Фундаментальные алгоритмы на C++

ЧАСТЬ 5

АЛГОРИТМЫ НА ГРАФАХ

Роберт Седжвик

торгово-издательский дом
DiaSoft

Москва • Санкт-Петербург • Киев

2002

Algorithms
THIRD EDITION
in C++

PARTS 5

GRAPH ALGORITHMS

Robert Sedgewick

Princeton University

ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

ББК 32.973.2
УДК 681.3.06(075)
С 83
Седжвик Роберт

С 83 Фундаментальные алгоритмы на C++. Алгоритмы на графах: Пер. с англ./Роберт Седжвик. – СПб: ООО «ДиаСофтЮП», 2002.– 496 с.

ISBN 5-93772-054-7

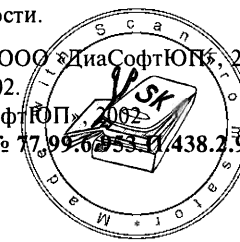
Эта книга посвящена глубокому исследованию всех основополагающих концепций и алгоритмов, которые, несомненно, относятся к категории «вечных». Тщательным образом проштудировав их, вы получите знания, которые *никогда* не устареют и которыми вы будете пользоваться *всегда*.

Краткость, точность, выверенность, актуальность, изобилие примеров и учебных заданий – вот лишь небольшой перечень очевидных достоинств книги. Иллюстрация алгоритмов на одном из наиболее эффективных языков программирования C++ лишний раз подчеркивает их популярность и «вечность». Подробно рассматривается широчайший спектр фундаментальных алгоритмов на графах, в числе которых: поиск в орграфах, неорграфах и сетях; построение минимальных остовных деревьев и кратчайших путей; вычисление потоков в сетях с различными характеристиками. Большое внимание уделяется рабочим характеристикам алгоритмов, а также их математическому выводу.

Книгу можно использовать в качестве курса лекций (как студентами, так и преподавателями), справочного пособия или просто «романа», получая при этом ни с чем не сравнимое удовольствие.

ББК 32.973.2

ISBN 5-93772-054-7 (рус.) © Перевод на русский язык. ООО «ДиаСофтЮП», 2002
ISBN 0-201-36118-3 (англ.) © Pearson Education, Inc., 2002.
© Оформление. ООО «ДиаСофтЮП», 2002
Гигиеническое заключение № 77.99.6.953.П.438.2.99 от 04.02.1999



Оглавление

Предисловие

Графы и алгоритмы на графах активно проникают во все современные компьютерные приложения. В этой книге описываются широко известные методы решения задач обработки графов, которые возникают на практике. Ее основная цель заключается в том, чтобы сделать эти методы и базовые принципы, составляющие их основу, доступными для все большего числа людей, которые в них нуждаются. Предлагаемый материал книги представлен таких образом, что сначала излагаются начальные сведения, начиная с базовой информации и основных понятий, с постепенным переходом к анализу классических методов, и завершается изучением современных технологий, которые все еще находятся на стадии разработки. Тщательно подобранные примеры, подробные рисунки и завершенные программные реализации сопровождаются подробным описанием алгоритмов и приложений.

Алгоритмы

Эта книга является второй из трех томов, задуманных в качестве обзора наиболее важных компьютерных алгоритмов, получивших в настоящее широкое распространение на практике. Первый том (части 1–4) охватывает фундаментальные понятия (часть 1), структуры данных (часть 2), алгоритмы сортировки (часть 3) и алгоритмы поиска (часть 4). В настоящем томе (часть 5) изложен материал, касающийся графов и алгоритмов на графах. Третий том (который находится в печати, части 6–8) охватывает строки (часть 6), вычислительную геометрию (часть 7) и современные алгоритмы и приложения.

Эти книги исключительно полезны на ранних стадиях курса обучения компьютерным наукам, сразу же после того, как студенты получат базовые навыки программирования и ознакомятся с компьютерными системами, и в то же время перед тем, как приступят к изучению специальных курсов по современным областям компьютерных наук или прикладных вычислительных систем. Эти книги принесут несомненную пользу в качестве материала для самообразования, а также справочного пособия для специалистов, занятых разработкой компьютерных систем и приложений, поскольку они содержат программные реализации полезных алгоритмов и подробные данные о рабочих характеристиках этих алгоритмов. Широкая перспектива, открывающаяся перед ними, делает эту серию книг подходящим введением в указанную выше область знаний.

Вместе эти три тома образуют *третье издание* книги, которая за многие годы получила широкое распространение в среде студентов и программистов по всему миру. Автор полностью переделал текст книги для этого издания и добавил несколько тысяч упражнений, сотни новых иллюстраций, десятки новых программ, а также снабдил все рисунки и программы развернутыми комментариями. Этот новый материал содержит как описание новых тем, так и более полный анализ многих классических алгоритмов. На протяжении всей книги основное внимание уделяется абстрактным типам данных, которые существенно расширяют область применения программ и делают их использование в средах объектно-ориентированного программирования более эффективным. Те, кто знаком с предыдущими изданиями настоящей книги, найдут в ней много новой информации; все читатели почерпнут из нее массу полезного педагогического материала, который обеспечивает четкое понимание основных концепций.

Эти книги предназначены не только для программистов и студентов, изучающих компьютерные науки. Все, кто работает с компьютером, хотят работать быстрее и решать все более крупные задачи. Алгоритмы, которые мы изучаем, представляют собой область знаний, быстро развивавшуюся в течение последних пятидесяти лет и ставшую основой для эффективного использования компьютеров на широком множестве приложений. Начиная с задач моделирования систем из N тел в физике и завершая задачами анализа в физике и задачами анализа генетического кода в молекулярной биологии, описанные здесь базовые методы стали основной частью современных научных исследований; от систем баз данных до поисковых механизмов в Internet, они стали важной частью современных программных систем. По мере того как сфера применения компьютерных приложений становится все шире, возрастает значение многих из базовых алгоритмов, особенно фундаментальных алгоритмов на графах, описание которых дано в этом томе. Назначение этой книги состоит в том, чтобы стать источником знаний для студентов и профессионалов, чтобы они понимали и при необходимости искусно использовали алгоритмы на графах в любом компьютерном приложении, каким бы оно ни было.

Круг рассматриваемых вопросов

Данная книга содержит шесть глав, в которых рассматриваются свойства и типы графов, поиск на графах, ориентированные графы, минимальные остовные деревья, кратчайшие пути и сети. Изложенные в ней описания предназначено для того, чтобы читатель получил по возможности более точное представление об основных свойствах широкого спектра фундаментальных алгоритмов на графах.

Собранный в книге материал вы сможете оценить по достоинству, имея за плечами курсы по изучению базовых принципов разработки и анализа алгоритмов и опыт программирования на языках высокого уровня, таких как C++, Java или C. Эта книга предполагает наличие у читателя соответствующей подготовки. Данный том предполагает знание механизма массивов, связных списков, структур, абстрактных типов данных (АТД), в нем используются очереди с приоритетами, таблицы символов, АТД объединения-поиска – все эти понятия подробно рассматриваются в частях 1–4 (и во многих других комментариях к алгоритмам и структурам данных).

Базовые свойства графов и алгоритмов на графах разработаны на основополагающих принципах, в то же время для их полного понимания очень часто необходимо глубоко погружаться в пучину сложных математических выкладок. Несмотря на то что обсуждение современных математических понятий носит конспективный характер, на уровне общих рассуждений и описаний от читателя, тем не менее, требуется более высокая математическая подготовка, нежели для работы с материалами, содержащимися в частях 1–4. Несмотря на это, читатели, обладающие различными уровнями математической подготовки, извлекут для себя немалую пользу из этой книги. К применению такого подхода вынуждает следующее обстоятельство: некоторые элементарные алгоритмы на графах, которые могут быть понятны и использованы каждым, лишь немногим отличаются от усовершенствованных алгоритмов, которые понимает далеко не каждый. Основная цель в подобных случаях – поместить важные алгоритмы в контекст других методов, а не требовать изучения всего математического аппарата. Однако строгий подход, на котором настаивают высококвалифицированные математики, часто приводят нас к созданию хо-

роших программ, в связи с чем автор стремился сохранить баланс между формальным подходом, на котором настаивают теоретики, и изложением материала, рекомендуемом практиками, не жертвуя при этом строгостью.

Использование материала в рамках учебных курсов

Что касается стиля изложения материала, то в этом плане преподавателю предоставляется широкая свобода действий, в зависимости от предпочтений преподавателя и подготовки студентов. Описанные в книге алгоритмы широко использовались в течение многих лет, они представляют собой совокупность знаний, необходимых как программисту-практику, так и студенту, изучающему теорию вычислительных систем. В данной книге содержится объем основного материала, достаточный для того, чтобы ее можно было использовать в качестве учебника по курсу алгоритмов и структур данных, в то же время она содержит достаточно материала, чтобы быть использованной в качестве учебника по курсу алгоритмов на графах. Возможно, одни преподаватели будут уделять основное внимание реализациям и практическим вопросам, а другие — анализу и теоретическим исследованиям.

Данная книга ориентирована на изучение алгоритмов, которые, скорее всего, будут использованы на практике. В ней содержится достаточно подробная информация об инструментальных средствах, позволяющих читателям уверенно реализовывать, отлаживать и запускать в работу алгоритмы решения различных задач или снабжать приложения необходимыми функциональными возможностями. В книгу включены полные реализации рассматриваемых в ней методов, равно как и описание работы этих программ на специально подобранном множестве примеров. Поскольку мы работаем с реальными программными кодами, а не пользуемся псевдокодами, эти программы можно быстро запустить в работу в рамках реальных приложений.

Действительно, одним из практических применений этих алгоритмов было создание сотен иллюстраций для данной книги. Благодаря этим иллюстрациям, суть многих алгоритмов становится понятной уже на интуитивном уровне.

В книге подробно рассматриваются рабочие характеристики алгоритмов и ситуации, в которых эти алгоритмы могут оказаться полезными. В контексте прослеживается связь с анализом алгоритмов и теорией вычислительных систем. Чтобы показать, почему предпочтение отдается тому или иному алгоритму, там, где это уместно, приводятся результаты эмпирических и аналитических исследований. В представляющих интерес случаях дается описание взаимосвязи между рассматриваемыми практическими алгоритмами и чисто теоретическими результатами. Специальная информация по рабочим характеристикам алгоритмов и их реализациям обобщается, выделяется и обсуждается на протяжении всей книги.

Язык программирования

Для всех реализаций используется язык программирования C++. В программах применяется широкое множество стандартных идиом C++, а в текст включены краткие описания каждой из таких конструкций.

Автор совместно с Крисом Ван Виком (Chris Van Wyk) разработали стиль программирования на C++, в основу которого положены классы, шаблоны и перегруженные операции, который, по нашему мнению, позволяет эффективно представлять алгоритмы и структуры данных в виде реальных программ. Мы стремимся к изящным, компактным, эффективным и переносимым реализациям. Везде, где это возможно, мы стремились сохранить этот стиль, чтобы сходные по действию программы выглядели похожими.

Цель настоящей книги заключается также и в том, чтобы представить алгоритмы в максимально простой и лаконичной форме. Для многих алгоритмов, приведенных в книге, схожесть сохраняется независимо от выбора языка: алгоритм Дейкстры (это лишь один из множества ярких примеров) остается алгоритмом Дейкстры независимо от того, представлен ли он на языке Algol-60, Basic, Fortran, Smalltalk, Ada, Pascal, C, C++, Modula-3, PostScript, Java или на одном из других бесчисленных языков или сред программирования, в которых он зарекомендовал себя как эффективный метод обработки графов. С одной стороны, мы разрабатываем тот или иной программный код с учетом опыта разработки алгоритмов на этих и на целом ряде других языков (C-версия этой книги уже доступна, а Java-версия пока готовится к печати). С другой стороны, некоторые особенности из перечисленных выше языков учитывают опыт их разработчиков, накопленный при работе с рядом других алгоритмов и структур данных, которые рассматриваются в этой книге. В конечном итоге мы приходим к заключению, что программный код, представленный в книге, служит точным описанием алгоритмов и весьма полезен на практике.

Благодарности

Многие читатели прислали мне исключительно полезные отзывы о предыдущих изданиях этой книги. В частности, в течение ряда лет предварительные наброски книги апробировались на сотнях студентов в Принстоне и Брауне. Особую благодарность хотелось бы выразить Трине Авери и Тому Фримену за оказанную помощь в выпуске первого издания; Джанет Инсерпи за проявленные ею творческий подход и изобретательность, чтобы заставить аппаратные и программные средства нашей примитивной и давно устаревшей компьютеризированной издательской системы напечатать первое издание книги; Марку Брауну за его участие в исследованиях по визуализации алгоритмов, которые во многом способствовали появлению в книге многочисленных рисунков, а также Дэйву Хенсону и Эндрю Эппелю за их готовность ответить на мои вопросы, связанные с языками программирования. Я хотел бы также поблагодарить многочисленных читателей, приславших отзывы на различные издания этой книги, в том числе Гая Олмсу, Джона Бентли, Марка Брауна, Джея Гришера, Аллана Хейдона, Кеннеди Лемке, Юди Манбер, Дану Ричардс, Джона Рейфа, М. Роенфельда, Стивена Сейдмана, Майка Квина и Вильяма Варда.

При подготовке нового издания я имел удовольствие работать с Питером Гордоном, Дебби Лафферти и Хелен Гольштейн из издательства Addison-Wesley, которые терпеливо опекали этот проект с момента его зарождения. Большое удовольствие доставила мне совместная работа с другими штатными сотрудниками этого издательства. Характер проекта сделал подготовку издания данной книги несколько непривычной задачей для многих из них, и я высоко ценю проявленную ими снисходительность. В частности, Мэрилин Рашш затратила немало усилий, чтобы втиснуть работу по изданию книги в жесткие временные рамки.

В процессе написания этой книги я приобрел трех новых наставников и хочу особо выразить им свою признательность. Во-первых, Стиву Саммиту, который внимательно проверил на техническом уровне первые варианты рукописи и предоставил буквально тысячи подробных комментариев, особенно в отношении программ. Стив хорошо понимал мое стремление снабдить книгу изящными и эффективные реализациями, и его комментарии помогли мне не только обеспечить определенное единообразие реализаций, но и существенно улучшить многие из них. Во-вторых, я хочу поблагодарить Лин Дюпре за тысячи подробных комментариев в отношении рукописи, которые помогли автору не только избежать и исправить грамматические ошибки, но и (что значительно важнее) выработать последовательный и связный стиль написания, что позволило собрать воедино устрашающую массу технического материала. Я исключительно благодарен полученной возможности поучиться у Стива и Лин — их вклад в разработку этой книги оказался решающим. В третьих, Крис Ван Вик реализовал и отладил все разработанные мною алгоритмы на C++, ответил на многочисленные вопросы, касающиеся C++, помог разработать соответствующий стиль на C++ и дважды внимательно прочел рукопись. Крис терпеливо сносил все мои придирки к разработанным им программам на C++, не возмущался, когда я отвергал многие из них, но по мере того, как мои знания языка C++ росли, мне приходилось возвращать эти программы практически в том виде, в каком они были написаны Крисом. Я бесконечно благодарен возможности многому научиться у Стива, Лин и Криса — они внесли решающий вклад в издание этой книги.

Многое из написанного здесь я узнал из лекций и трудов Дона Кнута - моего наставника в Стэнфорде. Хотя непосредственно Дон и не участвовал в написании этой книги, его влияние можно почувствовать на всем ее протяжении, ибо именно он поставил изучение алгоритмов на научную основу, благодаря чему вообще стало возможным появление подобного рода книг. Мой друг и коллега Филлип Флажоле, благодаря которому анализ алгоритмов стал вполне сформировавшейся областью исследований, оказал не меньшее влияние на этот труд.

Я глубоко признателен за оказанную мне поддержку Принстонскому университету, Брауновскому университету и Национальному институту исследований в области информатики и автоматики, где была проделана большая часть работы над книгой, а также Институту исследований защиты и Исследовательскому центру компании Xerox в Пало-Альто, где была завершена немалая часть работы. В основу многих глав этой книги положены исследования, которые щедро финансировались Национальным научным фондом и Отделом военно-морских исследований. И в заключение, я благодарю Билла Боуэна, Аарона Лемоника и Нейла Руденштайна за то, что они способствовали созданию в Принстоне академической обстановки, в которой я получил возможность подготовить эту книгу, несмотря на множество других возложенных на меня обязанностей.

