьмем упорядочение $\mathcal{O}$ городков и предположим, что городки «прибывают» в этом порядке. Предельная стоимость городка T_i в порядке $\mathcal{O}$ определяется следующим образом: если T_i является первым в порядке $\mathcal{O}$, то T_i платит k_i — стоимость всего кабеля на всем отрезке от точки подключения до T_i. В противном случае алгоритм просматривает множество городков, предшествующих T_i в порядке $\mathcal{O}$; допустим, из этих городков T_j находится на наибольшем расстоянии от точки подключения. При прибытии T_i предполагается, что кабель уже проложен до T_j, но не далее. Итак, если $j > i$ (T_j находится дальше T_i), предельная стоимость T_i равна 0, так как кабель уже проходит мимо T_i на своем пути к T_j. С другой стороны, если $j < i$, то предельная стоимость T_i равна $k(i - j)$: это стоимость продления кабеля от T_j к T_i.

(Предположим, $n = 3$, и городки прибывают в порядке T_1, T_3, T_2. Сначала T_1 платит k при прибытии. Затем прибывает T_3, которому достаточно заплатить $2k$ для продления кабеля от T_1. Наконец, при прибытии T_2 платить вообще ничего не нужно, так как кабель уже проложен мимо этого городка до T_3.)

Пусть X_i — случайная переменная, равная предельной стоимости городка T_i при случайном равномерном выборе порядка $\mathcal{O}$ из всех перестановок множества городков. По правилам распределения Шепли величина, которую городок T_i должен внести в общую стоимость кабеля, равна ожидаемому значению X.

Вопрос: какое из трех предложений (если оно есть) обеспечивает такое же распределение стоимостей, как механизм распределения Шепли? Приведите доказательство ответа.

6. Одна из (многих) сложных задач, связанных с расшифровкой генома, может быть сформулирована следующим абстрактным способом. Имеется множество из n маркеров $\{\mu_1, ..., \mu_n\}$ — позиций в хромосоме, требуется вывести линейное упорядочение этих маркеров. Вывод должен удовлетворять множеству k ограничений, каждое из которых задается триплетом (μ_i, μ_j, μ_k) и требует, чтобы маркер μ_j находился между μ_i и μ_k в общем упорядочении). (Обратите внимание: это ограничение не указывает, какой из маркеров μ_i или μ_k должен находиться на первом месте в упорядочении, а лишь требует, чтобы маркер μ_j находился между ними).

   Одновременное выполнение всех ограничений не всегда возможно, поэтому требуется получить упорядочение, выполняющее как можно больше из них. К сожалению, принятие решения о существовании упорядочения, выполняющего не менее k' из k ограничений, является *NP*-сложной задачей (доказывать это не нужно).

   Приведите константу $\alpha > 0$ (не зависящую от n) и алгоритм, обладающий следующим свойством: если возможно выполнить k^* ограничений, то алгоритм выдает упорядочение маркеров, выполняющее не менее αk^* ограничений. Алгоритм может быть рандомизированным; в таком случае он должен выдавать упорядочение, для которого *ожидаемое* количество выполненных ограничений не менее αk^*.

7. В разделе 13.4 был разработан аппроксимирующий алгоритм для задачи MAX 3-SAT с множителем 7/8, в котором предполагалось, что каждое условие содержит составляющие для трех разных переменных. В этой задаче рассматривается похожая задача MAX SAT: для заданного множества условий $C_1, ..., C_k$ по множеству переменных $X = \{x_1, ..., x_n\}$ найти логическое присваивание, выполняющее как можно большее количество условий. Каждое условие содержит минимум один литерал, и все переменные в одном условии различны, но в остальном никакие допущения относительно длины условий не делаются: одни условия могут состоять из многих переменных, а другие содержат всего одну.

   (a) Сначала рассмотрим рандомизированный аппроксимирующий алгоритм, который мы использовали для задачи MAX 3-SAT, с независимым присваиванием каждой переменной истинного или ложного значения с вероятностью 1/2. Покажите, что ожидаемое количество условий, выполненных этим случайным присваиванием, не менее $k/2$, то есть в ожидании выполняется по меньшей мере половина условий. Приведите пример, показывающий, что существуют экземпляры задачи MAX SAT, в которых никакое присваивание не выполняет более половины условий.

   (b) Если имеется условие, состоящее только из одного литерала (например, условие, включающее только x_1 или только $\overline{x_2}$), то оно может быть выполнено только одним способом: нужно присвоить переменной соответствующее

значение. Если есть два условия, одно из которых состоит только из x_i, а другое из его отрицания, возникает довольно очевидное противоречие.

Предположим, в нашем экземпляре нет таких «прямых конфликтов», то есть не существует переменной x_i, для которой условие $C = \{x_i\}$ существует одновременно с условием $C' = \{\overline{x_i}\}$. Измените приведенную выше рандомизированную процедуру, чтобы улучшить коэффициент аппроксимации с 1/2 до минимум 0,6. Другими словами, измените алгоритм так, чтобы ожидаемое количество условий, выполняемых процессом, было не менее $0{,}6k$.

(c) Предложите рандомизированный алгоритм с полиномиальным временем для обобщенной задачи MAX SAT, при котором ожидаемое количество условий, выполняемых алгоритмом, лежит в пределах множителя 0,6 от максимума. (Учтите, что по аналогии с (a) существуют экземпляры, в которых невозможно выполнить более $k/2$ условий; суть в том, что нам хотелось бы иметь эффективный алгоритм, способный в ожидании выполнить до 0,6 части условий от максимума, выполняемого при оптимальном присваивании.)

8. Имеется ненаправленный граф $G = (V, E)$ с n узлами и m ребрами. Для подмножества $X \subseteq V$ запись $G[X]$ будет обозначать подграф, *индуцированный по X*, — то есть граф, множеством узлов которого является X, а множество ребер состоит из всех ребер G, оба конца которых принадлежат X.

   Для заданного натурального числа $k \leq n$ требуется найти множество из k узлов, индуцирующее из G «плотный» подграф. Или в более точной формулировке, предложите алгоритм с полиномиальным временем, который выдает для заданного натурального числа $k \leq n$ множество $X \subseteq V$ из k узлов, обладающее тем свойством, что индуцированный подграф $G[X]$ содержит минимум $\frac{mk(k-1)}{n(n-1)}$ ребер.