Роберт Седжвик
Марли-де-Руа, Франция, февраль 1983 г.
Принстон, Нью-Джерси, январь 1990 г.
Джеймстаун, Род-Айленд, август 1997 г.

Предисловие консультанта по C++

Боб Сэджвик и я написали множество вариантов большей части программ, стремясь реализовать алгоритмы на графах в виде прозрачных и естественных программ. Поскольку существует великое разнообразие графов, порождающих множество касающихся их вопросов, мы согласились, что рано или поздно, но нам придется искать такую схему класса, которая будет работать на протяжении всей книги. Как ни странно, но эти поиски закончились использованием только двух схем: простой схемы, применяемой в главах 17–19, в которых ребра графа либо присутствуют, либо нет, и подхода, подобного контейнерам библиотеки STL (Standard Template Library — стандартная библиотека шаблонов) в главах 20–22, в которых с ребрами связана более обширная информация.

Классы C++ обладают существенными преимуществами для представления алгоритмов на графах. Мы используем классы для накопления полезных родовых функций на графах (наподобие ввода/вывода). В главе 18 мы используем классы для выделения операций, общих для нескольких различных методов поиска на графах. На протяжении всей книги мы применяем класс итератора к ребрам, исходящим из некоторой вершины, так что программы работают независимо от того, как хранится граф. Очень важно то, что мы упаковываем алгоритмы на графах в классы, конструктор которого осуществляют обработку графа и функции-элементы которого предоставляют нам доступ к информации об обнаруженных свойствах. Такая организация позволяет алгоритмам на графах без труда применять другие алгоритмы на графах в виде подпрограмм, например, программу 19.13 (транзитивное замыкание на основе сильных компонент), программу 20.8 (алгоритм Краскала построения минимального остовного дерева), программу 21.4 (построение всех кратчайших путей с использованием алгоритма Дейкстры), программу 21.6 (самый длинный путь в ориентированном ациклическом графе). Эта тенденция достигает наивысшей точки в главе 22, большая часть программ которой построена на более высоком уровне абстракции с использованием классов, которые были определены в предыдущих частях данной книги.

Дабы не входить в противоречие с материалом, изложенным в настоящей книге, наши программы используют разработанные в ней классы стеков и очередей, и мы применяем операции с явными указателями на односвязных списках в двухуровневых реализациях. Мы также принимаем два стилистических изменения из частей 1–4: конструкторы используют инициализацию вместо присваивания, и мы используем векторы STL вместо массивов. Ниже мы приводим перечень всех функций на векторах STL, которые были задействованы в наших программах:

- Конструктор по умолчанию создает пустой вектор.
- Конструктор **vec(n)** строит вектор из n элементов.
- Конструктор **vec(n, x)** строит вектор из n элементов, каждый из которых инициализируется значением **x**.
- Функция-элемент **vec.assign(n, x)** преобразует vec в вектор из n элементов, каждый из которых инициализируется значением **x**.
- Функция-элемент **vec.resize(n)** возрастает или убывает, получая пропускную способность **n**.

- Функция-элемент **vec.resize(n, x)** увеличивает или уменьшает **vec**, получающий пропускную способность **n**, и инициализирует любые новые элементы значением **x**.

STL также определяет оператор присваивания, конструктор копирования и деструктор, которые необходимы, чтобы сделать векторы объектами первого рода.

Прежде чем я приступил к работе над этими программами, я ознакомился с формальными описаниями и псевдокодами множества этих алгоритмов, однако реализовал только некоторые из них. Разработка деталей, необходимая для преобразования алгоритмов в работающие программы, оказалась для меня весьма поучительной, и было очень интересно наблюдать за тем, как они работают. Я надеюсь, что ознакомление с программами, опубликованными в этой книге, и их выполнение поможет вам достичь большего понимания собственно алгоритмов.

Мои искренние благодарности я направляю Джону Бентли, Брайану Кернингану и Тому Шиманski, от которых я узнал многое из того, что я сейчас знаю о программировании; Дебби Лафферти, которая предложила мне принять участие этом проекте. Я также приношу свою благодарность университету города Дрю и Принстонскому университету за безвозмездную поддержку.

Кристофер Ван Вик
Чатем, Нью Джерси, 2001 г.

Адаму, Эндрю, Бретт, Робби и, в первую очередь, Линде посвящается.

Примечания к упражнениям

Классификация упражнений — это занятие, сопряженное с рядом трудностей, поскольку читатели такой книги, как эта, обладают различными уровнями знаний и опыта. Тем не менее, определенное указание не помешает, поэтому многие упражнения помечены одним из четырех маркеров, дабы проще было выбрать соответствующий подход.

Упражнения, которые *проверяют, насколько хорошо вы усвоили материал*, помечены незаполненным треугольником, например:

▷ **18.34**. Рассмотрим граф

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

Вычертите дерево стандартного DFS на списках смежных вершин. Воспользуйтесь им для поиска мостов и реберно-связных компонент.

Чаще всего такие упражнения непосредственно связаны с примерами в тексте. Они не должны вызывать особых трудностей, в то же время их выполнение может прояснить факт или понятие, которые, возможно, ускользнули из внимания при чтении текста.

Упражнения, которые дополняют текст новой и требующей размышления информацией, помечены незаполненной окружностью, в частности:

19.106. Напишите программу, которая обобщает все возможные топологические упорядочения заданного графа DAG либо, если число таких упорядочений превышает границу, заданную в качестве аргумента, печатает это число.

Такие упражнения заставляют сосредоточиться на важных понятиях, связанных с материалом, изложенным в тексте, либо искать ответа на вопрос, который может возникнуть во время чтения. Возможно, читатели сочтут полезным прочесть эти упражнения даже при отсутствии времени на их выполнение.

Упражнения, которые имеют целью *бросить вызов* читателю, провоцируя его на решение трудной задачи, помечены черной точкой:

• **20.73**. Опишите, как вы будете искать дерево MST графа настолько большого, что в основной памяти одновременно могут находиться всего лишь V ребер.

Для выполнения таких упражнений требуется потратить значительное время, в зависимости от опыта читателя. В общем случае, лучше всего выполнять их в несколько приемов.

Несколько упражнений, которые *особенно трудны* (по сравнению с большинством других), помечены двумя черными точками, например:

•• **20.40**. Разработайте походящий генератор случайных графов с V вершинами и E ребрами, такой, что время выполнения алгоритма Прима, использующего поиск по приоритетам (программа 20.7), будет нелинейным.

Эти упражнения аналогичны вопросам, которые могут ставиться в научной литературе, однако материал книги может так подготовить читателей, что им доставит удовольствие попытаться ответить на них (а возможно, и преуспеть в этом).

Мы старались, чтобы все пометки были безотносительны к программной и математической подготовке читателей. Те упражнения, которые требуют наличия опыта по программированию или математическому анализу, очевидны. Мы настоятельно рекомендуем всем читателям проверить свое понимание алгоритмов, реализовав их. Тем не менее, упражнения подобные приведенному ниже, не представляют каких-либо трудностей для профессиональных программистов или студентов, изучающих программирование, но могут потребовать существенных усилий от тех, кто в последнее время по ряду причин программированием не занимался:

- **17.74**. Напишите программу, которая генерирует V случайных точек на плоскости, после чего строит граф, состоящий из ребер, соединяющих все пары точек, удаленных друг от друга на расстояние, не превышающее d (см. рис. 17.13 и программу 3.20). Определите, какое значение d следует выбрать, чтобы ожидаемое число ребер было равно E.

В том же духе мы рекомендуем читателям стремиться учитывать приводимые нами аналитические обоснования свойств всех алгоритмов. С другой стороны, упражнения, подобные приведенному ниже, не составят сложности для профессионального математика или студента, изучающего дискретную математику, однако наверняка потребуют значительных усилий от тех, кто давно не занимался математическим анализом:

- **19.5**. Сколько орграфов соответствует каждому неориентированному графу, содержащему V вершин и E ребер?

Книга снабжена настолько большим количеством упражнений, что их все невозможно прочесть и усвоить. Тем не менее, я надеюсь, что среди них есть достаточно таких, которые могут послужить для читателей большим стимулом к углубленному исследованию соответствующих тем.

Часть 5

Алгоритмы на графах

В этой части:

Глава 17

Свойства и типы графов

Многие вычислительные приложения естественным образом используют не только набор *элементов* (*items*), но также и *набор соединений* (*connections*) между парами таких элементов. Отношения, которые вытекают из этих соединений, немедленно вызывают множество естественных вопросов. Существует ли путь от одного такого элемента к другому, вдоль этих соединений? В какие другие элементы можно перейти из заданного элемента? Какой путь от одного элемента к другому считается наилучшим?

Чтобы смоделировать подобную ситуацию, мы пользуемся объектами, которые называются *графами* (*graphs*). В этой главе мы подвергнем подробному анализу основные свойства графов, тем самым подготавливая базу для изучения всевозможных алгоритмов, которые помогут ответить на вопросы, подобные сформулированным выше. Эти алгоритмы эффективно используют различные вычислительные средства, которые рассматривались в частях 1–4 многотомника. Они также служат той базой, без которой невозможно подступиться к проблемам, возникающим в важных приложениях, и решение которых нельзя представить без привлечения солидной алгоритмической технологии.

Теория графов, будучи крупной ветвью комбинаторной математики, интенсивно изучалась в течение не одной сотни лет. Было выявлено много важных и полезных свойств графов, однако многие задачи еще ждут своего решения. В этой книге, признавая, что еще многое предстоит изучить, мы извлекаем из всего массива обширных знаний о графах только то, что нам необходимо понять, и используем из всего разнообразия фундаментальных алгоритмов лишь те из них, которые в представляют несомненную практическую ценность.

Как и множество других предметных областей, которые приходилось изучать, алгоритмическое исследование графов возникло сравнительно недавно. И хотя некоторые фундаментальные алгоритмы открыты давно, все же большая часть интереснейших алгоритмов получена в течение нескольких последних десятилетий. Даже простейшие алгоритмы на графах позволяют получить полезные компьютерные программы, а изучаемые нами нетривиальные алгоритмы относятся к числу наиболее элегантных и интересных из известных алгоритмов.

Чтобы продемонстрировать все разнообразие приложений, использующих графы для обработки данных, начнем исследования алгоритмов в этой благодатной области с рассмотрения нескольких примеров.

Географические карты. Путешественник, прежде чем отправиться в путь, желает получить ответ на вопросы типа: "Какой маршрут из Принстона в Сан-Хосе потребует *наименьших расходов*?" Пассажир, для которого время дороже денег, хотел бы получить ответ на такой вопрос: "Каким путем *быстрее всего* добраться из Принстона в Сан-Хосе?" Чтобы ответить на вопросы подобного рода, мы выполняем обработку информации, характеризующей *соединения* (пути следования) между *элементами* (городами и населенными пунктами).

Гипертекст. Когда мы просматриваем Web-каталоги, мы сталкиваемся с документами, содержащими различные ссылки (соединения) на другие документы, и переходим от документа к документу, щелкая мышью на этих ссылках. Сама по себе "всемирная паутина" представляет собой граф, в котором в качестве элементов выступают документы, а соединения суть связи. Алгоритмы обработки графов являются важными компонентами поисковых механизмов, которые помогают определить местоположение информации в Web.

Микросхемы. Микросхемы содержат такие элементы, как транзисторы, резисторы, конденсаторы, которые связаны между собой сложнейшим образом. Для управления машинами, изготавливающими микросхемы, и для проверки, выполняют ли эти цепи заданные функции, используются компьютеры. Мы хотим получить ответы на простые вопросы наподобие "Имеются ли в цепи условия для короткого замыкания?" и на более сложные вопросы, такие как "Можно ли скомпоновать микросхему на кристалле таким образом, чтобы шины не пересекались?". В данном случае, ответ на первый вопрос зависит только от свойств соединений (шин), в то время как для ответа на второй вопрос потребуется подробная информация о шинах, элементах, которые эти шины соединяют, и физических ограничениях, накладываемых кристаллом.

Составление расписаний. Производственный процесс требует решения различных задач под воздействием некоторого множества ограничений, по условиям которых решение одной задачи не может быть начато до тех пор, пока не будет завершено решение другой задачи. Мы представляем эти ограничения в виде соединений между этими задачами (элементами), при этом перед нами возникает *задача составления расписаний* (*scheduling*) в ее классическом виде: как построить временной график решения задач таким образом, чтобы удовлетворить заданным ограничениям и завершить данный процесс за минимально возможное время?

Транзакции. Телефонная компания поддерживает базу данных телефонного трафика. В этом случае соединения представляют телефонные вызовы. Мы заинтересованы в том,

чтобы знать все, что касается характера структуры соединений, ибо хотим проложить провода и установить коммутаторы так, чтобы эффективно справляться с телефонным трафиком. Еще одним примером может служить то, как финансовое учреждение отслеживает операции купли/продажи на рынке. Соединение в рассматриваемом случае представляет собой передачу денег продавцом покупателю. Знание свойств структуры соединений в данном случае помогает лучше понять особенности рынка.

Задачи поиска сочетаний. Студенты обращаются с заявлениями на замещение должностей в таких общественных организациях, как общественные клубы, университеты или высшие медицинские учебные заведения. Элементы соответствуют студентам и институтам, тогда как связи соответствуют приложениям. Мы хотим найти методы, устанавливающие соответствие между вакантными должностями и заинтересованными в их получении студентами.

Сети. Сеть вычислительных машин состоит из взаимосвязанных узлов, которые посылают, передают дальше и получают сообщения различных видов. Мы заинтересованы не только в обеспечении возможности получать сообщения из любого другого узла, но и в том, чтобы эта возможность сохранялась для всех пар узлов и в случае изменения конфигурации сети. Например, возможно, потребуется проверить работу конкретной сети, чтобы убедиться в отсутствии некоторого небольшого подмножества узлов или соединений, настолько критичного для работы всей сети, что его утрата может привести к разъединению остальных пар узлов.

Структура программы. Компилятор строит графы для представления структуры вызовов крупной системы программного обеспечения. Элементами в этом случае являются различные функции или программные модули, составляющие систему; соединения отождествляются либо с возможностью того, что одна функция может вызвать другую функцию (статический анализ), или с фактическим вызовом, когда система находится в рабочем состоянии (динамический анализ). Нам нужно выполнить анализ графа, чтобы определить, как достичь максимальной эффективности при выделении системных ресурсов для конкретной программы.

Эти примеры демонстрируют, насколько широк диапазон приложений, для которых граф служит подходящей абстракцией, и, соответственно, диапазон вычислительных задач, с которыми доведется столкнуться при работе с графами. Эти задачи являются главной темой настоящей книги. Во многих подобного рода приложениях, с которыми приходится иметь дело на практике, объем обрабатываемых данных поистине огромен, так что именно эффективность алгоритма решает, будет ли вообще работать соответствующее приложение.

Нам уже приходилось сталкиваться с графами в части 1. В самом деле, первые алгоритмы, которые мы изучали самым подробным образом, т.е. алгоритмы объединения-поиска, описанные в главе 1, представляют собой простейшие алгоритмы на графах. Мы также использовали графы в главе 3 в качестве иллюстрации приложения двухмерных массивов и связных списков, в главе 5 графы послужили иллюстрацией отношений между рекурсивными программами и фундаментальными структурами данных. Любая связная структура данных может быть представлена в виде графа, а некоторые известные алгоритмы обработки деревьев и других связных структур представляют собой частные случаи алгоритмов на графах. Назначение данной главы заключается в том, чтобы обеспечить

контекст для изучения широкого набора алгоритмов на графах, от простейших, описанных в части 1, до очень сложных, описываемых в главах 18–22.

Как всегда, мы хотим знать, какой из возможных алгоритмов решения конкретной задачи обладает максимальной эффективностью. Задача изучения рабочих характеристик алгоритмов на графах представляет собой достаточно сложную проблему, что можно объяснить следующими причинами:

- Стоимость алгоритма зависит не только от свойств множества элементов, но также и от многочисленных свойств соединений (и глобальных свойств графа, обусловленных свойствами соединений).
- Трудность разработки точных моделей типов графов, с которыми, возможно, придется столкнуться.

Мы часто исходим из предположения, что используемым алгоритмам на графах придется работать в условиях, наиболее неблагоприятных для их рабочих характеристик, даже если во многих случаях на практике такие предположения представляют собой пессимистическую оценку. К счастью, как мы убедимся далее, многие из этих алгоритмов *оптимальны* и не требуют больших непроизводительных затрат. В отличие от них, существуют алгоритмы, которые затрачивают одинаковые ресурсы на обработку всех графов заданного размера. Мы можем с достаточной степенью точности предсказать, как поведут себя такие алгоритмы в конкретных условиях. В тех случаях, когда такие предсказания невозможны, нам придется уделить особое внимание свойствам графов различных типов, которые, в соответствии с нашими ожиданиями, проявятся в конкретных практических ситуациях, и оценить, как эти свойства могут повлиять на производительность выбранных алгоритмов.

Мы начнем с изучения основных определений графов и свойств графов, а также с освоения стандартной терминологии, которая используется для их описания. Затем мы дадим определения интерфейсов АТД (ATD, abstract data type – абстрактный тип данных), которыми будем пользоваться при изучении алгоритмов на графах, двух наиболее важных структур данных, применяемых для представления графов – *матрицы смежности* (*adjacency-matrix*) и *списков смежных вершин* (*adjacency-lists*), а также различных подходов к реализации базовых функций над абстрактными типами данных (АТД-функций). Затем мы рассмотрим клиентские программы, способные строить случайные графы, которые могут использоваться для тестирования разработанных алгоритмов и для изучения свойств графов. Весь этот материал служит фундаментом, позволяющим применять алгоритмы обработки графов для решения трех классических задач, связанных с поиском путей в графах, которые служат также иллюстрацией того, что по сложности задачи обработки графов существенно отличаются друг от друга даже в тех случаях, когда различия между ними далеко не очевидны. Эта глава завершается обзором наиболее важных задач обработки графов, которые будут рассматриваться в этой книге, в контексте сложности их решения.

17.1. Глоссарий

С графами связана обширная терминология. Большинство употребляемых терминов имеют прямое определение, и для удобства ссылок целесообразно рассматривать их в каком-то одном месте, а именно, в данном разделе. Некоторые из понятий мы уже употребляли в главе 1 при изучении базовых алгоритмов, другие останутся невостребованными до тех пор, пока мы не перейдем к изучению соответствующих им новейших алгоритмов в главах 18-22.

Определение 17.1. Граф *есть некоторое множество* **вершин** *и некоторое множество* **ребер**, *соединяющих пары различных вершин (одно ребро может соединять максимум одну пару вершин).*

Мы используем цифры от **0** до **V-1** в качестве имен вершин графа, состоящего из V вершин. Основная причина выбора именно этой системы обозначений заключается в том, что мы получаем быстрый доступ к информации, соответствующей каждой вершине, путем индексирования векторов. В разделе 17.6 будет рассмотрена программа, которая применяет таблицу идентификаторов с тем, чтобы установить отображение "один-к-одному" с целью связать V произвольных имен с вершин с V целыми числами в диапазоне от 0 до $V-1$. Имея в своем распоряжении такую программу, мы можем употреблять индексы как имена вершин (для удобства обозначений) без ущерба для универсальности рассуждений. Иногда мы будем предполагать, что множество вершин определено неявно, взяв за основу для определения графов множество ребер и учитывая только те вершины, которые закреплены, по меньшей мере, за одним ребром. Во избежание громоздких выражений, как то: "граф, состоящий из 10 вершин со следующим набором ребер", мы часто явно не указываем числа вершин, если оно следует из контекста. Далее будем придерживаться соглашения, в соответствии с которым число вершин в заданном графе всегда обозначается через V, а число ребер – через E.

В качестве стандартного определения графа (с которым первый раз довелось столкнуться в главе 5) будет принято определение 17.1, но при этом заметим, что в нем использованы два технических упрощения. Во-первых, оно не позволяет дублировать ребра (математики иногда называют такие ребра *параллельными* (*parallel*), а граф, который может содержать такие ребра, *мультиграфом* (*multigraph*)). Во-вторых, оно не допускает ребер, замыкающихся на одну и ту же вершину; такое ребро называются *петлей* (*self-loop*). Графы, в которых нет параллельных ребер или петлей, иногда называют *простыми графами* (*simple graph*).

Мы употребляем понятия простых графов в формальных определениях, поскольку таким путем легче выразить их основные свойства, а также ввиду того, что параллельные ребра и петли во многих приложениях не нужны. Например, мы можем ограничить число ребер простого графа заданным числом вершин.

Свойство 17.1. *Граф, состоящий из V вершин, содержит не более* $V(V-1)/2$ *ребер.*

Доказательство: Общее число возможных пар вершин равно V^2, в том числе V петель, при этом связи между различными вершинами учитываются дважды, следовательно, максимальное число ребер не превосходит значения $(V2 - V)/2 = V(V-1)/2$. ■

Эти ограничения не имеют места в условиях существования параллельных ребер: граф, не принадлежащий к категории простейших, может содержать всего лишь две вершины и миллиарды ребер, соединяющие их (или даже одну вершину и миллиарды петель).

В некоторых приложениях удаление параллельных ребер и петель можно рассматривать как задачу обработки данных, которую должны решать сами приложения. В других приложениях, возможно, вообще не имеет смысла проверять, представляет ли заданный набор ребер простой граф. На протяжении данной книги, там, где удобно проводить анализ конкретного приложения или разрабатывать алгоритм с применением расширенного определения графа, использующего понятия параллельных ребер или петель, мы будем это делать. Например, петли играют важнейшую роль в классическом алгоритме, который будем изучаться в разделе 17.4; параллельные ребра широко используются в приложениях, которые будут рассматриваться в главе 22. В общем случае из контекста ясно, в каком смысле используется термин "граф": "простой граф", "мультиграф" или "мультиграф с петлями".

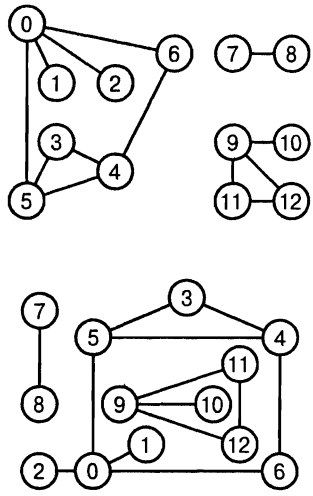
Математики употребляют термины *вершина* (*vertex*) и *узел* (*node*) попеременно, но мы будем главным образом пользоваться термином *вершина* при обсуждении графов и термином *узел* при обсуждении представлений графов, например, структур данных в C++. Обычно мы полагаем, что у вершины есть имя и что она может нести другую связанную с ней информацию. Аналогично, слова *дуга* (*arc*), *ребро* (*edge*) и *связь* (*link*) также широко используются математиками для описания абстракций, предусматривающих соединение двух вершин, однако мы последовательно будем употреблять термин *ребро* при обсуждении графов и термин *связь* при обсуждении структур данных в C++.

Если имеется ребро, соединяющее две вершины, будем говорить, что обе эти вершины *смежные* (*adjacent*) по отношению друг к другу, а ребро *инцидентно* (*incident on*) этим вершинам. Степень (*degree*) вершины есть число ребер, инцидентных этой вершине. Мы употребляем обозначение **v-w** для обозначения ребра, соединяющего вершины **v** и **w**; обозначение **w-v** представляет собой еще одно возможное обозначение того же ребра.

Подграф (*subgraph*) представляет собой подмножество ребер некоторого графа (и связанных за ними вершин), которые сами образуют граф. По условиям многих вычислительных задач требуется определить на некотором графе подграфы различных типов. Если выделить некоторое подмножество вершин графа и все ребра графа, соединяющие пары вершин этого подмножества, то такое подмножество называется *индуцированным подграфом* (*induced subgraph*), ассоциированным с этими вершинами.

Мы можем начертить граф, обозная точками его вершины и соединяя эти точки линиями, которые будут служить ребрами. Чертеж дает нам некоторое представление о структуре графа; но это представление может оказаться обманчивым, ибо определение графа дается вне зависимости от его представления. Например, два чертежа и список ребер, которые приводятся на рис. 17.1, представляют один и тот же граф, поскольку этот граф – всего лишь (неупорядоченное) множество его вершин и (неупорядоченное) множество его ребер (пар вершин), и ничего более. Достаточно рассматривать граф просто как некоторое множество ребер, однако мы будем изучать и другие представления, которые лучше других подходят для использования в качестве базы для структур данных типа граф, рассматриваемых в разделе 17.4.

Перенесение вершин заданного графа на плоскость и вычерчивание этих вершин и соединяющих их ребер позволяет получить *чертеж графа* (*graph drawing*). Возможно множество вариантов размещения вершин, стилей изображения ребер, поскольку эстетические требования, предъявляемые к изображению, практически бесконечны. Алгоритмы построения чертежей, соблюдающие различные естественные ограничения, подверглись интенсивному изучению, в результате которых были получены многие удачные приложения (*см. раздел ссылок*). Например, одно из простейших ограничений есть требование, согласно которому ребра не должны пересекаться. *Планарный граф* (*planar graph*) принадлежит к числу тех, которые можно построить без пересечения ребер. В зависимости от того, является ли граф планарным (плоским) или нет, возникает заманчивая задача, которой мы коротко коснемся в разделе 17.8. Возможность построения чертежа графа представляет собой благодатное поле для исследований, в то же время на практике построить хороший чертеж графа не так-то просто. Многие графы с очень большим числом вершин и ребер, являются абстрактными объектами, чертеж которых неосуществим.



0-5 5-4 7-8
4-3 0-2 9-11
0-1 11-12 5-3
9-12 9-10
6-4 0-6

РИСУНОК 17.1. ТРИ РАЗЛИЧНЫХ ПРЕДСТАВЛЕНИЯ ОДНОГО И ТОГО ЖЕ ГРАФА

Граф определяется его вершинами и его ребрами, но не способом его изображения на чертеже. Оба чертежа изображают один и тот же граф, этот же граф представлен списком ребер (внизу), при этом учитывается то обстоятельство, что рассматриваемый граф содержит 13 вершин, помеченных номерами от 0 до 12.

В некоторых приложениях, например, выполняющих анализ географических карт или электрических схем, для построения чертежа графа требуется обширная информационная база, поскольку их вершины соответствуют точкам на плоскости, а расстояния между ними должны быть выдержаны в определенном масштабе. Мы называем такие графы *эвклидовыми* (*Euclidean graph*). Во множестве других приложений, имеющих дело с графами, представляющими отношения или временные графики, они просто служат носителями информации о связности, при этом даже не предъявляются какие-либо требования к геометрическому расположению вершин. Мы рассмотрим примеры алгоритмов, которые используют геометрическую информацию эвклидовых графов, в главах 20 и 21, но сначала поработаем с алгоритмами, которые вообще не используют геометрическую информацию, и подчеркнем, что графы в общем случае не зависят от конкретного представления в виде чертежа или данных в компьютере.

Если целиком сосредоточиться на исследовании свойств соединений, то мы, возможно, предпочтем рассматривать метки вершин как предназначенные только для удобства обозначения, а два графа считать одинаковыми, если они отличаются друг от друга только метками вершин. Два графа называются *изоморфными* (*isomorphic*), если можно поменять метки вершин на одном из них таким образом, чтобы набор ребер этого графа стал идентичным набору ребер другого графа. Обнаружение изоморфизма двух графов представляет собой сложную вычислительную задачу (см. рис.17.2 и упражнение 17.5). Ее слож-

ность объясняется тем обстоятельством, что существует $V!$ способов обозначения вершин, т.е. их слишком много, чтобы перепробовать каждый из них. Поэтому, несмотря на потенциальную привлекательность уменьшения числа различных структур графа, которое становиться возможным, если рассматривать изоморфные графы как идентичные структуры, это делается довольно-таки редко.

Как и в случае деревьев, которые мы изучали в главе 5, нас очень часто интересуют базовые структурные свойства, которые мы можем определить, рассматривая характерные последовательности ребер в графе.

Определение 17.2. Путь (path) *в графе есть последовательность вершин, в которой каждая следующая вершина (после первой), является смежной с предыдущей вершиной на этом пути. Все вершины и ребра, составляющие простой путь, различны.* **Циклом (cycle)** *называется простой путь, у которого первая и последняя вершина одна и та же.*

Иногда мы используем термин *циклический путь* (*cyclic path*) для обозначения пути, у которого первая и последняя вершина одна и та же (и который в других отношениях не обязательно является простым); мы также употребляем термин *контур* (*tour*) для обозначения циклического пути, который включает каждую вершину. Эквивалентное определение рассматривает путь как последовательность ребер, которая соединяет соседние вершины. Мы отражаем это обстоятельство в используемых нами обозначениях, соединяя имена вершин в путь точно так, как мы соединяем их в представлении ребра. Например, простой путь, показанный на 17.1, содержит последовательности **3-4-6-0-2** и **9-11-12**, а циклами графа являются последовательности **0-6-4-3-5-0** и **5-4-3-5**. По определению *длина* (*length*) пути или цикла представляет собой количество образующих их ребер.

Мы принимаем соглашение, по которому каждая отдельная вершина есть путь длины 0 (путь из некоторой вершины в ту же вершину, не содержащий ребер, отличных от петель). Помимо этого соглашения, в графе, не содержащем параллельных ребер и петель, в котором конкретная пара вершин однозначно определяет некоторое ребро, пути должны состоять, по меньшей мере, из двух различных вершин, а циклы должны содержать, по меньшей мере, три различных ребра и три различных вершины.

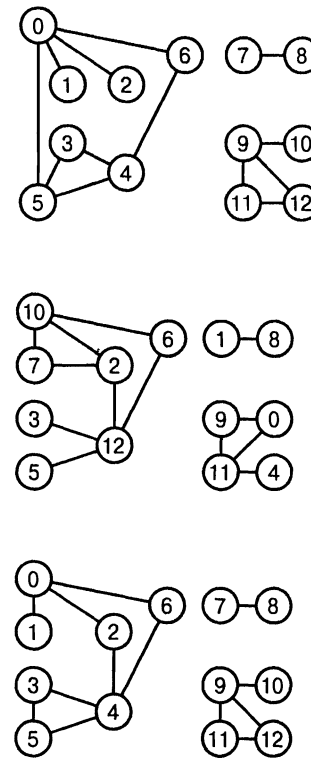


РИСУНОК 17.2. ПРИМЕРЫ ИЗОМОРФИЗМА ГРАФОВ

*Два верхних графа изоморфны, поскольку можно переобозначить вершины таким образом, что оба набора ребер становятся идентичными (чтобы сделать граф в середине таким же, как и верхний граф, поменяйте **10** на **4**, **7** на **3**, **2** на **5**, **3** на **1**, **12** на **0**, **5** на **2**, **9** на **11**, **0** на **12**, **11** на **9**, **1** на **7** и **4** на **10**). Нижний граф не изоморфен двум другим, поскольку не существует такого способа переименования его вершин, чтобы множество его ребер стало идентично аналогичным множествам двух первых графов.*

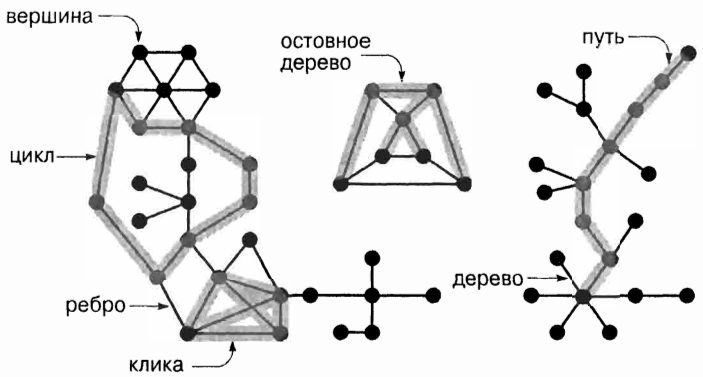


РИСУНОК 17.3. ТЕРМИНОЛОГИЯ, УПОТРЕБЛЯЕМАЯ В ТЕОРИИ ГРАФОВ

Изображенный на диаграмме граф содержит 55 вершин, 70 ребер и 3 связных компоненты. Одна из связных компонент представляет собой дерево (справа). В графе имеется множество циклов, один из этих циклов выделен как крупная связная компонента (слева). На диаграмме также показано остовное дерево, содержащееся в связной компоненте небольшого размера (в центре). Как единое целое, рассматриваемый граф не содержит остовных деревьев, поскольку он не является связным графом.