   Приведите либо (a) детерминированный алгоритм, либо (b) рандомизированный алгоритм с полиномиальным оиждаемым временем выполнения, который выводит только правильные ответы.

9. Допустим, вы разрабатываете стратегии для продажи товаров на популярном аукционном сайте. В отличие от других аукционных сайтов, на вашем используется *однопроходной аукцион*, в котором каждая ставка либо немедленно (и безвозвратно) принимается, либо отклоняется. Ниже приведена более конкретная схема работы сайта.

   - Продавец выставляет товар для продажи.
   - Покупатели приходят последовательно.
   - При появлении покупатель i делает ставку $b_i > 0$.
   - Продавец немедленно решает, принять эту ставку или нет. Если ставка принята, то товар продается, а будущие покупатели теряют возможность его купить. Если ставка отклонена, то покупатель i уходит, а ставка отменяется; только после этого продавец сможет увидеть заявки будущих покупателей.

Предположим, товар выставлен на продажу и имеются n покупателей с разными ставками. Также предположим, что покупатели приходят в случайном порядке и продавцу известно количество n покупателей. Требуется разработать стратегию, при которой у продавца имеется разумная вероятность принять максимальную из n заявок. Под «стратегией» подразумевается правило, по которому продавец решает, принять или отклонить каждую ставку, на основании исключительно значения n и последовательности ставок, сделанных до настоящего момента.

Например, продавец всегда может принимать первую ставку. В результате вероятность того, что он примет высшую из n заявок, составит всего $1/n$, так как высшая ставка должна оказаться первой из поданных.

Предложите стратегию, при которой продавец принимает высшую из n ставок с вероятностью не менее $1/4$ независимо от значения n. (Для простоты считайте, что число n четно.) Докажите, что ваша стратегия обеспечивает эту вероятностную гарантию.

10. Рассмотрим очень простую систему проведения веб-аукциона, которая работает следующим образом. Имеются n *агентов*; агент i делает ставку b_i, которая является положительным целым числом. Будем считать, что все ставки b_i отличны друг от друга. Агенты появляются в порядке, выбираемом случайным образом с равномерным распределением, каждый делает свою ставку b_i, а система постоянно поддерживает переменную b^*, равную наивысшей из сделанных на данный момент ставок. (Переменная b^* инициализируется 0.)

    Приведите оценку ожидаемого количества обновлений b^* при выполнении этой процедуры как функцию параметров задачи.

    Пример. Допустим, $b_1 = 20$, $b_2 = 25$ и $b_3 = 10$, а агенты прибывают в порядке 1, 3, 2. Тогда b^* обновляется для заявок агентов 1 и 2, но не для 3.

11. Алгоритмы распределения нагрузки для параллельных или распределенных систем стремятся равномерно распределить группу вычислительных заданий по нескольким машинам. Это делается для того, чтобы избежать пиковых нагрузок на отдельных машинах. Если централизованная координация возможна, то нагрузка может быть распределена почти идеально. Но что, если задания поступают из разных источников, которые не могут координироваться? Как было показано в разделе 13.10, задания можно распределять между машинами случайным образом, надеясь, что рандомизация поможет предотвратить несбалансированность. Разумеется, такое решение в общем случае уступает идеальному централизованному решению, но оно может быть вполне эффективным. Попробуем проанализировать некоторые вариации и расширения простой эвристики распределения нагрузки, описанной в разделе 13.10.

    Предположим, имеются k машин и k заданий, ожидающих обработки. Каждое задание назначается на одну из k машин независимо и случайным образом (с равной вероятностью для всех машин).

(a) Обозначим $N(k)$ ожидаемое количество машин, не получивших ни одного задания, так что $N(k)/k$ — ожидаемая часть бездействующих машин. Чему равен предел $\lim_{k\to\infty} N(k)/k$? Приведите доказательство.

(b) Допустим, машины не могут ставить в очередь лишние задания, так что если случайное распределение назначает на машину M больше одного задания, то M выполняет первое из полученных заданий и отклоняет все остальные. Пусть $R(k)$ — ожидаемое количество отклоненных заданий; соответственно $R(k)/k$ — ожидаемая часть отклоненных заданий. Чему равен предел $\lim_{k\to\infty} R(k)/k$? Приведите доказательство.

(c) Предположим, машины имеют встроенный буфер чуть большего размера; каждая машина M выполняет первые два полученных задания и отклоняет все остальные. Обозначим $R2(k)$ ожидаемое количество заданий, отклоненных по этому правилу. Чему равен предел $\lim_{k\to\infty} R_2(k)/k$? Приведите доказательство.

12. Рассмотрим аналогию с алгоритмом Каргера для нахождения минимального разреза *s–t*. Ребра будут стягиваться в итеративном режиме с использованием рандомизированной процедуры. Пусть в заданной итерации s и t — (возможно) стянутые узлы, содержащие исходные узлы s и t соответственно. Чтобы гарантировать, что узлы s и t не будут сжаты, на каждой итерации удаляются все ребра, соединяющие s и t, а стягиваемое ребро выбирается случайным образом среди остальных ребер. Приведите пример, показывающий, что вероятность нахождения этим методом минимального разреза *s–t* может быть экспоненциально малой.

13. Рассмотрим классический эксперимент с шарами и корзинами. Как обычно, каждый шар независимо попадает в одну из двух корзин, при этом попадание в каждую из корзин равновероятно. Ожидаемое количество шаров в каждой корзине равно n. В этой задаче исследуется вопрос возможной величины их разности. Пусть X_1 и X_2 — количество шаров в первой и второй корзине соответственно (X_1 и X_2 — случайные переменные). Докажите, что для любого $\varepsilon > 0$ существует константа $c > 0$, для которой вероятность $\Pr\left[X_1 - X_2 > c\sqrt{n}\right] \le \varepsilon$.

14. Ученые, разрабатывающие параллельные физические модели, обращаются к вам по поводу следующей задачи. Имеется множество P из k базовых процессов, требуется назначить каждый процесс на одну из двух машин M_1 или M_2. Затем они будут выполнять серию из n заданий J_1, ..., J_n. Каждое задание J_i представляется множеством $P_i \subseteq P$ из ровно $2n$ базовых процессов, которые должны выполняться (каждый на назначенной машине) во время обработки задания. Распределение базовых процессов по машинам называется *идеально сбалансированным*, если для каждого задания J_i ровно n базовых процессов, связанных с J_i, были назначены на каждую из двух машин. Распределение базовых процессов между машинами будет называться *почти сбалансированным*, если для каждого задания J_i не более $\frac{4}{3}n$ базовых процессов, связанных с J_i, были назначены на одну машину.