Мы называем два простых пути *непересекающимися* (*disjoint*), если они не содержат общих вершин, кроме, разве что, их конечных точек. Это условие несколько слабее, чем требование отсутствия в обоих путях каких-либо общих вершин, и более полезное, поскольку в этом случае мы можем соединять простые непересекающиеся пути из s в t и из t в u и получать простой непересекающийся путь из s в u, если вершины s и u различны. Иногда используется термин *непересекающиеся по вершинам* (*vertex disjoint*), чтобы отличить эту ситуацию от более сильного условия *непересекающиеся по ребрам* (*edge disjoint*), когда мы требуем, чтобы пути не имели общих ребер.

Определение 17.3. *Граф называется* **связным графом (connected graph)**, *если существует путь из каждой вершины в любую другую вершину графа. Несвязный граф состоит из некоторого множества связных компонент, которые представляют собой максимальные связные подграфы.*

Термин *максимальный связный подграф* (*maximal connected subgraph*) означает, что не существует пути из вершины такого подграфа в любую другую вершину графа, который не содержался бы в подграфе. Интуитивно, если бы вершины были физическими объектами, такими как, скажем, узлы или бусинки, а ребра были бы физическими соединениями, такими как, например, нити или провода, то в случае связного графа, находясь в какой-либо его вершине, можно было бы смотать нить в один клубок, в то время как несвязный граф распался бы на два или большее число таких клубков.

Определение 17.4. *Ациклический связный граф называется* **деревом (tree)** *(см. главу 5). Множество деревьев называется* **лесом (forest)**. **Остовное дерево (spanning tree)** *связного графа есть подграф, который содержит все вершины этого графа и представляет собой единое дерево.* **Остовный лес (spanning forest)** *графа есть подграф, который содержит все вершины этого графа и при этом является лесом.*

Например, граф, изображенный на рис. 17.1, имеет три связных компоненты и охватывается лесом **7-8 9-10 9-11 9-12 0-1 0-2 0-5 5-3 5-4 4-6** (существует множество других остовных лесов). На рис. 17.3 эти и некоторые другие свойства отображены на большем из графов.

Более подробно мы исследуем деревья в главе 4, теперь же рассмотрим различные эквивалентные определения. Например, граф G с V вершинами есть дерево тогда и только тогда, когда он удовлетворяет одному из четырех условий:

- G содержит $V-1$ ребро и ни одного цикла.
- G содержит $V-1$ ребро и представляет собой связный граф.
- Каждую пару вершин в G соединяет в точности один простой путь.
- G представляет собой связный граф, в то же время при удалении любого из ребер он перестает быть связным.

Любое из указанных выше условий необходимо и достаточно для доказательства остальных трех, и на их основе можно вывести другие свойства деревьев (см. упражнение 17.1). Формально, мы должны выбрать одно из указанных условий в качестве определения; фактически, они все вместе могут служить определениями и свободно применяться при выборе, например, "ациклического связного графа" в определении 17.4.

Графы, у которых присутствуют все ребра, называются *полными графами* (*complete graph*) (см. рис. 17.4). Мы определяем *дополнение* (*complement*) графа G методом построения, взяв для начала полный граф, имеющий то же число вершин, что и исходный графа G, и удалив из него все ребра графа G. *Объединением* (*union*) двух графов является граф, порожденный объединением множеств ребер этих графов. Объединение графа и его дополнения есть полный граф. Все графы, имеющие V вершин суть подграфы полного графа с V вершинами. Общее число различных графов с V вершинами равно $2^{V(V-1)/2}$ (число различных способов выбора подмножеств из $V(V-1)/2$ возможных ребер). Полный подграф называется *кликой* (*clique*).

Большинство графов, с какими нам приходится сталкиваться на практике, содержат лишь небольшую часть всех возможных ребер. Чтобы представить это понятие в числовом выражении, положим *насыщенность* (*density*) графа равной среднему значению степеней его вершин, т.е. $2E/V$. Насыщенный граф есть граф, средняя степень вершин которого про-

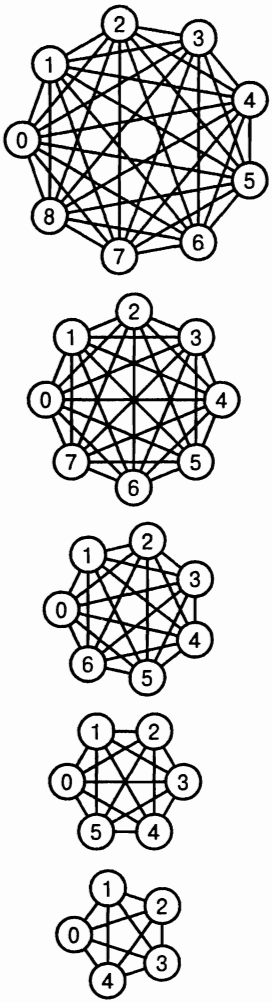


РИСУНОК 17.4. ПОЛНЫЕ ГРАФЫ

Представленные на диаграмме полные графы, в которых каждая вершина соединена с любой другой вершиной, содержат, соответственно, 10, 15, 21, 28 и 36 ребер (снизу вверх). Каждый граф, содержащий от 5 до 9 вершин (существует более чем 68 миллиардов таких графов) есть подграф одного из этих графов.

порциональна V; *разреженный граф* (*sparse graph*) есть граф, дополнение которого насыщено. Другими словами, мы считаем граф насыщенным, если число его ребер E пропорционально V^2 и разреженным в противном случае. Такое "асимптотическое" определение недостаточно точно характеризует тот или иной граф, однако общая картина ясна: можно с уверенностью утверждать, что граф, который состоит из миллиона вершин и десятков миллионов ребер есть разреженный граф, в то время как граф, который состоит из нескольких тысяч вершин и миллионов ребер, есть плотный граф. Мы еще можем рассчитывать на то, что нам удастся успешно выполнить обработку разреженного графа, однако плотный граф с миллиардами вершин содержит несметное количество ребер.

Информация о том, с каким графом мы имеем дело, с плотным или разреженным, в общем случае является ключевым фактором выбора эффективного алгоритма обработки графа. Например, для решения той или иной задачи мы можем разработать два алгоритма, причем первому из них для ее решения понадобится V^2 действий, а другому – $E \lg E$ действий. Эти формулы показывают, что второй алгоритм лучше подходит для разреженных алгоритмов, в то время как первому алгоритму следует отдавать предпочтение при обработке плотного графа. Например, плотный граф с многими миллионами ребер может иметь всего лишь несколько тысяч вершин: в данном случае V^2 и E суть величины одного порядка, при этом быстродействие алгоритма V^2 в 20 раз выше, чем быстродействие алгоритма $E \lg E$. С другой стороны, разреженный граф с миллионами ребер обладает миллионами вершин, следовательно, алгоритм $E \lg E$ будет в *миллионы* раз быстрее алгоритма V^2. Мы можем прийти к различным компромиссам на основании более подробного изучения этих формул, но в общем случае для практических целей вполне достаточно терминов *разреженный* (*sparse*) и *насыщенный* (*dense*), чтобы можно было получить представление об основных рабочих характеристиках требуемых алгоритмов.

При анализе алгоритмов обработки графов мы полагаем, что значения V/E ограничены сверху небольшой константой, благодаря чему мы можем упростить такие выражения как $V(V+E)$ до VE. Это предположение подтверждается только в тех случаях, когда число ребер невелико по сравнению с числом вершин, что представляет собой редкую ситуацию. Как правило, число ребер намного превосходит число вершин (V/E намного меньше 1).

Двухдольный граф (*bipartite graph*) есть граф, множество вершин которого можно разделить на такие два подмножества, что любое ребро соединяют вершину одного подмножества только с вершиной другого подмножества. На рис. 17.5 приводится пример двухдольного графа. Двухдольные графы естественным образом возникают во многих ситуациях, таких как задачи поиска соответствий, описанные в начале этой главы. Любой подграф двухдольного графа сохраняет это свойство.

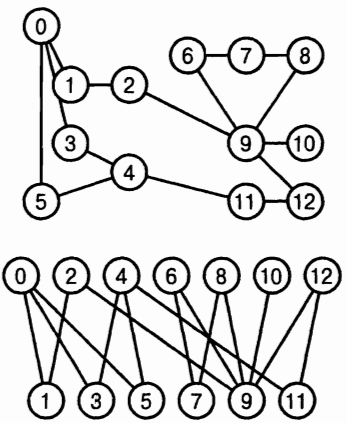


РИСУНОК 17.5. ДВУХДОЛЬНЫЙ ГРАФ
Все ребра этого графа соединяют вершины с нечетными номерами с вершинами с четными номерами, т.е. это двухдольный граф. Нижняя диаграмма делает это свойство очевидным.

Графы, которые мы рассматривали до сих пор, носят название *неориентированных графов* (*undirected graphs*). В ориентированных графах (*directed graphs*), из-

вестных еще как *орграфы* (*orgraph*), ребра однонаправленные: мы рассматриваем пару вершин, определяющую конкретное ребро, как *упорядоченную* (*ordered*) пару, которая определяет однонаправленную смежность в том смысле, что возможность перехода из первой вершины во вторую отнюдь не означает переход из второй вершины в первую. Многие применения (например, графы, представляющие Web, графы, используемые при составлении расписаний, или графы, применяемые для обработки телефонных звонков) естественным образом описываются в виде орграфов.

Ребра в орграфах мы называем *ориентированными ребрами* (*directed edges*), хотя в общем случае это свойство вытекает из контекста (некоторые авторы для обозначения ориентированных ребер применяют термин *дуга* (*arc*)). Первая вершина ориентированного ребра называется *началом* (*source*); вторая вершина называется *концом* (*destination*). (Некоторые авторы употребляют, соответственно, термины *хвост* (*tail*) и *голова* (*head*), чтобы отличить вершины ориентированного графа, однако мы будем избегать таких обозначений, поскольку мы употребляем эти же термины при реализации структур данных.) На диаграммах мы изображаем ориентированные ребра в виде стрелок, направленных из начала в конец, и часто говорим, что ребро *указывает* (*points*) на ту или иную вершину. Когда мы используем обозначение **w-v** по отношению к орграфу, мы делаем это с целью показать, что ребро, которое исходит из **w** и заходит в **v**, отличается от ребра **v-w**, которое исходит из **v** и заходит в **w**. Мы также говорим о *полустепени исхода* (*outdegree*) и *полустепени захода* (*indegree*) некоторой вершины (соответственно, число ребер, для которых она служит началом, и число ребер, для которых она служит концом).

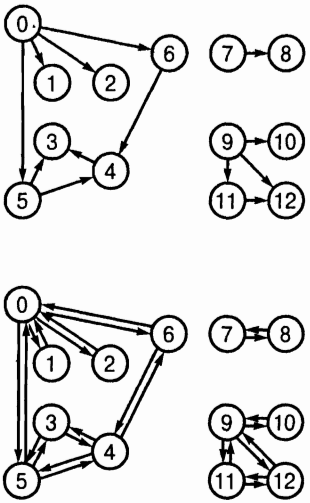


РИСУНОК 17.6. ДВА ОРГРАФА

Чертеж вверху есть представление графа, приведенного в качестве примера на рис. 17.1, интерпретируемое как ориентированный граф, при этом мы рассматриваем ребра как упорядоченные пары и изображаем их в виде стрелок, ведущих из первой вершины во вторую. Этот граф является DAG-графом. Чертеж в нижней части рисунка – это представление неориентированного графа, показанного на рис. 17.1, который может служить иллюстрацией способа, обычно выбираемого нами для представления неориентированных графов: в виде орграфа, в котором каждому соединению соответствуют два ребра (по одному в каждом направлении).

Иногда целесообразно рассматривать неориентированный граф как орграф, у которого имеются два ориентированных ребра (по одному в каждом направлении); в других случаях полезно рассматривать неориентированный граф просто как некоторую совокупность соединений. Обычно для ориентированных и для неориентированных графов (см. рис. 17.6) мы используем одно и то же представление, о чем подробно пойдет речь в разделе 17.4. Иначе говоря, в общем случае мы поддерживаем два представления каждого ребра неориентированных графов, каждое из них указывает в одном из направлений, так что мы имеем возможность сразу же ответить на вопросы типа "Какие вершины соединены с вершиной *v*?"

Глава 19 посвящена изучению структурных свойств орграфов; в общем случае они более сложные, чем соответствующие свойства неориентированных графов. *Направленный цикл* (*directed cycle*) в орграфе – это цикл, в котором пары смежных вершин появляются в порядке, указанном (устанавливаемом) ребрами графа. *Граф DAG* (*Directed Acyclic Graph – ориентированный ациклический граф*) есть орграф, который не содержит направленных циклов. DAG-граф (ациклический орграф) и дерево (ациклический неориентированный граф) – это отнюдь не тождественные понятия. Иногда мы обращаемся к *базовому неориентированному графу* (*underlying undirected graph*), лежащему в основе орграфа, подразумевая под ним неориентированный граф, определяемый тем же множеством ребер, при этом, однако, эти ребра не рассматриваются как ориентированные.

В главах 20–22 содержится главным образом анализ алгоритмов решения различных вычислительных задач, связанных с использованием графов, в которых в виде вершин и ребер представлена другая информация. В случае *взвешенного графа* (*weighted graph*) с каждым ребром мы связываем числа (*weights – веса*), которые в общем случае представляют собой расстояние либо стоимость. Мы также можем присвоить вес каждой вершине либо несколько весов каждой вершине и каждому ребру. В главе 20 мы будем изучать взвешенные неориентированные графы, которые мы обычно называем *сетями* (*networks*). Алгоритмы в главе 22 решают классические задачи, которые возникают в конкретных интерпретациях сетей, известных как *транспортные сети* (*flow network*).

Уже из главы 1 стало ясно, что комбинаторная структура графа получила широкое распространение. Масштаб распространения этой структуры тем более поразителен в связи с тем, что начало ей дала простая математическая абстракция. Эта лежащая в основе простота отражается во многих программных кодах, которые мы разрабатываем для базовой обработки графов. Тем не менее, такая простота часто заслоняет собой сложные динамические свойства, которые требуют глубокого понимания комбинаторных свойств самих графов. Зачастую очень трудно убедить самих себя в том, что алгоритм работает на графе в соответствии с замыслом, поскольку программа получается настолько компактной, что в это даже трудно поверить.

Упражнения

17.1. Докажите, что в любом ациклическом связном графе с V вершинами имеется $V-1$ ребро.

▷ **17.2**. Постройте все связные подграфы графа

0-1, 0-2, 0-3, 1-3, 2-3.

▷ **17.3**. Составьте список неизоморфных циклов графа, представленного на рис. 17.1. Например, если в вашем списке содержится цикл **3-4-5-4**, в нем не могут находиться циклы **3-5-4-3**, **3-5-3-4**, **4-3-5-4**, **5-3-4-5** или **5-4-3-5**.

17.4. Исследуйте граф

4-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

Определите число связных компонент, постройте остовный лес, составьте список простых путей, содержащих, по меньшей мере, три вершины, а также список всех неизоморфных циклов (см. упражнение 17.1).

○ **17.5**. Исследуйте граф, заданный следующими четырьмя наборами ребер:

```
0-1 0-2 0-3 1-3 1-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
0-1 0-2 0-3 0-3 1-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
0-1 1-2 1-3 0-3 0-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
4-1 7-9 6-2 7-3 5-0 0-2 0-8 1-6 3-9 6-3 2-8 1-5 9-8 4-5 4-7
```

Какие из этих графов изоморфны друг другу? Какой из этих графов планарный?

17.6. Какой процент из более чем 68 миллиардов графов, о которых шла речь в подписи к рис. 17.4, состоит из менее чем девяти вершин?

▷ **17.7**. Сколько различных подграфов содержатся в заданном графе с V вершинами и E ребрами?

• **17.8**. Определите жесткие верхние и нижние границы числа связных компонент графа с V вершинами и E ребрами.

○ **17.9**. Сколько существует неориентированных графов, содержащих V вершин и E ребер?

•••**17.10**. Если считать графы различными, только когда они не изоморфны, сколько существует различных графов, содержащих V вершин и E ребер?

17.11. Сколько графов, содержащих V вершин, можно отнести к числу двухдольных графов?

17.2. АТД графа

Мы разрабатываем наши алгоритмы обработки графов, используя для этой цели абстрактный тип данных (АТД), который позволяет сформулировать фундаментальные задачи, используя стандартный механизм, введенный в главе 4. Программа 17.1 представляет собой интерфейс АТД, который будет использоваться для этих целей. Базовые представления и реализации графа для этого АТД рассматриваются в главах 17.3–17.5. Далее в этой книге, всякий раз, когда мы будем рассматривать новые задачи обработки графов, мы будем также проводить исследования алгоритмов их решения и реализации этих алгоритмов в контексте клиентских программ и абстрактных типов данных, которые получают доступ к графам через этот интерфейс. Такая схема позволит решать задачи обработки графов в диапазоне от элементарных функций поддержки до сложных решений трудных проблем.

В основу рассматриваемого интерфейса положен применяемый нами стандартный механизм, который скрывает представления и реализации от клиентских программ (см. раздел 4.8). Он также включает определение простой структуры, которая позволяет нашим программам манипулировать ребрами некоторым единообразным способом. Такой интерфейс предоставляет в распоряжение клиентов базовый механизм, который позволяет им строить графы (сначала строится граф, затем к нему добавляются ребра), осуществлять поддержку графов (путем удаления некоторых ребер и добавления новых) и исследовать графы (путем использования итератора для обработки вершин, смежных по отношению к заданной вершине).

Программа 17.1. Интерфейс АТД графа

Этот интерфейс представляет собой отправную точку для реализации и тестирования алгоритмов обработки данных. Он определяет два типа данных: тривиальный тип данных **Edge**, включающий функцию конструктора, которая строит ребро по двум заданным вершинам, и тип данных **GRAPH**, определение которого дается в соответствии с методологией стандартного, независимого от представления интерфейса АТД, описанного в главе 4.

Конструктор **GRAPH** принимает два аргумента: целое, задающее число вершин, и булевское значение, которое показывает, является ли граф ориентированным или неориентированным (орграф), при этом неориентированный граф подразумевается по умолчанию.

Базовые операции, которые мы используем для обработки графов и орграфов, суть функции АТД, обеспечивающие их построение и уничтожение, функции для подсчета числа вершин и ребер, а также для добавления и удаления ребер. Класс итератора **adjIterator** позволяет клиентам проводить обработку любых вершин, смежных по отношению к заданной вершине. Программы 17.2 и 17.3 служат иллюстрацией его использования.

```
struct Edge
  { int v, w;
    Edge(int v = -1, int w = -1) : v(v), w(w) { }
  };
class GRAPH
  { private:
        // Код, зависящий от реализации
    public:
        GRAPH(int, bool);
        ~GRAPH();
        int V() const;
        int E() const;
        bool directed() const;
        int insert(Edge);
        int remove(Edge);
        bool edge(int, int);
        class adjIterator
          {
            public:
              adjIterator(const GRAPH &, int);
              int beg();
              int nxt();
              bool end();
          };
};
```

АТД в программе 17.1 представляет собой основной механизм, который позволяет нам разрабатывать и осуществлять тестирование алгоритмов; это отнюдь не универсальный интерфейс. Как обычно, мы работаем с простейшим интерфейсом, который поддерживает базовые операции обработки графов, исследование которых мы намерены провести. Определение такого интерфейса для использования в практических приложениях требует достижения многочисленных компромиссов между простотой, эффективностью и универсальностью. Далее мы рассмотрим лишь некоторые из таких компромиссов; остальные мы будем анализировать в контексте реализаций и приложений, которые будут встречаться на протяжении всей этой книги.

Конструктор графа принимает максимально возможное число вершин графа в качестве аргумента с тем, чтобы реализации могли выделять соответствующее пространство памяти. Мы принимаем это соглашение единственно с целью сделать программный код по возможности компактным и удобным для чтения. Более универсальный граф может включать в свой интерфейс возможность добавления и удаления как вершин, так и ребер; это обстоятельство налагает более строгие требования на структуры данных, используемые для реализации АТД. Мы можем остановить свой выбор на некотором промежуточном уровне абстракции и учитывать возможность разработки интерфейсов, поддерживающих высокоуровневые абстрактные операции для работы с графами, которые можно использовать в реализациях. Мы еще раз коротко проанализируем эту идею в разделе 17.5 после того, как рассмотрим несколько конкретных представлений и реализаций.

В АТД графа общего вида следует учитывать параллельные ребра и петли, поскольку ничто не мешает клиентской программе вызвать функцию **insert** с использованием в качестве параметра ребра, которое уже существует (параллельное ребро), или ребра с одинаковыми индексами вершин, его образующих. Возможно, что в некоторых приложениях придется запретить использование таких ребер, зато в других приложениях их включение желательно, а в некоторых приложениях они попросту игнорируются. Манипулирование петлями тривиально, в то время как поддержка параллельных ребер требует существенных затрат ресурсов в зависимости от представления графа. В некоторых ситуациях целесообразно включить функцию АТД *удалить параллельные ребра* (*remove parallel edges*). Далее, реализации могут разрешить объединение параллельных ребер, а клиентские программы могут их удалять или выполнять над параллельными ребрами другие операции, если получат на то соответствующие полномочия. Мы вернемся к рассмотрению этих проблем в разделах 17.4 и 17.5.

Программа 17.2 представляет собой функцию, которая служит иллюстрацией использования класса итератора в АТД графа. Эта функция извлекает из графа некоторое множество ребер и передает их функции **vector** из библиотеки STL (Standard Template Library — стандартная библиотека шаблонов) C++, которая выполняет построение вектора. Сам граф есть ни что иное, как множество ребер, и нам довольно часто нужно получить граф именно в такой форме, независимо от его внутреннего представления. Порядок, в котором ребра располагаются в векторе, не играет роли и меняется от приложения к приложению. Мы используем шаблон этих функций с тем, чтобы обеспечить возможность использования различных реализаций АДТ графа.

Программа 17.2. Пример клиентской функции обработки графов

Эта функция осуществляет один из способов использования АТД графа с целью реализации базовой операции обработки графов, в некотором смысле независимой от представления. Она возвращает все ребра графа в виде вектора.

Эта реализация служит иллюстрацией основы большинства программ, которые мы будем рассматривать: мы производим обработку каждого ребра графа путем проверки всех вершин, смежных с каждой конкретной вершиной. В общем случае мы не вызываем функции **beg**, **end** и **nxt** никаким другим способом кроме того, который используется в этой программе, и это обстоятельство позволяет лучше оценить рабочие характеристики нашей реализации (см. раздел 17.5).

```
template <class Graph>
vector <Edge> edges(Graph &G)
  { int E = 0;
    vector <Edge> a(G.E());
    for (int v = 0; v < G.V(); v++)
      {
        typename Graph::adjIterator A(G, v);
        for (int w = A.beg(); !A.end(); w = A.nxt())
          if (G.directed() || v < w)
            a[E++] = Edge(v, w);
      }
    return a;
}
```

```
 0:  1 2 5 6
 1:  0
 2:  0
 3:  4 5
 4:  3 5 6
 5:  0 3 4
 6:  0 4
 7:  8
 8:  7
 9:  10 11 12
10:  9
11:  9 12
12:  9 11
```

РИСУНОК 17.7. ФОРМАТ СПИСКА СМЕЖНЫХ ВЕРШИН

*Эта таблица служит еще одним способом представления графа, приведенного на рис. 17.1; мы связываем каждую вершину с множеством смежных с ней вершин (которые соединены с ней посредством одного ребра). Каждое ребро фигурирует в двух множествах: для каждого ребра **u-v** графа вершина **u** содержится в множестве, соответствующем вершине **v**, а вершина **v** содержится в множестве, соответствующем вершине **u**.*

Программа 17.3 представляет собой другой пример использования класса итератора в АТД графов для распечатки таблиц вершин, смежных с каждой конкретной вершиной, как показано на рис. 17.7. Программные коды этих двух примеров во многом подобны, как они подобны программным реализациям многочисленных алгоритмов обработки графов. Следует отметить тот замечательный факт, что мы можем построить все алгоритмы, изучаемые в данной книге, на основе абстракции обработки всех вершин, смежных с каждой конкретной вершиной (что эквивалентно обработке всех ребер в графе), как и в указанных выше функциях.

Программа 17.3. Клиентская функция печати графа

Данная реализация функции **show** класса **io** программы 17.4 использует АТД графа для печати таблицы вершин, смежных с каждой вершиной графа. Порядок, в котором вершины появляются в таблице, зависит от представления графа и реализации АТД (см. рис. 17.7).

```
template <class Graph>
void IO<Graph>::show(const Graph &G)
  {
    for (int s = 0; s < G.V(); s++)
      {
        cout.width(2); cout << s << ":";
        typename Graph::adjIterator A(G, s);
        for (int t = A.beg(); !A.end(); t = A.nxt())
          { cout.width(2); cout << t << " "; }
        cout << endl;
      }
}
```

Как следует из рассуждений, проведенных в разделе 17.5, мы часто упаковываем логически связанные функции в отдельный класс. Программа 17.4 представляет собой интерфейс такого класса. В ней дается определение функции **show**, используемой в программе 17.3, а также двух других функций, которые вставляют в граф ребра, полученные из стандартного ввода (для ознакомления с реализациями этих функций обратитесь к упражнению 17.12 и программе 17.14).

В общем случае задачи обработки графов, которые исследуются в этой книге, подразделяются на три обширных категории:

- Вычисления значений некоторых размеров графа.
- Выбор некоторого подмножества ребер графа.
- Ответы на вопросы, касающиеся некоторых свойств графа.

Примеры, относящиеся к первой категории, суть количество связных компонент и длина кратчайшего пути между двумя заданными вершинами графа; примеры, относящиеся ко второй категории, связаны с остовным деревом и циклом наибольшей длины, который содержит заданную вершину; примеры третьей категории составляют вопросы наподобие: находятся ли заданные вершины в одной и той же компоненте. В самом деле, условия, которые мы определили в разделе 17.1, немедленно выводят нас на множество вычислительных задач.

Программа 17.4. Интерфейс ввода/вывода для функций обработки графов

Этот класс служит иллюстрацией способа упаковки родственных функций в единый класс. Он определяет функции, выполняющие печать графа (см. программу 17.3), вставку ребер, вводимых в виде пар целых чисел через стандартный ввод (см. упражнение 17.12) и вставку ребер, вводимых в виде пар символов через стандартный ввод (см. программу 17.14).

```
template <class Graph>
class IO
  {
    public:
      static void show(const Graph &);
      static void scanEZ(Graph &);
      static void scan(Graph &);
  };
```

Наш подход к решению такого рода задач предполагает построение абстрактных типов данных (АТД), которые являются клиентами базовых АТД из программы 17.1, однако это, в свою очередь, позволяет определить клиентские программы, требуемые для решения реальных задач. Например, программа 17.5 есть интерфейс для АТД связности графа. Мы можем написать клиентские программы, использующие этот АТД для построения объектов, которые вычисляют количество связных компонентов в графе и которые могут проверить, находятся ли любые две вершины в одной и той же связной компоненте. Описание реализаций этого АТД и их рабочие характеристики мы даем в разделе 18.5; мы разрабатываем АТД подобного рода на протяжении всей книги. Обычно такие АТД содержат общедоступную функцию-элемент, выполняющую *предварительную обработку (preprocessing)* (обычно это конструктор), приватные элементы данных, которые содержат информацию, полученную во время предварительной обработки, и общедоступные функции-элементы обслуживания *запроса (query)*, которые используют эту информацию для предоставления клиентам информации о графе.

В этой книге мы работаем главным образом со *статическими* (*static*) графами, которые содержат фиксированное число вершин V и ребер E. В общем случае мы выполняем построение графов, осуществляя E вызовов функции **insert** с последующим их обработкой либо в результате вызова соответствующей функции АТД, которая принимает граф

в качестве аргумента и возвращает некоторую информацию, касающуюся графа, либо за счет использования объектов указанного выше вида, которые служат для предварительной обработки графа с целью обеспечить возможность эффективного ответа на запросы, касающиеся графа. В любом случае, изменения графа путем обращения к функциям **insert** и **remove** обусловливает необходимость повторной обработки графа. Динамические задачи, в рамках которых мы хотим совместить обработку графов с добавлением или удалением вершин и ребер графа, приводят нас в область *интерактивных алгоритмов* (*on-line algorithm*), известных также как *динамические алгоритмы* (*dynamic algorithms*), в которой приходится сталкиваться с другими сложными проблемами. Например, проблема связности, которую мы решали с помощью алгоритма объединения-поиска в главе 1, представляет собой пример интерактивного алгоритма, поскольку мы можем получить информацию о связности графа во время включения ребер в этот граф. АТД из программы 17.1 поддерживает операции *вставить ребро* (*insert edge*) и *удалить ребро* (*remove edge*), благодаря чему клиенты могут использовать их для внесения изменений в графы, однако выполнение некоторых последовательностей операций сопряжено со снижением производительности. Например, алгоритмы объединения-поиска могут потребовать повторной обработки всего графа, если клиент использует операцию *удалить ребро*. Большая часть проблем, связанных с обработкой графов, с которыми мы сталкиваемся при добавлении или удалении нескольких ребер, может кардинально изменить природу графа, чем и объясняется необходимость его повторной обработки.

Программа 17.5. Интерфейс связности

Данный интерфейс АТД служит иллюстрацией типичной парадигмы, которую мы используем для реализации алгоритмов обработки графов. Он предоставляет клиенту возможность строить объекты, которые выполняют обработку графа таким образом, чтобы отвечать на запросы, касающиеся связности этого графа. Функция-элемент **count** возвращает число связных компонент графа, а функция-элемент **connect** проверяет, имеется ли связь между двумя заданными вершинами. Программа 18.4 представляет собой реализацию этого интерфейса.

```
template <class Graph>
class CC
  {
   private:
        // Код, зависящий от реализации
   public:
        CC(const Graph &);
        int count();
        bool connect(int, int);
  };
```

Одна из наиболее важных и сложных проблем, которая встает перед нами при обработке графа, заключается в том, чтобы получить четкое понимание рабочих характеристик реализаций и убедиться в том, что клиентские программы правильно ими пользуются. Как и в случае более простых задач, которые рассматривались в частях 1–4, выбранная методика использования абстрактных типов данных позволяет решать эти проблемы в логической последовательности.

Программа 17.6. Пример клиентской программы обработки графов

Эта программа служит иллюстрацией использования абстрактных типов данных (АТД), описанных в настоящем разделе и использующих соглашения об АТД, сформулированные в разделе 4.5. Она выполняет построение графа с V вершинами, вставляет в него ребра, получаемые из стандартного ввода, выполняет печать полученного графа, если его размеры позволяют сделать это, и вычисляет (и выводит на печать) количество связных компонент. Предполагается, что программы 17.1, 17.4 и 17.5 (с реализациями) содержатся, соответственно, в файлах **GRAPH.cc**, **IO.cc** и **CC.cc**.

```
#include  <iostream.h>
#include  <stdlib.h>
#include  "GRAPH.cc"
#include  "IO.cc"
#include  "CC.cc"
main(int  argc,  char  *argv[])
{  int  V  =  atoi(argv[1]);
   GRAPH  G(V);
   IO<GRAPH>::scan(G);
   if  (V  <  20)  IO<GRAPH>::show(G);
   cout  <<  G.E()  <<  "  edges  ";
   CC<GRAPH>  Gcc(G);
   cout  <<  Gcc.count()  <<  "  components"  <<  endl;
}
```

Программа 17.6 служит примером клиентской программы обработки графов. Она использует базовый АТД программы 17.1, класс ввода/вывода программы 17.4 для считывания графа из стандартного ввода и его печати на стандартном устройстве вывода, а также класс связности программы 17.5 для отыскания количества его связных компонент. Мы используем подобные, но в то же время более сложные клиентские программы для построения других типов графов, для тестирования алгоритмов, для изучения других свойств графов и для использования графов при решении других проблем. Эту базовую схему можно применять в любых приложениях, выполняющих обработку графов.

В разделах 17.3–17.5 исследуются основные классические представления графа и реализации функций АТД из программы 17.1. Для нас эти реализации представляют собой основу для расширения интерфейсов с тем, чтобы охватить задачи обработки графов, которые станут объектами нашего внимания на протяжении ряда последующих глав.