(a) Покажите, что для произвольно больших значений n существуют серии заданий $J_1, ..., J_n$, для которых не существует идеально сбалансированного распределения.

(b) Предположим, $n \geq 200$. Предложите алгоритм, который получает произвольную серию заданий $J_1, ..., J_n$, и строит почти сбалансированное распределение базовых процессов между машинами. Ваш алгоритм может быть рандомизированным; в этом случае ожидаемое время выполнения должно быть полиномиальным и алгоритм всегда должен выдавать правильный ответ.

15. Предположим, имеется очень большое множество S вещественных чисел, и вы хотите вычислить предполагаемую медиану этих чисел на основании выборки. Можно считать, что все числа в S различны. Пусть $n = |S|$; число x будет называться ε-*приближенной медианой* S, если по крайней мере $\left(\frac{1}{2} - \varepsilon\right)n$ чисел в S меньше x и по крайней мере $\left(\frac{1}{2} - \varepsilon\right)n$ чисел в S больше x.

Рассмотрим алгоритм, который выбирает случайное подмножество $S' \subseteq S$ с равномерным распределением, вычисляет медиану S' и возвращает ее как приближенную медиану S. Покажите, что существует абсолютная константа c, независимая от n, так что при применении алгоритма с выборкой S' размера c с вероятностью 0,99 возвращаемое число будет (0,05)-приближенной медианой S. (Рассмотрите либо версию алгоритма, строящую S' посредством формирования выборки с заменой, чтобы элемент S мог выбираться многократно, либо версию без замены.)

16. Рассмотрим следующий (частично определенный) метод безопасной передачи сообщения между отправителем и получателем. Сообщение будет представлено в виде битовой строки. Пусть $\sum = \{0,1\}$ и $\sum^*$ — множество всех строк из 0 и более битов (например, $0,00,1110001 \in \sum^*$). «Пустая строка», не содержащая битов, будет обозначаться $\lambda \in \sum^*$.

Отправитель и получатель совместно используют секретную функцию $f : \sum^* \times \sum \to \sum$. Иначе говоря, f получает слово и бит и возвращает бит. Получая последовательность битов $\alpha \in \sum^*$, получатель выполняет следующий метод для ее расшифровки.

```
Пусть α = α1α2...αn, где n – количество битов в α.
Целью является получение расширенного n-битного сообщения
β = β1β2...βn
Присвоить β1 = f(λ,α1)
Для i = 2,3,4, ..., n
   Присвоить βi = f(β1β2...βi-1,αi)
Конец цикла
Вывести β
```

Происходящее можно рассматривать как своего рода «потоковый шифр с обратной связью». Недостаток такого решения заключается в том, что если любой

бит α_i будет поврежден при передаче, он повредит вычисленное значение β_j для всех $j \geq i$.

Рассмотрим следующую задачу: отправитель S хочет передать сообщение (простой текст) β каждому из k получателей $R_1, \ldots, R_k$. С каждым получателем он обменивается секретной функцией $f^{(i)}$ (разной для разных получателей). Затем каждому получателю отправляется свое зашифрованное сообщение $\alpha^{(i)}$, такое что $\alpha^{(i)}$ расшифровывается в β при выполнении описанного выше алгоритма с функцией $f^{(i)}$.

К сожалению, каналы связи сильно зашумлены, и каждый из n битов в каждой из k передач подвергается независимому искажению (то есть принимает противоположное значение) с вероятностью 1/4. Таким образом, любой получатель с большой вероятностью не сможет расшифровать сообщение сам по себе. Покажите, что если значение k достаточно велико как функция n, то k получателей смогут совместными усилиями восстановить текстовое сообщение: для этого они собираются вместе и, не раскрывая никакие из $\alpha^{(i)}$ или $f^{(i)}$, в интерактивном режиме выполняют алгоритм, который строит правильное содержимое β с вероятностью не менее 9/10. (Насколько большим должно быть значение k в вашем алгоритме?)

17. Рассмотрим простую модель азартной игры при неблагоприятных шансах. В начале общий выигрыш равен 0. Проводится серия из n раундов; в каждом раунде выигрыш возрастает на 1 с вероятностью 1/3 и убывает на 1 с вероятностью 2/3.

    Покажите, что для ожидаемого количества шагов, при котором выигрыш положителен, можно определить верхнюю границу в виде абсолютной константы, не зависящей от значения n.

18. В этой задаче рассматривается рандомизированный алгоритм для задачи о вершинном покрытии.

```
Инициализировать S = Ø
Пока S не является вершинным покрытием,
   Выбрать ребро e, не покрываемое S
   Выбрать один конец e случайным образом (с равной вероятностью).
   Добавить выбранный узел в S.
Конец Пока
```

    Нас интересует ожидаемая стоимость вершинного покрытия, выбираемого этим алгоритмом.

    (a) Является ли этот алгоритм c-аппроксимирующим алгоритмом для задачи о вершинном покрытии с минимальным весом для некоторой константы c? Докажите свой ответ.

    (b) Является ли этот алгоритм c-аппроксимирующим алгоритмом для задачи о вершинном покрытии с минимальной мощностью для некоторой константы c? Докажите свой ответ.

Подсказка: обозначим p_e вероятность того, что ребро e выбирается как непокрытое ребро в этом алгоритме. Сможете ли вы выразить ожидаемое значение решения через эти вероятности? Чтобы ограничить значение оптимального решения в контексте вероятностей p_e, попробуйте ограничить сумму вероятностей для ребер, инцидентных заданной вершине v, а именно $\sum_{e \text{ incident to } v} p_e$.

Примечания и дополнительная литература

В области применения рандомизации в алгоритмах ведутся активные исследования; в частности, этой теме посвящены книги Мотвани и Рагхавана (Motwani, Raghavan, 1995), а также Митценмахера и Упфала (Mitzenmacher, Upfal, 2005). Как видно из материала этой главы, вероятностные аргументы, используемые при изучении базовых рандомизированных алгоритмов, часто имеют дискретный, комбинаторный оттенок; такой стиль вероятностного анализа рассматривается в книге Феллера (Feller, 1957).