Первое решение, которое следует принять при разработке реализации АТД, касается того, каким представлением графа необходимо воспользоваться. Во-первых, мы должны уметь приноравливаться ко всем типам графов, с которыми нам могут встретиться в приложениях (мы, скорее всего, не захотим впустую растрачивать пространство памяти). Во-вторых, мы должны уметь эффективно строить требуемые структуры данных. В третьих, мы хотим построить эффективные алгоритмы решения задач, связанных с обработкой графов, и не быть связанными ограничениями, накладываемыми представлением графа. Эти требования остаются в силе для любой рассматриваемой нами предметной области; здесь мы еще раз обращаем на них внимание, поскольку, как будет показано далее, различные представления являются причиной возникновения больших различий даже в случае самых простых задач.

Например, мы можем рассматривать представление *вектора ребер* (*vector of edges*) в качестве базы для реализации АТД (см. упражнение 17.16). Это прямое представление яв-

ляется простым, однако оно не позволяет эффективно выполнять базовые операции обработки графов, к изучению которых мы вскоре приступим. У нас будет возможность убедиться в том, что большинство приложений, выполняющих обработку графов, можно применять с большим или меньшим успехом, если пользоваться всего лишь двумя несколько более сложными по сравнению с вектором ребер представлениями, а именно, в виде *матриц смежности* (*adjacency-matrix*) и в виде *списков смежных вершин* (*adjacency-list*) графа. Эти представления графов, которые будут рассматриваться в разделах 17.3 и 17.4, основаны на элементарных структурах данных (в самом деле, мы их обсуждали как в главе 3, так и в главе 5, как примеры применения последовательного и связного распределения). Выбор какого-либо из них зависит главным образом от того, является ли граф насыщенным или разреженным, хотя, как обычно, характер выполняемых операций также играет важную роль при принятии решения в пользу того или другого представления.

Упражнения

▷ **17.12**. Разработайте реализацию функции **scanEZ** из программы 17.4: напишите функцию, которая строит граф путем считывания ребер (пар целых чисел в диапазоне от 0 до $V-1$) со стандартного устройства ввода.

▷ **17.13**. Составьте программу клиентского АДТ, которая добавляет ребра из заданного вектора в заданный граф.

▷ **17.14**. Напишите функцию, которая вызывает функцию **edges** и распечатывает все ребра графа в формате, используемом в данном тексте (цифры, обозначающие вершины, разделенные дефисом).

○ **17.15**. Разработайте реализацию для АДТ связности из программы 17.5, используя алгоритм объединения-поиска (см. главу 1).

● **17.8**. Постройте реализацию функций из программы 17.1, которая использует вектор ребер для представления графа. Используйте метод реализации "в лоб" функции, которая удаляет ребро **v-w** путем просмотра указанного вектора с целью отыскания ребра **v-w** или **w-v** и последующей замены найденного ребра на последнее ребро вектора. Воспользуйтесь подобным просмотром для реализации итератора. *Примечание*: Предварительное ознакомление с содержимым раздела 17.3 упростит вашу задачу.

17.3. Представление графа в виде матрицы смежности

Представление *матрицы смежности* (*adjacency-matrix*) графа есть матрица булевских значений размерности V на V, элемент которой, стоящий на пересечении v-й строки и w-го столбца принимает значение 1, если в графе имеется ребро, соединяющее вершину v с вершиной w, и 0 в противном случае. На рис. 17.8 показан соответствующий пример.

Программа 17.7 является реализацией интерфейса АДТ графа, которая использует прямое представление этой матрицы, построенной как вектор векторов в соответствии с рис. 17.9. Это двухмерная таблица существования, элемент **adj[v][w]** которой принимает значение **true**, если в графе существует ребро, соединяющее вершину **v** с вершиной **w**, и значение **false** в противном случае. Обратите внимание на то обстоятельство, что сохранение этого свойства в неориентированном графе требуется, чтобы каждое ребро было представ-

лено двумя вхождениями: ребро **v-w** представлено значением **true** как для **adj[v][w]**, так и для **adj[w][v]**, что соответствует ребру **w-v**.

Имя **DenseGRAPH** в программе 17.7 подчеркивает тот факт, что рассматриваемая реализация больше подходит для насыщенных графов, чем для разреженных, и отличает ее от других реализаций. Клиентская программа может воспользоваться определением типа **typedef**, чтобы сделать этот тип эквивалентным типу **GRAPH** или же явно воспользоваться именем **DenseGRAPH**.

В матрице смежности, которая представляет граф **G**, строка **v** есть вектор, представляющий собой таблицу существования, **i**-е вхождение которого равно **true**, если **i** есть вершина, смежная с **v** (ребро **v-i** содержится в графе **G**). Следовательно, чтобы обеспечить клиентам возможность обрабатывать вершины, смежные с **v**, необходима программа, которая просматривает этот вектор с целью нахождения значений **true**, как это делает программа 17.8. Мы должны всегда иметь в виду, что при использовании такая реализация обработки всех вершин, смежных с заданной вершиной, требует (по меньшей мере) время, пропорциональное числу вершин V графа, независимо от того, сколько таких вершин существует.

Как уже говорилось в разделе 17.2, наш интерфейс требует, чтобы в момент инициализации графа клиенту было известно число вершин. При необходимости мы могли бы обеспечить возможность добавления и удаления вершин (см. упражнение 17.21). Основное свойство конструктора в программе 17.7 заключается в том, что он выполняет инициализацию графа, устанавливая значения **false** всем элементам матрицы. Мы должны иметь в виду, что эта операция требует для своего выполнения время, пропорциональное V^2, независимо от того, сколько ребер в графе. Из соображений краткости изложения в программу 17.7 не включена проверка на возникновение ошибок по причине нехватки памяти – обычная практика предполагает включение подобного рода проверок непосредственно перед тем, как такая программа будет использована (см. упражнение 17.24).

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	1	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

РИСУНОК 17.8 ПРЕДСТАВЛЕНИЕ МАТРИЦЫ СМЕЖНОСТИ ГРАФА

Рассматриваемая булева матрица является еще одним представлением графа, показанного на рис. 17.1. В строке v *и столбце* w *(**true**) этой матрицы стоит 1 (**true**), если в графе имеется ребро, соединяющее вершину* v *с вершиной* w*, и 0 (**false**), если такое ребро отсутствует. Эта матрица симметрична относительно главной диагонали. Например, шестая строка (и шестой столбец) показывает, что вершина **6** соединена с вершинами **0** и **4**. Для некоторых приложений мы принимаем соглашение, по условиям которого каждая вершина соединена сама с собой, и ставим единицы на главной диагонали. Большие области нулей в верхнем правом углу и нижнем левом углу матрицы обусловлены выбранным нами способом обозначения вершин для данного конкретного примера и не являются характеристиками рассматриваемого графика (за исключением того факта, что они указывают на принадлежность этого графа к категории разреженных).*

Программа 17.7. Реализация АТД графа

Данный класс представляет собой прямую реализацию интерфейса из программы 17.1, основанную на представлении графа в виде вектора булевых векторов (см. рис. 17.9). Включение и удаление ребер выполняется за постоянное время. Дубли запросов на

включение ребра (функция **insert**) игнорируются "без лишнего шума", однако клиенты могут пользоваться функцией **edge** для проверки, существует ли то или иное ребро. Для построения графа требуется время, пропорциональное V^2.

```
class  DenseGRAPH
{ int Vcnt, Ecnt; bool digraph;
  vector <vector <bool> > adj;
public:
  DenseGRAPH(int V, bool digraph = false) :
    adj(V), Vcnt(V), Ecnt(0), digraph(digraph)
    {
      for (int i = 0; i < V; i++)
        adj[i].assign(V, false);
    }
  int V() const { return Vcnt; }
  int E() const { return Ecnt; }
  bool directed() const { return digraph; }
  void insert(Edge e)
    { int v = e.v, w = e.w;
      if (adj[v][w] == false) Ecnt++;
      adj[v][w] = true;
      if (!digraph) adj[w][v] = true;
    }
  void remove(Edge e)
    { int v = e.v, w = e.w;
      if (adj[v][w] == true) Ecnt--;
      adj[v][w] = false;
      if (!digraph) adj[w][v] = false;
    }
  bool edge(int v, int w) const
    { return adj[v][w]; }
  class adjIterator;
  friend class adjIterator;
};
```

Чтобы добавить в граф ребро, мы присваиваем элементам указанной матрицы значение **false** (одно для орграфов, два для неориентированных графов). Такое представление не допускает параллельных ребер: если в граф нужно вставить ребро, для которого соответствующие вхождения матрицы уже получили значение 1, то программа не дает результата. В некоторых проектах АТД может оказаться целесообразным информировать клиента о попытке включить параллельное ребро, возможно, код возврата операции **insert**. Это представление допускает использование петель: ребро **v-v** представляется ненулевым значением элемента **a[v][v]**.

Программа 17.8. Итератор для представления матрицы смежности

Данная реализация итератора для программы 17.7 использует индекс *i* для просмотра значений строки *v* матрицы смежности (**adj[v]**), пропуская элементы, имеющие значение **false**. Вызов функции **beg()**, за которым следует последовательность вызовов функций **nxt()** (проверка того, что значением функции **end()** является **false** перед каждым таким вызовом), позволяет получить последовательность вершин, смежных с вершиной **v** графа **G** в порядке возрастания индексов вершин.

```
class  DenseGRAPH::adjIterator
{ const DenseGRAPH &G;
  int i, v;
```

```
public:
   adjIterator(const DenseGRAPH &G, int v) :
    G(G), v(v), i(-1) { }
   int beg()
    { i = -1; return nxt(); }
   int nxt()
    {
       for (i++; i < G.V(); i++)
        if (G.adj[v][i] == true) return i;
       return -1;
    }
   bool end()
    { return i >= G.V(); }
};
```

Чтобы удалить ребро, мы присваиваем указанным элементам матрицы значение **false**. При попытке удалить несуществующее ребро (одно из тех, для которых значения соответствующих элементов матрицы есть **false**), программа не дает результата. Опять-таки, в некоторых проектах АТД мы можем посчитать целесообразным уведомлять клиентские программы о возникновении такого рода условий.

Если мы выполняем разработку крупных графов или большое количество небольших по размерам графов, либо в тех случаях, когда по тем или иным причинам ощущается нехватка памяти, существует несколько способов экономии памяти. Например, матрицы смежности, которые служат представлением неориентированного графа, симметричны: элемент **a[v][w]** всегда равен элементу **a [w][v]**. Следовательно, мы можем сэкономить память, сохранив только половину симметричной матрицы (см. упражнение 17.22). Другой способ экономии значительного пространства памяти заключается в использовании матрицы битов (предполагая, что функция **vector<bool>** этого не делает). Таким образом, например, мы можем получить представление графа, состоящего примерно из 64000 вершин в примерно 64 миллионах 64-битовых слов (см. упражнение 17.23). Эти реализации связаны с небольшими осложнениями, поскольку нам необходимо добавить операцию по проверке существования ребра (см. упражнение 17.20). (Мы не используем такой операции в наших реализациях, поскольку можем проверить, существует ли ребро **v-w** путем проверки значения **a[v][w]**). Подобные методы экономии пространства памяти эффективны, но их осуществление связано с дополнительными непроизводительными затратами ресурсов, которые могут пагубно сказаться на внутреннем цикле приложения, критического по времени выполнения.

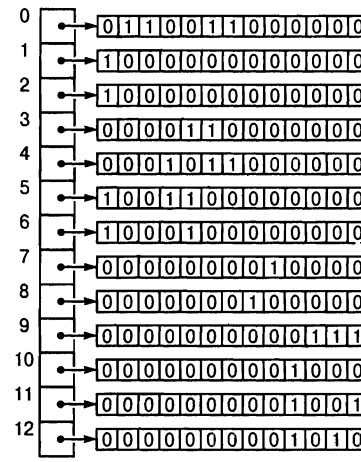


РИСУНОК 17.9. СТРУКТУРА ДАННЫХ МАТРИЦЫ СМЕЖНОСТИ

На этом рисунке изображено C++-представление графа, показанного на рис. 17.1, в виде вектора векторов.

Многие приложения привязывают к каждому ребру различную информацию – в таких случаях мы можем придать матрице смежности более универсальный характер, позволив ей хранить любую информацию, а не только булевские значения. Какой бы тип данных мы не использовали для представления элементов матрицы,

все равно необходимо включить признаки, указывающие, существует ли соответствующее ребро или нет. В главах 20 и 21 мы проведем исследование таких представлений.

Использование матриц смежности зависит от назначения в качестве имен вершин целых чисел в диапазоне от 0 до $V-1$. Подобного рода назначения можно выполнять множеством различных способов; в качестве примера, в разделе 17.6 мы рассмотрим программу, которая выполняет эту процедуру. Таким образом, конкретная матрица значений 0-1, которую мы представили в виде вектора векторов в языке C++, не является единственным возможным представлением заданного графа в виде матрицы смежности, ибо другая программа может присвоить другие имена вершин индексам, которые мы используем для обозначения строк и столбцов. Две матрицы, которые на первый взгляд существенно отличаются друг от друга, на самом деле могут представлять один и тот же граф (см. упражнение 17.17). Это замечание может быть использовано как одна из формулировок проблемы изоморфизма графа: несмотря на то, что мы хотели бы знать, являются ли две различные матрицы представлением одного и того же графа, еще никто не изобрел алгоритма, который всегда бы мог эффективно решать эту задачу. Эта трудность носит фундаментальный характер. Например, наши возможности найти эффективное решение различных важных проблем обработки графов полностью зависят от способа нумерации вершин (смотрите, например, упражнение 17.26).

Программа 17.3, которую мы рассматривали в разделе 17.2, распечатывает таблицу вершин, смежных с каждой конкретной вершиной. Когда она используется совместно с реализацией в программе 17.7, она распечатывает список вершин в порядке возрастания их индекса, как это сделано на рис. 17.7. Обратите, однако, внимание на тот факт, что она не является составной частью определения класса **adjIterator**, что она совершает обход вершин в порядке возрастания индексов, посему разработка клиента АТД, который производит распечатку представления графа в виде матрицы смежности – отнюдь не тривиальная задача (см. упражнение 17.18). Выходные данные, порождаемые этими программами, сами являются представлениями графа, служащими наглядной иллюстрацией выводов, к которым мы пришли, взяв в качестве основного критерия производительность алгоритма. Для печати такой матрицы потребуется пространство на странице, достаточное для размещения всех V^2 элементов; чтобы распечатать списки, нужно пространство, достаточное для размещения $V+E$ чисел. В случае разреженных графов, когда V^2 огромно по сравнению с $V+E$, мы предпочитаем воспользоваться списками, а в случае насыщенного графа, когда E и V^2 сравнимы, — матрицей. Нам вскоре представится возможность убедиться в том, что мы придем к тем же выводам, когда будем сравнивать представление графа в виде матрицы смежности и его основной альтернативой — явным представлением графа в виде списков.

Представление графа в виде матрицы смежности не особенно подходит для разреженных графов: нам необходимо V^2 битов памяти и выполнить V^2 действий, чтобы построить такое представление. В насыщенном графе, в котором число ребер (число единичных битов в матрице) пропорционально V^2, такая цена может быть приемлемой, поскольку для обработки ребер понадобится время, пропорциональное V^2, независимо от того, какое представление используется. В случае разреженного графа, однако, именно инициализация матрицы может оказаться доминирующей составляющей времени выполнения алгоритма. Более того, может случиться так, что нам не хватит пространства для разме-

щения матрицы в памяти. Например, мы можем столкнуться с графом, содержащим миллионы вершин, и не захотим, а может быть, и не сможем, заплатить за это цену, выражающуюся в триллионах вхождений 0 в матрицу смежности.

С другой стороны, когда возникает необходимость обработки графа огромных размеров, 0-вхождения, представляющие отсутствующие ребра, вызывают увеличение наших потребностей в памяти, выражаемое некоторым постоянным множителем, и предоставляет возможность определить, существует ли некоторое конкретное ребро за постоянное время. Например, использование параллельных ребер автоматически запрещается в матрицах смежности, но оно требует значительных затрат в других представлениях графа. Если в нашем распоряжении имеется достаточно пространства памяти, чтобы разместить матрицу смежности, либо если значение V^2 настолько мало, что и время обработки графа пренебрежимо мало, либо мы выполняем сложный алгоритм, требующий для своего завершения более, чем V^2 действий, то представление графа в виде матрицы смежности может оказаться оптимальным выбором, вне зависимости от того, насколько насыщенным является граф.

Упражнения

▷ **17.17**. Постройте представления трех графов, изображенных на рис. 17.2, в виде матриц смежности.

○ **17.18**. Постройте реализацию функции **show** для независимого от представления графа пакета **io** из программы 17.4, которая выполняет распечатку двухмерную матрицу нулей и единиц, подобную той, что приводится на рис. 17.8. *Примечание*: Вы не должны зависеть от итератора, который порождает вершины в порядке следования их индексов.

17.19. Пусть задан некоторый граф; исследуйте другой граф, идентичный первому за исключением того, что имена (соответствующие целочисленные значения) вершин заменены местами. Как матрицы смежности этих двух графов соотносятся друг с другом?

▷ **17.20**. Добавьте функцию **edge** в АТД графа, которая позволит клиентам проверять, существует ли ребро, соединяющие две заданные вершины, и постройте реализацию, ориентированную на представление графа в виде матрицы смежности.

○ **17.21**. Добавьте в АТД графа функции, которые предоставляют клиентам возможность включать и удалять вершины, и постройте реализацию, ориентированную на представление графа в виде матрицы смежности.

▷ **17.22**. Внесите изменения в программу 17.7, расширенную в соответствии с условиями, сформулированными в упражнении 17.20, с целью снижения требований, предъявляемых этой программой к пространству памяти, примерно наполовину, благодаря тому, что в массив не включаются вхождения **a[v][w]**, для которых **w** больше, чем **v**.

17.23. Внесите изменения в программу 17.7, расширенную в соответствии с условиями, сформулированными в упражнении 17.20, с тем, чтобы на компьютере, слово которого состоит из B битов, граф с V вершинами был представлен примерно V^2/B (вместо V^2) словами. Проведите эмпирические испытания с целью оценки влияния, оказываемого подобного рода упаковкой битов в слова, на время выполнения операций АТД.

17.24. Опишите, что произойдет, если в момент вызова конструктора, используемого в программе 17.7, имеет место недостаток памяти для размещения матрицы смежности, и внесите в программные коды необходимые изменения, позволяющие справиться с возникающей при этом ситуацией.

17.25. Разработайте версию программы 17.7, которая использует единственный вектор, содержащий V^2 элементов.

o **17.26**. Предположим, что имеется группа из k вершин, индексы которых представляют собой последовательные целые числа. Как по виду матрицы смежности определить, образует ли эта группа вершин клику? Напишите функцию клиентского АТД, которая за время, пропорциональное V^2, находит максимальную группу вершин с последовательными индексами, которые образуют клику.

17.4. Представление графа в виде списка смежных вершин

Стандартное представление графа, которому обычно отдают предпочтение, когда граф не относится к числу насыщенных, называется представлением в виде *списков смежных вершин* (*adjacency-lists*), в рамках которого мы отслеживаем все вершины, соединенные с каждой вершиной, включенной в связанный список этой вершины. Мы поддерживаем вектор списков, так что достаточно указать вершину, чтобы получить немедленный доступ к ее списку; мы используем связные списки для того, чтобы ввести новое ребро за постоянное время.

Программа 17.9 представляет собой реализацию интерфейса АТД из программы 17.1, основанную на рассматриваемом подходе, а на рис. 17.10 приводится соответствующий пример. Чтобы добавить в это представление графа ребро, соединяющее вершину **v** с **w**, мы добавляем **w** в список смежности вершины **v** и **v** в список смежности вершины **w**. Таким образом, мы можем выполнить ввод новых ребер за постоянное время, однако используемое при этом общее пространство памяти пропорционально числу вершин плюс число ребер (в отличие от пропорциональности квадрату числа вершин, что имеет место в случае представления графа в виде матрицы смежности). Что касается неориентированных графов, то ребра опять фигурируют в двух различных местах, ибо ребро, соединяющее вершину **v** с **w**, представлено как узлы в обоих списках смежных вершин. Оба включения обязательны, в противном случае мы не сможем толково ответить на такие простые вопросы как: "Какие вер-

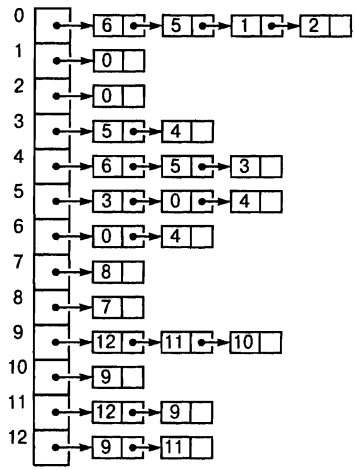


РИСУНОК 17.10. СТРУКТУРА ДАННЫХ СПИСКА СМЕЖНЫХ ВЕРШИН

Данный рисунок отображает представление графа, приведенного на рис. 17.1, в виде массива связных списков. Использованное для этого представления пространство памяти пропорционально сумме числа вершин и числа ребер. Чтобы найти индексы вершин, связанных с заданной вершиной v, *мы просматриваем* v*-ю позицию этого массива, которая содержит указатель на связный список, содержащий один узел для каждой вершины, соединенной с* v. *Порядок, в котором узлы расположены в списках, зависит от метода, использованного для построения этих списков.*

шины являются смежными с вершиной v?" Программа 10.7 реализует итератор, который дает ответ клиентам, задающим подобного рода вопросы, за время, пропорциональное числу таких вершин.

Реализации в программах 17.9 и 17.10 являются низкоуровневыми. В качестве альтернативы можно воспользоваться функцией **list** библиотеки STL, чтобы реализовать каждый связный список (см. упражнение 17.30). Недостаток такого подхода заключается в том, что реализации функции **list** библиотеки STL требуют поддержки гораздо большего числа операций, чем требуется, а это обычно влечет за собой непроизводительные затраты, которые могут неблагоприятно повлиять на все разрабатываемые нами алгоритмы (см. упражнение 17.31). В самом деле, все наши алгоритмы на графах используют интерфейс АТД **Graph**, следовательно, эта реализация служит подходящим местом для инкапсуляции всех низкоуровневых операций, благодаря чему достигается требуемая эффективность и не затрагиваются другие программы. Другое преимущество использования представления графа в виде связного списка заключается в том, что оно предлагает конкретную базу для оценки рабочих характеристик наших приложений.

Еще одним важным фактором, который следует учесть, является тот факт, что реализация в программах 17.9 и 17.10, в основу которой положены связные списки, не полна по причине отсутствия деструктора и конструктора копирования. Для многих приложений это обстоятельство приводит к неожиданным результатам или острым проблемам, обусловленным снижением производительности. Эти функции представляют собой прямое расширение функций, используемых в реализации очереди первого класса в программе 4.22 (см. упражнение 17.29). На протяжении всей книги мы полагаем, что объекты **SparceMultiGRAPH** содержат их. Использование функции **list** из библиотеки STL вместо однонаправленных списков обладает несомненным преимуществом, которое заключается в том, что отпадает необходимость в дополнительном программном коде, поскольку теперь соответствующий деструктор и конструктор копирования определяются автоматически. Например, объекты **DenseGRAPH**, построенные в программе 17.7, должным образом порождаются и уничтожаются клиентскими программами, которые манипулируют ими, так как они построены на базе объектов из библиотеки STL.

Программа 17.9. Реализация АТД графа (списки смежных вершин)

Реализация интерфейса в программе 17.1 использует вектор связных списков, каждый из которых соответствует одной вершине. Она эквивалентна представлению программы 3.15, где ребро **v-w** представлено узлом вершины **w** в списке вершины **v** и узлом вершины **v** в списке вершины **w**.

Реализации функций **remove** и **edge** мы оставляем на самостоятельную проработку, равно как и реализации конструктора копирования и деструктора. Программный код реализации функции **insert** обеспечивает постоянное время включения ребра за счет отказа от проверки наличия дубликатов ребер, общее пространство используемой памяти пропорционально $V + E$; это представление больше всего подходит для разреженных мультиграфов.

Клиентские программы могут воспользоваться функцией **typedef**, чтобы сделать этот тип эквивалентным типу **GRAPH** или непосредственно воспользоваться объектами **SparceMultiGRAPH.**

```
class  SparseMultiGRAPH
{ int Vcnt, Ecnt; bool digraph;
  struct node
   { int v; node* next;
     node(int x, node* t) { v = x; next = t; }
   };
  typedef node* link;
  vector <link> adj;
public:
  SparseMultiGRAPH(int V, bool digraph = false) :
   adj(V), Vcnt(V), Ecnt(0), digraph(digraph)
   { adj.assign(V, 0); }
  int V() const { return Vcnt; }
  int E() const { return Ecnt; }
  bool directed() const { return digraph; }
  void insert(Edge e)
   { int v = e.v, w = e.w;
     adj[v] = new node(w, adj[v]);
     if (!digraph) adj[w] = new node(v, adj[w]);
     Ecnt++;
   }
  void remove(Edge e);
  bool edge(int v, int w) const;
  class adjIterator;
  friend class adjIterator;
};
```

Программа 17.10. Итератор для представления графа в виде списка смежных вершин

Данная итерация итератора для программы 17.9 поддерживает связь **t** при обходе связного списка, ассоциированного с вершиной **v**. Вызов функции **beg()**, за которым следует последовательность вызовов функции **nxt()** (перед каждым таким вызовом производится проверка значения функции **end()**, вызов функции **nxt()** осуществляется в случае, если значением функции **end()** является **false**), позволяет получить последовательность вершин, смежных по отношению к вершине **v** в графе **G**.

```
class  SparseMultiGRAPH::adjIterator
{ const SparseMultiGRAPH &G;
  int v;
  link t;
public:
  adjIterator(const SparseMultiGRAPH &G, int v) :
   G(G), v(v) { t = 0; }
  int beg()
   { t = G.adj[v]; return t ? t->v : -1; }
  int nxt()
   { if (t) t = t->next; return t ? t->v : -1; }
  bool end()
   { return t == 0; }
};
```

В отличие от программы 17.7 программа 17.9 строит мультиграфы благодаря тому, что она не удаляет параллельных ребер. Выявление дубликатов ребер в структуре списка смежных вершин делает необходимым просмотр списков и может потребовать затрат времени, пропорциональных *V*. Аналогично, в программе 17.9 отсутствует реализация операции *удалить ребро (remove edge)* или проверка наличия ребра. Включение реализаций этих

функций не представляет каких-либо трудностей (см. упражнение 17.28), но в то же время на выполнение каждой такой операции может потребоваться время, пропорциональное V, которое затрачивается на поиск в списках узлов, представляющих эти ребра. Подобные затраты делают представление графа в виде списка смежных вершин неприемлемым для приложений, выполняющих обработку особо крупных графов, по условиям которых наличие параллельных ребер исключается, или для приложений, в рамках которых интенсивно используются операции *remove edge (удалить ребро)* или *edge existence (проверка наличия ребра)*. В разделе 17.5 мы будем обсуждать реализации списка смежных вершин, которые обеспечивают выполнение операций *remove edge* и *edge existence* за постоянное время.

Когда в качестве имен вершин графа используются обозначения, отличные от целых чисел, то (как и в случае матриц смежности) две разные программы могут связывать имена вершин с целыми числами в диапазоне от 0 до $V-1$ двумя различными способами, приводящими к образованию двух различных структур списка смежных вершин (см., например, программу 17.15). Мы не можем рассчитывать на то, что сумеем определить, представляют ли различные структуры один и тот же граф, ввиду сложности проблемы изоморфизма графов.

Более того, существует множество представлений графа с заданным числом вершин в виде списков смежных вершин даже при заданной нумерации вершин. Не важно, в каком порядке появляются ребра в списке смежных вершин, структура списка смежных вершин представляет один и тот же граф (см. упражнение 17.33). Полезно знать об этом свойстве списков смежных вершин, поскольку порядок, в котором ребра появляются в списках смежные вершины, влияет, в свою очередь, на порядок, в котором ребра обрабатываются алгоритмами. Это значит, что структура списка смежных вершин определяет, под каким углом зрения используемые нами алгоритмы видят сам граф. И хотя алгоритм должен дать правильный ответ вне зависимости от того, в каком порядке расположены ребра в списке смежных вершин, он должен прийти к правильному ответу в условиях различных последовательностей вычислений, обусловленных различными видами упорядочения ребер. Если алгоритм не обязательно должен проверять все ребра, образующие граф, это обстоятельство может повлиять на продолжительность выполнения данной операции. Кроме того, если имеются несколько правильных решений, то различные упорядочения входных данных могут привести к различным выходным результатам.

Основное преимущество представления графа в виде списка смежных вершин перед представлением в виде матрицы смежности заключается в том, что на его реализацию затрачивается пространство памяти, пропорциональное $V+E$, а не V^2, как в случае представления в виде матрицы смежности. Основной недостаток этого представления состоит в том, что проверка существования конкретных ребер может потребовать времени, пропорционального V, в противовес постоянному времени в случае матрицы смежности. Эти различия приводят, главным образом, к различию в использовании связных списков и векторов для представления множеств вершин, инцидентных каждой вершине.

Таким образом, мы снова убеждаемся в том, что понимание основных свойств связных структур данных и векторов критично для построения реализаций АТД графа. Наш интерес к такому различию в производительности обусловливается тем фактом, что мы хотим избежать реализаций, эффективность которых падает до недопустимого уровня в условиях, когда от АТД требуется выполнение широкого спектра операций. В разделе 17.5

мы будем обсуждать вопросы применения базовых структур данных с целью использования теоретических достоинств обоих рассматриваемых структур на практике. Тем не менее, программа 17.9 является простой реализацией, обладающей основными характеристиками, которые мы должны изучить с тем, чтобы уметь разрабатывать эффективные алгоритмы обработки разреженных графов.

Упражнения

▷ **17.27**. В стиле примера 17.10 покажите структуру списков смежных вершин, построение которой производится при использовании программы 17.9 с целью вставки ребер

3-7, **1-4**, **7-8**, **0-5**, **5-2**, **3-8**, **2-9**, **0-6**, **4-9**, **2-6**, **6-4**

(в указанном порядке) в первоначально пустой граф.

17.28. Предложите свои реализации функций **remove** и **edge** для класса графов, представленных в виде списков смежных вершин (программа 17.9). *Примечание*: Наличие дубликатов ребер допускается, однако при этом вполне достаточно удалить любое ребро, соединяющее заданные вершины.

17.29. Включите конструктор копирования и деструктор в класс графов, представленных в виде списков смежных вершин (программа 17.9). *Указание*: См. программу 4.22.

○ **17.30**. Внесите изменения в реализацию программы 17.9 и 17.10, в которых используется класс **SparceMultiGRAPH**, с целью получить возможность воспользоваться классом **list** из библиотеки STL вместо связного списка для представления каждого списка смежных вершин.

17.31. Примените эмпирические критерии с целью сравнения построенной вами в упражнении 17.30 реализации класса **SparceMultiGRAPH** с реализацией, приведенной в тексте. На примере специально подобранного множества значений V сравните значения времени прогона клиентской программы, которая строит полные графы с V вершинами, после чего удаляет из них ребра с помощью программы 17.2.

○ **17.32**. Приведите простой пример представления графа в виде списков смежных вершин, которые не могли быть построены средствами программы 17.9 путем многократной вставки ребер.

17.33. Сколько различных представлений графа в виде списков смежных вершин, представляют один и тот же граф, показанный на рис. 17.10?

○ **17.34**. Включите объявление общедоступной функции-элемента в АТД графа (программа 17.1), которая удаляет петли и параллельные ребра. Составьте тривиальную реализацию этой функции для класса, построенного на базе матрицы смежности (программа 17.7), и реализацию этой функции для класса, построенного на базе списков смежных вершин (программа 17.9), время выполнения которой пропорционально E, а используемое ею дополнительное пространство памяти пропорционально V.

17.35. Напишите версию программы 17.9, которая не допускает существования параллельных ребер (путем просмотра списка смежных вершин во избежание включения дубликатов при каждой вставке ребра) и петель. Сравните полученную вами реализацию с реализацией, описанной в упражнении 17.34. Какая из них лучше подходит для работы со статическими графами? *Примечание*: Чтобы получить оптимальную реализацию, изучите упражнение 17.49.

17.36. Напишите клиентскую программу АТД графа, которая возвращает результат удаления петель, параллельных ребер и вершин степени 0 (изолированных вершин) из заданного графа. *Примечание*: Время прогона вашей программы должно линейно зависеть от размера представления графа.