Применение рандомизации для разрешения конфликтов типично для многих систем и сетевых приложений. Например, в коммуникационных средах в стиле Ethernet рандомизированные протоколы используются для сокращения количества коллизий между разными отправителями; эта тема рассматривается в книге Бертсекаса и Галлагера (Bertsekas, Gellager, 1992).

Рандомизированный алгоритм для задачи о минимальном разрезе, описанной в тексте, был разработан Каргером. После дополнительных оптимизаций Каргера и Штейна (Karger, Stein, 1996) он стал одним из самых эффективных методов решения задачи о минимальном разрезе. Дополнительные расширения и возможности применения алгоритма описаны в диссертации Каргера (Karger, 1995).

Аппроксимирующий алгоритм для задачи MAX 3-SAT был разработан Джонсоном (Johnson, 1974) для статьи, в которой были опубликованы ранние аппроксимирующие алгоритмы для *NP*-сложных задач. Неожиданный вывод из этого раздела — что каждый экземпляр 3-SAT имеет присваивание, выполняющее не менее 7/8 условий, — является примером *вероятностного метода*, тогда как существование комбинаторной структуры с нужным свойством аргументируется просто тем, что случайная структура обладает данным свойством с положительной вероятностью. В этой области комбинаторики были проведены серьезные исследования; многие практические применения данного метода описаны в книге Элона и Спенсера (Alon, Spencer, 2000).

Хеширование стало предметом самостоятельных масштабных исследований (как в теоретических, так и в практических условиях), и у базового метода существует много разновидностей. Метод, которому посвящен раздел 13.6, был предложен Картером и Вегманом (Carter, Wegman, 1979). Идея применения рандомизации для нахождения ближайшей пары точек на плоскости была предложена Рабином (Rabin, 1976) в авторитетной ранней статье, посвященной применению рандомизации во многих алгоритмических ситуациях. Алгоритм, описанный в этой главе, был разработан Голином и др. (Golin, 1995). Метод, применяемый для ограничения

количества операций со словарем, в котором суммируется ожидаемая работа по всем этапам в случайном порядке, иногда называется *обратным анализом*; изначально он был предложен Чу (Chew, 1985) для сопутствующей геометрической задачи. Другие применения обратного анализа описаны в работе Зейдела (Seidel, 1993).

Гарантии эффективности для алгоритма кэширования LRU принадлежат Слитору и Тарьяну (Steator, Torjan, 1985), а авторами границы рандомизированного алгоритма маркировки являются Фиат, Карп, Лаби, Макгиох, Слитор и Янг (Fiat, Karp, Luby, McGeoch, Sleator, Young, 1991). На более общем уровне в статье Слитора и Тарьяна выделяется понятие *оперативных алгоритмов*, которые должны обрабатывать входные данные без информации о будущем; кэширование — одна из фундаментальных областей, требующих применения таких алгоритмов. Теме оперативных алгоритмов посвящена книга Бородина и Эль-Янива (Borodin, El-Yaniv, 1998), в которой приведено много других результатов, относящихся к кэшированию.

Существует много других способов формулировки границ из раздела 13.9, демонстрирующих, что значительное отклонение суммы независимых случайных переменных, принимающих значения 0 и 1, от среднего значения маловероятно. Результаты такого рода обычно называются *границами Чернова* или *границами Чернова-Хеффдинга* на основании работ Чернова (Chernoff, 1952) и Хеффдинга (Hoeffding, 1963). В книгах Элона и Спенсера (Alon, Spencer, 1992), Мотвани и Рагхавана (Motwani, Raghavan, 1995), Митценмахера и Упфала (Mitzenmacher, Upfal, 2005) такие границы рассматриваются более подробно, с дополнительными примерами применения.

Результаты пакетной маршрутизации в контексте максимальной длины и максимальной загрузки были получены Лейтоном, Маггсом и Рао (Leighton, Maggs, Rao, 1994). Маршрутизация — еще одна область, в которой рандомизация эффективно работает на сокращение конфликтов и пиковых нагрузок; многие применения этого принципа рассматриваются в книге Лейтона (Leighton, 1992).

Примечания к упражнениям

Упражнение 6 основано на результатах Бенни Чора и Мадху Судана; упражнение 9 представляет собой версию «задачи о секретаре», популяризация которой часто приписывается Мартину Гарднеру.

Эпилог: алгоритмы, которые работают бесконечно

У каждого десятилетия свои увлекательные головоломки; и если в начале 1980-х годов бесспорное первое место среди математических развлечений занимал кубик Рубика, то «Тетрис» вызывает похожее чувство ностальгии по концу 80-х и началу 90-х годов. У кубика Рубика и «Тетриса» есть кое-что общее: математический оттенок, базирующийся на стилизованных геометрических формах, — но пожалуй, различия между ними представляют еще больший интерес.

Кубик Рубика — игра, сложность которой определяется огромным пространством поиска; к заданной перемешанной конфигурации кубика требуется применить серию операций, чтобы достичь конечной цели. С другой стороны, «Тетрис» (в его «чистой» форме) имеет куда более размытое определение успеха; вместо конкретной конечной точки вы сталкиваетесь с бесконечным потоком событий, на которые необходимо непрерывно реагировать.

Эти отличительные признаки «Тетриса» напоминают аналогичные темы, проявившиеся в недавних работах по анализу алгоритмов. Все чаще встречаются ситуации, в которых традиционные представления об алгоритмах — в начале выполнения алгоритм получает входные данные, выполняется за конечное количество шагов и выдает результат — уже неприменимы. Попробуйте представить себе маршрутизаторы в Интернете, которые передают пакеты и пытаются избежать перегрузок; или децентрализованные механизмы обмена файлами, которые копируют и распространяют контент по требованию пользователей; или средства машинного обучения, формирующие прогностические модели концепций, изменяющихся со временем; во всех этих случаях мы имеем дело с алгоритмами, практически предназначенными для *бесконечного выполнения*. Признаком успешного выполнения является не окончательный результат, а способность алгоритма сохранить работоспособность в постоянно изменяющейся среде, постоянно подкидывающей новые задачи. В таких ситуациях мы переходим из мира кубика Рубика в мир «Тетриса».

Существует много разных примеров, на которых можно исследовать эту тему. В последнем разделе книги мы рассмотрим одну из самых интересных ситуаций: разработку алгоритмов для высокоскоростной коммутации пакетов в Интернете.

Задача

Пакет, перемещающийся от источника к получателю в Интернете, можно представить как маршрут обхода пути в большом графе, узлам которого соответствуют сетевые коммутаторы, а ребрам — кабели, которые связывают коммутаторы. У каждого пакета p имеется заголовок, по которому коммутатор при поступлении пакета p по входному каналу может определить выходной канал, через который p следует отправить. Таким образом, задача сетевого коммутатора — взять поток пакетов, прибывающих по входным каналам, и как можно быстрее переместить каждый пакет в конкретный выходной канал, по которому он должен быть отправлен. Насколько быстро? В средах с высокой нагрузкой пакет может поступать на каждый входной канал через несколько десятков наносекунд; если пакеты не будут передаваться на соответствующие выходные каналы со сравнимой скоростью, то трафик начнет накапливаться, что приведет к потере пакетов.

Чтобы рассуждать об алгоритмах, работающих внутри коммутатора, мы определим модель коммутатора следующим образом: он имеет n входных каналов I_1, ..., I_n и n выходных каналов O_1, ..., O_n. Пакеты прибывают по входным каналам; с пакетом p связывается тип ввода/вывода ($I[p]$, $O[p]$), который указывает, что он поступил по входному каналу $I[p]$ и должен отправиться по выходному каналу $O[p]$. В системе ведется дискретный отсчет времени; за один шаг на каждом входном канале прибывает не более одного нового пакета и не более одного пакета может отправиться по каждому выходному каналу.

Рассмотрим пример на рис. Е.1. За один квант времени на пустой коммутатор прибывают три пакета p, q и r по входным каналам I_1, I_3 и I_4 соответственно. Пакет p предназначен для выходного канала O_1, пакет q — для канала O_3, и пакет r тоже предназначен для O_3. С отправкой пакета по каналу O_1 проблем нет; но по каналу O_3 может быть отправлен только один пакет, поэтому коммутатор должен разрешить конфликт между q и r. Как это сделать?

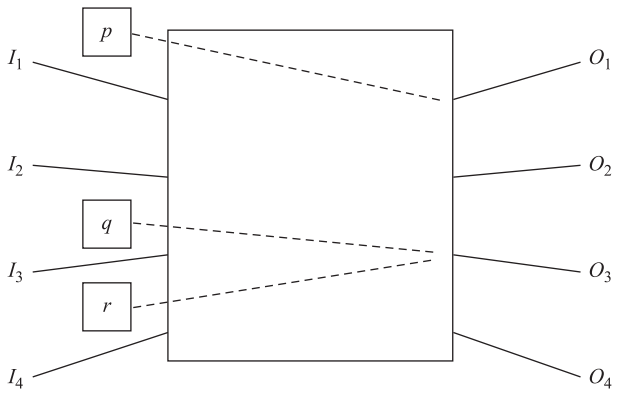


Рис. Е.1. Коммутатор с $n = 4$ входными и выходными каналами. За один временной шаг прибывают пакеты p, q и r

Простейшее поведение коммутатора в такой ситуации называется *чистой выходной очередью*; по сути, это идеализированная картина того, что бы мы хотели видеть от коммутатора. В этой модели все пакеты, прибывающие на некотором шаге, помещаются в выходной буфер, связанный с выходным каналом, и отправляется один из пакетов из каждого выходного буфера. Ниже приведена более конкретная модель одного временного шага.

```
Один шаг в дисциплине чистой выходной очереди:
   Пакеты прибывают по входным каналам
   Каждый пакет p типа (I[p], O[p]) перемещается в выходной буфер O[p]
   Из каждого выходного буфера отправляется не более одного пакета
```

Итак, на рис. Е.1 заданный временной шаг может закончиться тем, что пакеты p и q будут отправлены по своим выходным каналам, а пакет r останется в выходном буфере O_3. (В дальнейшем обсуждении этого примера предполагается, что при принятии решений q отдается предпочтение перед r.) В этой модели коммутатор представляется как идеальный объект, через который пакеты беспрепятственно проходят к своему выходному буферу.

Однако на практике пакет, поступивший по входному каналу, необходимо скопировать в соответствующий выходной канал, а эта операция требует некоторых вычислений, которые блокируют входной и выходной каналы на несколько наносекунд. Итак, в действительности ограничения коммутатора создают некоторые препятствия при перемещении пакетов из входных каналов в выходные.

Самая жесткая модель этих ограничений — *очереди ввода/вывода* — работает следующим образом: имеется входной буфер для каждого входного канала I, а также выходной буфер для каждого выходного канала O. При поступлении пакет немедленно помещается в соответствующий входной буфер. За один временной шаг коммутатор может прочитать не более одного пакета из каждого входного буфера и записать не более одного пакета в каждый выходной буфер. Итак, в модели очередей ввода/вывода пример на рис. Е.1 будет работать так: каждый из пакетов p, q и r помещается в отдельный входной буфер; коммутатор перемещает p и q в их выходные буферы, но не может переместить все три пакета, потому что для этого пришлось бы записать два пакета в выходной буфер O_3. Таким образом, первый шаг завершится отправкой пакетов p и q по их выходным каналам, а пакет r остается во входном буфере I_4 (вместо выходного буфера O_3).

На более общем уровне ограничение на количество операций чтения и записи выглядит так: если пакеты $p_1, ..., p_\ell$ перемещаются за один временной шаг из входных буферов в выходные, то все их входные и выходные буферы должны быть различны. Другими словами, типы $\{(I[p_i], O[p_i]): i = 1,2, ..., \ell\}$ должны образовать двудольное паросочетание. Тогда один временной шаг можно смоделировать следующим образом.

```
Один шаг в дисциплине очередей ввода/вывода:
   Пакеты прибывают по входным каналам и помещаются во входные буферы
   Пакеты, типы которых образуют паросочетание, перемещаются
      в соответствующие выходные буферы
   Из каждого выходного буфера отправляется не более одного пакета
```

Выбор перемещаемого паросочетания пока остается не определенным; важность этого момента станет ясна позднее.

Итак, при использовании очередей ввода/вывода коммутатор создает некоторое «трение» на пути перемещения пакетов, причем это объективно наблюдаемое явление: если рассматривать коммутатор как «черный ящик» и просто наблюдать за последовательностью отправок по выходным каналам, можно заметить различия между чистой выходной очередью и очередями ввода/вывода. Рассмотрим пример, в котором первый шаг в точности повторяет рис. E.1, а на втором шаге прибывает один пакет s типа (I_4, O_4). При использовании чистой выходной очереди пакеты p и q отправятся на первом шаге, а r и s — на втором. С другой стороны, при использовании очередей ввода/вывода происходит последовательность событий, изображенная на рис. E.2: в конце первого шага r по-прежнему находится во входном буфере I_4, а в конце второго шага один из пакетов r или s также остается во входном буфере I_4, ожидая отправки. Конфликт между r и s называется *блокировкой очереди*; из-за него коммутатор с очередями ввода/вывода по своим характеристикам задержки уступает чистой выходной очереди.

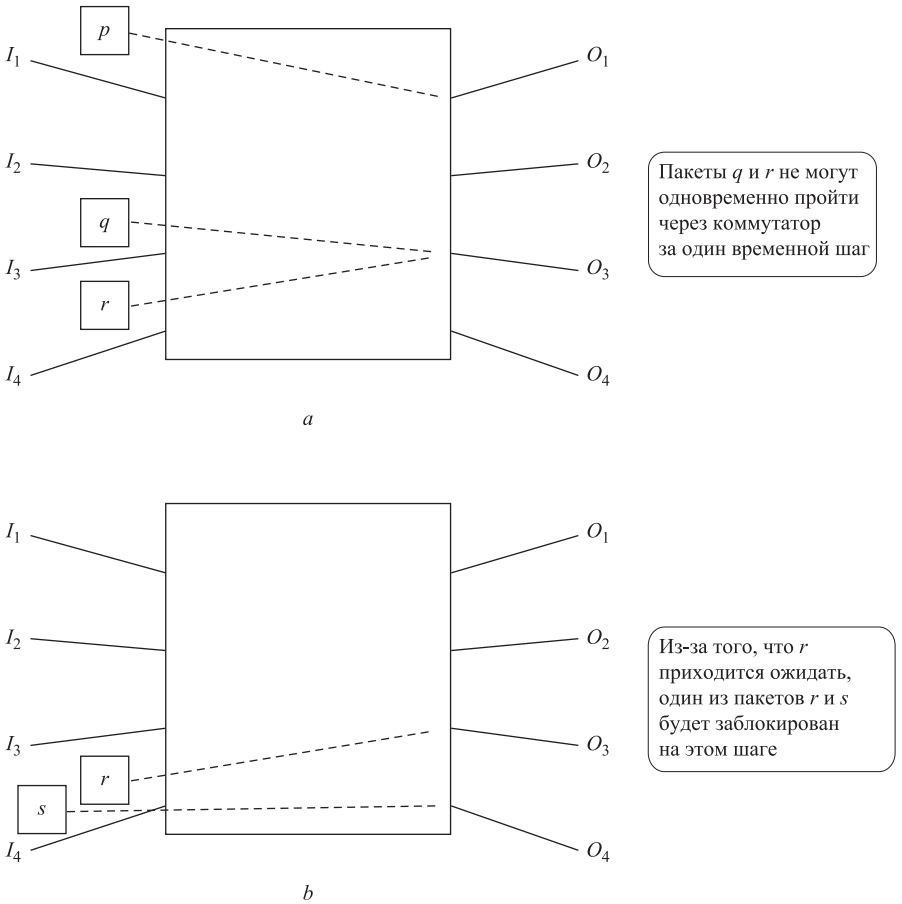


Рис. E.2. Двухшаговый процесс, приводящий к блокировке очереди

Моделирование чистой выходной очереди

Поведение чистой выходной очереди было бы удобным, но из сказанного выше понятно, что препятствует реализации коммутатора с таким поведением: за один временной шаг (продолжительностью всего десятки наносекунд) переместить пакеты из одного из n входных каналов в общий выходной буфер в общем случае не удастся.

Но что, если взять коммутатор с очередями ввода/вывода и ускорить его работу, перемещая несколько сочетаний за шаг вместо одного? Возможно ли смоделировать коммутатор, использующий чистую выходную очередь? Под этим термином здесь понимается, что последовательность отправок по выходным каналам (при которой коммутатор рассматривается как «черный ящик») должна быть одинаковой для реализации поведения чистой выходной очереди и ускоренного алгоритма очередей ввода/вывода.

Нетрудно понять, что ускорения в n раз будет достаточно: если мы сможем перемещать n сочетаний на каждом шаге, то даже если все прибывающие пакеты должны попасть в один выходной буфер, мы смогли бы переместить их все за один шаг. Но ускорение в n раз совершенно нереально; и если рассматривать происходящее как худший случай, появляется неприятная мысль о том, что без ускорения в n раз не обойтись, — в конце концов, что делать, если все прибывающие пакеты *действительно* должны быть направлены в один выходной буфер?

В этом разделе будет показано, что хватит гораздо более умеренного ускорения. Мы опишем поразительный результат, который получили Чуан, Гоэл, Маккеун и Прабхакар (Chuang, Goel, McKeown, Prabhakar, 1999), — он показывает, что коммутатор, использующий очереди ввода/вывода с ускорением 2, может моделировать коммутатор с чистой выходной очередью. На уровне здравого смысла этот результат использует тот факт, что внутреннее поведение коммутатора не обязано напоминать поведение чистой выходной очереди, если последовательность отправок пакетов с выходных каналов остается неизменной. (Следовательно, продолжая пример из предыдущего абзаца, не обязательно перемещать все n входных пакетов в общий выходной буфер за один шаг; для этого можно выделить больше времени, потому что их отправки по общему выходному каналу все равно будут распределены по более длинному периоду времени.)

Разработка алгоритма

Просто для ясности приведем модель для ускорения 2.

```
Один шаг с ускоренной реализацией очередей ввода/вывода:
   Пакеты поступают по входным каналам и помещаются во входные буферы
   Пакеты, типы которых образуют паросочетание, перемещаются
      в соответствующие выходные буферы
   Из каждого выходного буфера отправляется не более одного пакета
   Пакеты, типы которых образуют паросочетание, перемещаются
      в соответствующие выходные буферы
   Пакеты, типы которых образуют паросочетание, перемещаются
      в соответствующие выходные буферы
```

Чтобы доказать, что эта модель способна имитировать чистую выходную очередь, необходимо разрешить важный, недостаточно определенный момент: какие паросочетания должны перемещаться на каждом шаге? Ответ на этот вопрос образует основу результата, и мы придем к нему через серию промежуточных шагов. Начнем с одного простого наблюдения: если пакет типа (I, O) является частью паросочетания, выбранного коммутатором, то коммутатор переместит пакет этого типа с самым *ранним* временем отправки.

Организация входных и выходных буферов

Чтобы решить, какие два паросочетания коммутатор должен переместить на заданном шаге, необходимо определить некоторые характеристики, определяющие текущее состояние коммутатора относительно чистой выходной очереди. Для начала для пакета p определяется его время отправки $TL(p)$, равное временному шагу, на котором он был бы отправлен по выходному каналу коммутатора при использовании чистой выходной очереди. Мы стремимся к тому, чтобы каждый пакет p отправлялся из коммутатора (с ускоренными очередями ввода/вывода) ровно на шаге $TL(p)$.

На концептуальном уровне каждый входной буфер представляет собой упорядоченный список; при этом сохраняется возможность вставки прибывающих пакетов в середину порядка, а также перемещения пакета в выходной буфер даже тогда, когда он еще не находится в начале очереди. Несмотря на это, линейное упорядочение буфера образует полезную метрику прогресса. С другой стороны, выходные буферы упорядочивать не нужно; когда приходит время отправки пакета, он просто покидает буфер. Конфигурация напоминает терминал аэропорта: входные буферы соответствуют стойкам регистрации, выходные — залам ожидания, а внутренняя реализация коммутатора — сильно загруженному контрольному пункту службы безопасности. Основное напряжение приходится на входные буферы: если не успеть к началу очереди до завершения регистрации, то вы можете опоздать на самолет; к счастью, служащие аэропорта могут извлечь вас из середины очереди и провести через контрольный пункт в ускоренном темпе. Напротив, в выходных буферах можно расслабиться: пассажиры спокойно сидят, пока не будет объявлена посадка на их рейс, после чего просто отправляются на самолет. Главное — преодолеть «затор» в середине, чтобы пассажиры успели к отлету.

В частности, из этой аналогии следует, что нам не нужно беспокоиться о пакетах, уже находящихся в выходных буферах; они отправятся в нужное время. Соответственно мы будем называть пакет p необработанным, если он все еще находится во входном буфере, и определим некоторые полезные метрики для таких пакетов. *Входным окном* $IC(p)$ называется количество пакетов, находящихся перед пакетом p в его входном буфере. *Выходным окном* $OC(p)$ называется количество пакетов, уже находящихся в выходном буфере p и имеющих более раннее время отправки. У необработанного пакета p дела идут хорошо, если $OC(p)$ намного больше $IC(p)$; в этом случае p находится ближе к началу очереди в его входном буфере, а в выходном буфере перед ним еще остается много пакетов. Для представления

этой связи мы определим метрику $Slack(p) = OC(p) - IC(p)$, заметив, что большие значения $Slack(p)$ являются предпочтительными.

Итак, наш план: паросочетания будут передаваться по коммутатору так, чтобы постоянно сохранялись следующие два свойства:

(i) $Slack(p) \geq 0$ для всех необработанных пакетов p;

(ii) На любом шаге, начинающемся с $IC(p) = OC(p) = 0$, пакет p будет перемещаться в свой выходной буфер в первом паросочетании.

Сначала нужно убедиться в том, что сохранения этих двух свойств достаточно.

(E.1) Если свойства (i) и (ii) выполняются для всех необработанных пакетов во все моменты времени, то каждый пакет p будет отправлен в свое положенное время $TL(p)$.

Доказательство. Если p находится в своем выходном буфере в начале шага $TL(p)$, то, очевидно, он может быть отправлен; в противном случае он бы находился во входном буфере. В таком случае $OC(p) = 0$ в начале шага. Согласно свойству (i), имеем $Slack(p) = OC(p) - IC(p) \geq 0$, а следовательно, $IC(p) = 0$. Затем из свойства (ii) следует, что пакет p будет перемещен в выходной буфер в первом паросочетании этого шага, а следовательно, также будет отправлен в этом шаге. ■

Как выясняется, свойство (ii) гарантируется достаточно легко (и оно естественно выполняется в приведенном ниже решении), поэтому мы сосредоточимся на непростой задаче выбора паросочетаний для поддержания свойства (i).

Перемещение паросочетания через коммутатор

Когда пакет p только прибывает во входной канал, он вставляется как можно ближе к концу входного буфера (вероятно, где-то в середине) в соответствии с требованием $Slack(p) \geq 0$. Это гарантирует, что свойство (i) будет изначально выполняться для p.

Чтобы поддерживать неотрицательность *Slack* с течением времени, необходимо побеспокоиться о компенсации событий, приводящих к убыванию $Slack(p)$. Вернемся к описанию одного временного шага и подумаем, как происходит такое убывание.

```
Один шаг с ускоренной реализацией очередей ввода/вывода:
   Пакеты поступают по входным каналам и помещаются во входные буферы.
   Коммутатор перемещает паросочетание
   Из каждого выходного буфера отправляется не более одного пакета
   Коммутатор перемещает паросочетание
```

Рассмотрим пакет p, остающийся необработанным в начале временного шага. В фазе прибытия $IC(p)$ может увеличиться на 1, если прибывающий пакет помещается во входной буфер перед p. Это приведет к уменьшению $Slack(p)$ на 1. В фазе отправки этого шага $OC(p)$ может уменьшиться на 1, так как пакет с более ранним временем отправки уже не будет находиться в выходном буфере. Это тоже приведет к уменьшению $Slack(p)$ на 1. Итак, $Slack(p)$ теоретически может уменьшиться на 1 в каждой из фаз прибытия и отправки. Соответственно выполнение свойства

(i) будет обеспечено, если мы сможем гарантировать, что *Slack*(*p*) увеличивается по меньшей мере на 1 при каждом перемещении паросочетания через коммутатор. Как этого добиться?

Если перемещаемое паросочетание включает пакет из *I*[*p*], находящийся перед *p*, то *IC*(*p*) уменьшается, а следовательно, *Slack*(*p*) увеличится. Если паросочетание включает пакет, предназначенный для *O*[*p*], но имеющий более раннее время отправки, чем *p*, то *OC*(*p*) и *Slack*(*p*) возрастут. Итак, проблема возникает только в том случае, если не происходит ни одно из этих событий. На рис. E.3 приведено схематичное изображение такой ситуации. Предположим, пакет *x* перемещается из *I*[*p*] даже в том случае, если он находится ближе к концу очереди, а пакет *y* перемещается в *O*[*p*], хотя он имеет более позднее время отправки. В такой ситуации буферы *I*[*p*] и *O*[*p*] работают «некорректно»: для *I*[*p*] было бы лучше отправить пакет, находящийся ближе к началу очереди (такой, как *p*), а для *O*[*p*] было бы лучше получить пакет с более ранним временем отправки (такой, как *p*). В сочетании два буфера образуют некий аналог неустойчивости из задачи об устойчивом паросочетании.

Мы можем этот принцип выразить точно — и он станет ключом для завершения алгоритма. Допустим, для выходного буфера *O* входной буфер *I* является *предпочтительным* перед *I′*, если самое раннее время отправки среди пакетов типа (*I*, *O*) меньше самого раннего времени отправки между пакетами типа (*I′*, *O*). (Иначе говоря, если у буфера *I* существует более срочная потребность отправить что-то в буфер *O*.) Кроме того, для входного буфера *I* выходной буфер *O* является предпочтительным перед выходным буфером *O′*, если ближайший к началу пакет типа (*I*, *O*) опережает ближайший к началу пакет типа (*I*, *O′*) в упорядочении *I*. Для каждого буфера по этим правилам строится список предпочтений; и если пакетов типа (*I*, *O*) вообще нет, то *I* и *O* помещаются в конец списка предпочтений друг друга, с произвольной разбивкой связей. Наконец, алгоритм определяет устойчивое паросочетание *M* с учетом этих списков предпочтений, и коммутатор перемещает это паросочетание *M*.

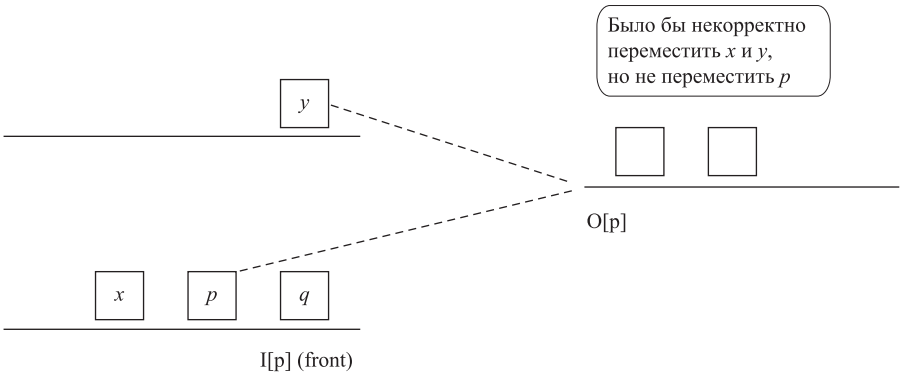


Рис. E.3. Выбор перемещаемого паросочетания

Анализ алгоритма

Следующее утверждение показывает, что выбор устойчивого паросочетания действительно дает алгоритм с нужными гарантиями быстродействия.

(E.2) Предположим, коммутатор всегда перемещает устойчивое паросочетание M в отношении списков предпочтений, определенных выше. (И для каждого типа (I, O), содержащегося в M, выбирается пакет этого типа с самым ранним временем отправки.) Тогда для всех необработанных пакетов p при перемещении паросочетания M значение $\mathtt{Slack}(p)$ увеличивается минимум на 1.

Доказательство. Рассмотрим любой необработанный пакет p. Продолжая предшествующее обсуждение, предположим, что в составе паросочетания M не перемещается никакой пакет, опережающий p в $I[p]$, и никакой пакет, предназначенный для $O[p]$, с более ранним временем отправки. Итак, пара $(I[p], O[p])$ не принадлежит M; предположим, что пары $(I', O[p])$ и $(I[p], O')$ принадлежат M.

Пакет p имеет более раннее время отправки, чем любой пакет типа $(I', O[p])$, и опережает все пакеты типа $(I[p], O')$ в упорядочении $I[p]$. Отсюда следует, что для $I[p]$ буфер $O[p]$ является предпочтительным перед O', и для $O[p]$ буфер $I[p]$ является предпочтительным перед I'. Следовательно, пара $(I[p], O[p])$ образует неустойчивость, что противоречит предположению об устойчивости M. ■

Следовательно, перемещая устойчивое паросочетание на каждом шаге, коммутатор поддерживает свойство $\mathtt{Slack}(p) \geq 0$ для всех пакетов p; следовательно, в соответствии с (E.1) мы доказали следующее:

(E.3) Перемещая два устойчивых паросочетания на каждом временном шаге, в соответствии с только что определенными предпочтениями, коммутатор может имитировать поведение чистой выходной очереди.

Получается, алгоритм совершенно неожиданно встречается в теме, с которой начиналась книга, — но вместо пар из мужчин и женщин или работодателей и работников здесь формируются пары входных и выходных каналов в высокопроизводительном интернет-маршрутизаторе.

Мы всего лишь мельком затронули тему алгоритмов, работающих бесконечно в условиях бесконечного потока новых событий. В этой интересной области полно нерешенных проблем и открытых направлений. Но этой темой мы займемся как-нибудь в другой раз и в другой книге, а эта книга подошла к концу.

Об авторах

Джон Клейнберг (Jon Kleinberg) — профессор теории вычислительных систем в Корнелльском университете. Получил докторскую степень в Массачусетском технологическом институте в 1996 году.

Исследования Клейнберга посвящены алгоритмам, прежде всего связанным со структурой сетей и информации, и их практическим применениям в области компьютерных технологий, оптимизации, анализа данных и вычислительной биологии. Его работы по теме сетевого анализа помогли заложить основу современного поколения поисковых систем Интернета.

Ева Тардос (Eva Tardos) — профессор теории вычислительных систем в Корнелльском университете. Получила докторскую степень в Университе Этвёша в Будапеште (Венгрия) в 1984 году. Член Американской академии наук и искусств и Ассоциации по вычислительной технике США.

Научные интересы Тардос связаны с проектированием и анализом алгоритмов для решения задач из теории графов и сетей. Наибольшую известность получили ее работы в области алгоритмов сетевого потока и аппроксимирующих алгоритмов для решения сетевых задач. Ее недавние работы были посвящены алгоритмической теории игр.