• **17.37**. Напишите клиентскую программу АТД графа, которая возвращает для заданного графа результат удаления петель, вырождающихся путей, которые состоят исключительно из вершин степени 2. Точнее, каждая вершина степени 2 в графе без параллельных ребер появляется в некотором пути **u- ... -w**, где **u** и **w** – это вершины степени 2, которые могут быть, а могут и не быть, равными. Замените любой путь ребром **u-w**, а затем удалите все неиспользованные вершины степени 2 подобно тому, как это сделано в упражнении 17.37. *Примечание*: Эта операция может привести к появлению петель и параллельных ребер, но она сохраняет степени вершин, которые не были удалены.

▷ **17.38**. Приведите пример (мульти)графа, который можно построить путем применения преобразования, описанного в упражнении 17.37, к примеру графа, показанному на рис. 17.1.

17.5. Вариации, расширения и затраты

В этом разделе мы опишем несколько способов совершенствования представлений графов, которые мы исследовали в разделах 17.3 и 17.4. Рассматриваемые вопросы можно разделить на три категории. Во-первых, базовые механизмы матрицы смежности и списков смежных вершин графа допускают расширения, которые позволяют получать представления других типов графов. В соответствующих главах мы проведем более подробные исследование расширений подобного рода и рассмотрим соответствующие примеры; в данном разделе мы дадим их краткий обзор. Во-вторых, мы рассмотрим структуры АТД графа, обладающие большим набором свойств, чем структура, выбранная нами в качестве базовой, и реализации, которые используют более развитые структуры данных с целью построения более совершенных реализаций этих структур. В третьих, мы обсудим принимаемый нами общий подход к решению задач обработки графов, который предусматривает построение классов, инкапсулирующих специфику задач и использующих АТД базовых графов.

Реализации, которые были построены в разделах 17.7 и 17.9, выполняют построение орграфов, если вызов конструктора содержит второй аргумент, значением которого является **true**. Как показано на рис. 17.11, каждое ребро входит в представление графа только один раз. Ребро **v-w** в орграфе представлено единицей в элементе матрицы смежности, расположенном на пересечении строки **v** и столбца **w** или вершиной **w** в списке смежных вершин вершины **v** в представлении графа в виде множества списков смежных вершин. Эти представления проще, чем соответствующие представления, которые были выбраны для неориентированных графов, однако асимметрия, характерная для графов этого типа, делает их более сложными комбинаторными объектами, чем неориентированные графы, в чем мы убедимся, ознакомившись с содержимым главы 19. Например, стандартное представление графа в виде списка его смежных вершин не обеспечивает возможности применять прямые методы выявления всех ребер, входящих в заданную вершину орграфа, и упомянутое обстоятельство вынуждает нас искать другие представления графа в случаях, когда требуется поддержка этой операции.

Что касается *взвешенных графов (weighed graphs)* и *сетей (networks)*, то мы наполняем матрицу смежности структурами, содержащими информацию о ребрах (включая данные об их наличии или отсутствии), заменяя ими соответствующие булевские значения; в представлении графа в виде списков смежных вершин мы включаем эту информацию в элементы списков смежных вершин.

Часто возникает необходимость привязывать к вершинам или ребрам графа еще больше информации, чтобы графы могли моделировать более сложные объекты. В качестве одного из возможных вариантов мы можем ассоциировать дополнительную информацию с каждым ребром путем расширения типа **Edge**, употребляемого в программе 17.1, с последующим использованием примеров этого типа в матрицах смежности или в узлах списков в списках смежных вершин. Или, поскольку имена вершин суть целые числа в диапазоне от 0 до $V-1$, мы можем воспользоваться векторами, индексированными именами вершин, чтобы привязать к вершинам дополнительную информацию, возможно, с использованием соответствующих АТД. Мы будем рассматривать абстрактные типы данных (АТД) этого вида в главах 20–22. С другой стороны, мы можем воспользоваться отдельными АТД символьных таблиц для привязки дополнительной информации к каждой вершине и к каждому ребру (см. упражнение 17.48 и программу 17.15).

Чтобы успешно решать различные проблемы обработки графов, мы часто определяем классы, которые содержат дополнительные специализированные структуры данных, связанные с графами. К числу наиболее распространенных структур данных такого рода относятся векторы, индексированные именами вершин, как мы уже могли убедиться в главе 1, в которой мы воспользовались векторами, индексированными именами вершин, для ответа на запросы о связности графов. На протяжении данной книги мы используем векторы, индексированные именами вершин, во многих реализациях.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	1	1	0	0	1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0
4	0	0	0	1	1	0	0	0	0	0	0	0	0
5	0	0	0	1	1	1	0	0	0	0	0	0	0
6	0	0	0	0	1	0	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	1	1	1
10	0	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	1
12	0	0	0	0	0	0	0	0	0	0	0	0	1

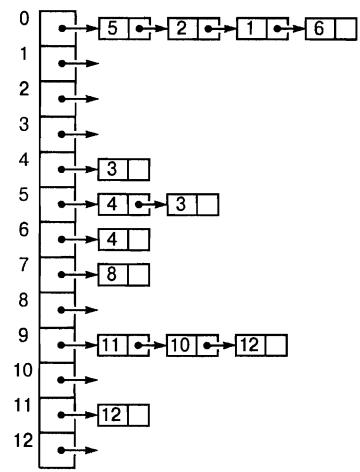


РИСУНОК 17.11. ПРЕДСТАВЛЕНИЯ ОРГРАФА

В представлениях орграфа в виде матрицы смежности и в виде списков смежных вершин каждое ребро представлено только один раз. В этом можно убедиться, ознакомившись с представлениями множества ребер, которые показаны на рис. 17.1 в виде матрицы смежности (вверху) и списков смежных вершин (внизу) и интерпретируемого как орграф (см. рис. 17.6, сверху).

В качестве примера предположим, что мы хотим знать, является ли вершина **v** графа изолированной. Равна ли степень вершины **v** нулю? Что касается представления графа в виде списка смежных вершин, то мы находим эту информацию немедленно, просто проверив, не равно ли значение **adj[v]** нулю. Однако для случая представления графа в виде матрицы смежности мы должны проверить все V элементов, соответствующих строке или

столбцу v, чтобы убедиться, что вершина не соединена ни с какой другой вершиной. В то же время, в случае представления графа в виде векторов ребер у нас нет ничего лучшего, как только проверить все E ребер, чтобы проверить, имеются ли какие-либо ребра, содержащие вершину v. Мы должны предоставить клиентским программам средства, которые позволят им избежать подобного рода вычислений, требующих больших затрат времени. Как уже отмечалось в разделе 17.2, один из способов достижения этой цели заключается в том, чтобы определить АТД клиента, для задачи, подобной рассмотренной в примере для программы 17.11. Такая реализация после выполнения предварительной обработки графа за время, пропорциональное размеру его представления, позволит клиентам определить степень любой вершины за постоянное время. Этот подход нельзя считать усовершенствованием, если клиент стремится определить степень только одной вершины, в то же время он обеспечивает существенную экономию ресурсов тем клиентам, которые хотят знать значения степеней некоторого множества вершин. Существенное различие в производительности алгоритма решения достаточно простой задачи в зависимости от условий его применения весьма характерно для обработки графов.

Программа 17.11. Реализация класса, определяющего степени вершин

Этот класс предлагает клиентским программам способ определения степени любой заданной вершины графа в классе **GRAPH** за постоянное время после предварительной обработки в конструкторе, время которой линейно зависит от размеров графа (линейно зависимая обработка). Реализация основана на поддержке вектора степеней вершин, индексированного именами вершин, как элемента набора данных, и перегрузки **[]** как общедоступной функции-элемента. Мы инициализируем все элементы нулями, а затем выполняем обработку всех ребер графа, увеличивая на единицу значения соответствующих вхождений для каждого ребра.

Мы используем классы, подобные этому, на всем протяжении данной книги при создании объектно-ориентированных реализаций функций обработки графов как клиентов класса **GRAPH**.

```
template <class Graph> class DEGREE
{ const Graph &G;
  vector <int> degree;
public:
  DEGREE(const Graph &G) : G(G), degree(G.V(), 0)
    {
      for (int v = 0; v < G.V(); v++)
      { typename Graph::adjIterator A(G, v);
        for (int w = A.beg(); !A.end(); w = A.nxt())
          degree[v]++;
      }
    }
  int operator[](int v) const
    { return degree[v]; }
};
```

Для каждой задачи обработки графа из числа рассматриваемых в данной книге мы инкапсулируем ее решение в классах, подобных приведенному, с приватными элементами данных и общедоступными функциями-элементами, специфичными для каждой задачи. Клиенты создают объекты, функции-элементы которых реализуют обработку графов. Такой подход получает дальнейшее развитие в расширении интерфейса АДТ графа за

счет определения множества взаимодействующих классов. Любое множество таких классов определяет интерфейс обработки графов, но каждый инкапсулирует собственные приватные данные и функции-элементы.

Существует много других способов разработки интерфейсов на C++. Одно из направлений дальнейших действий состоит в простом включении общедоступных функций-элементов (и любых других приватных элементов данных и функций-элементов, которые, возможно, потребуются) в определение базового АТД класса **GRAPH**. В то время как такой подход обладает всеми достоинствами, которыми мы восхищались в главе 4, ему свойственны также и серьезные недостатки, ибо такая сфера деятельности, как обработка графов, намного шире, чем виды базовых структур данных, которые служили объектом анализа в главе 4. Из всего множества этих недостатков в качестве основных выделим следующие:

- Существует намного больше подлежащих реализации функций обработки графов, чем те, которые мы можем должным образом определить в одном интерфейсе.
- Простые задачи обработки графов должны пользоваться теми же интерфейсами, которые необходимы для решения сложных задач.
- Одна функция-элемент может получить доступ к элементам данных, предназначенных для использования другой функцией-элементом вопреки принципам инкапсуляции, которым мы намерены следовать.

Интерфейсы этого типа получили известность как насыщенные или "*толстые*" *(fat)* интерфейсы. В книге, изобилующей алгоритмами обработки графов, подобного рода интерфейс и на самом деле становится "толстым".

Другой подход предусматривает использование механизма наследования для определения различных типов графов, который предоставляет клиентам различные наборы средств решения задач обработки графов. Сравнение этого замысловатого подхода с более простым подходом, который мы приняли к использованию, – это вполне достойное занятие при изучении проблем программного обеспечения, однако оно еще больше уводит нас от изучения алгоритмов обработки графов, нашей основной цели.

Таблица 17.1 показывает зависимость стоимости различных простых операций обработки графов от выбираемого представления графа. Эту таблицу следует внимательно изучить, прежде чем переходить к реализации более сложных операций; она поможет вам выработать четкое понимание трудности реализации различных простейших операций. Большинство значений стоимости вытекает непосредственно из анализа программных кодов, за исключением последней строки, которая будет подробно рассматриваться в конце данного раздела.

Таблица 17.1. Стоимости выполнения операций обработки графов для худших случаев

Рабочие характеристики основных операций АТД, осуществляющих обработку графов, существенно различаются в зависимости от выбора представления графа, даже если рассматривать только простейшие операции. Этот вывод следует из данных приводимой ниже таблицы стоимостей операций, составленной для худших случаев (эти различия не выходят за пределы постоянного множителя для больших значений V и E). Приведенные стоимости получены для простых реализаций, которые были описаны в предыдущих разделах; различные модификации, которые оказывают влияние на величину стоимостей, описываются в данном разделе.

	Массив ребер	Матрица смежности	Списки смежных вершин
Занимаемое пространство памяти (space)	E	V^2	$V+E$
Инициализировать пустой объект (initialize empty)	1	V^2	V
Копировать (copy)	E	V^2	E
Уничтожить (destroy)	1	V	E
Вставить ребра (insert edge)	1	1	1
Найти/удалить ребро (find/remove)	E	1	V
Вершина **v** изолирована? (is **v** isolated?)	E	V	1
Путь от вершины **u** к вершине **v**? (path from **u** to **v**?)	$E \lg^* V$	V^2	$V+E$

Во многих случаях удается модифицировать представление графа таким образом, что простые операции становятся более эффективными, при этом принимаем меры против того, чтобы стоимость других простых операций не увеличивалась. Например, вхождение в таблице, соответствующее строке *destroy* и столбцу матрицы смежности, есть следствие (артефакт) выбора схемы распределения типа вектор векторов для двухмерных матриц (см. раздел 3.7). Нетрудно уменьшить эти затраты, сделав их постоянными (см. упражнение 17.25). С другой стороны, если ребра графа представляют собой достаточно сложные структуры, для которых элементы матрицы служат указателями, то операция *destroy* над матрицей смежности потребует затрат, пропорциональных V^2.

В силу того, что некоторые операции используются в типовых приложениях особенно часто, подробно рассмотрим операции *find edge* (*найти ребро*) и *remove edge* (*удалить ребро*). В частности, операция *find edge* нужна нам для того, чтобы иметь возможность удалить или запретить включение параллельных ребер. Как следует из раздела 13.4, эти операции тривиальны, если мы пользуемся представлением графа в виде матрицы смежности – нам достаточно всего лишь проверить или установить значение элемента матрицы, который можно индексировать непосредственно. В то же время, как можно обеспечить эффективную реализацию этих операций в условиях представления графа в виде списка смежных вершин? В языке C++ можно воспользоваться библиотекой STL; здесь мы описываем базовые механизмы, чтобы получить представление о проблемах обеспечения эффективности. Один из подходов описан ниже, другой подход упоминается в упражнении 17.50. Оба подхода основаны на использовании реализаций таблицы символов. Например, если мы используем реализации динамической хэш-таблицы (см. раздел 14.5), оба подхода требуют памяти, пропорциональной E, и позволяют выполнять и ту и другую операции за постоянное время (в среднем, с амортизацией расходов на реализацию).

В частности, чтобы реализовать операцию *find edge*, когда мы используем списки смежных вершин, можно воспользоваться вспомогательной таблицей соответствия символов для ребер. Мы можем присвоить ребру **v-w** целочисленный ключ **v*V+w** и задействовать функцию *map* из библиотеки STL или любую реализацию таблицы символов, описание которых приводится в части 4. (Для неориентированных графов можно присваивать одни и те же ключи ребрам **v-w** и **w-v**.) Мы можем вставить каждое ребро в таблицу символов, предварительно проверив, не было ли оно включено раньше. Мы можем остановить свой выбор либо на том, чтобы блокировать включение параллельных ребер (см. упражнение

17.49), либо на ведении дублированных записей в таблице соответствия символов, фиксирующих параллельные ребра (см. упражнение 17.50). В настоящем контексте основной интерес к этому методу вызывается тем, что он делает возможной такую реализацию операции *find edge*, которая обеспечивает ее выполнение за постоянное время для представления графа в виде списка смежных вершин.

Чтобы иметь возможность удалять ребра, в записи таблицы символов для каждого ребра необходим указатель на его представление в структуре списка смежных вершин. Но даже этой информации недостаточно для того, чтобы дать возможность удалять ребра за постоянное время, если только списки не являются дважды связными (см. раздел 3.4). Более того, в случае неориентированных графов нельзя ограничиваться только удалением узла из списка смежных вершин, поскольку каждое ребро содержится в двух различных списках смежных вершин. Один из способов устранения этого затруднения заключается в том, чтобы поместить оба указателя в таблицу соответствия символов; другой предусматривает связывание двух узлов, соответствующих конкретному ребру (см. упражнение 17.46). Какое бы из двух этих решений мы не выбрали, мы получаем возможность удалить ребро за постоянное время.

Удаление вершин требует больших затрат. В представлении графа в виде матрицы смежности мы, по существу, должны удалить из матрицы соответствующие строку и столбец, что требует не меньших затрат, чем построение новой матрицы смежности меньшего размера (хотя в этом случае эти издержки можно уменьшить, если воспользоваться тем же механизмом, который применяется при работе с динамическими хэш-таблицами). Если мы используем представление графа в виде списков смежных вершин, то сразу видим, что недостаточно только удалить узлы из списка смежных вершин той или иной вершины, поскольку каждый узел списка смежных вершин определяет другую вершину, список смежных вершин которой мы обязаны просмотреть, чтобы удалить другой узел, который представляет то же ребро. В соответствии с изложенным в предыдущем параграфе, если мы хотим удалить вершину за время, пропорциональное числу вершин V, необходимо обеспечить поддержку удаления ребра за постоянное время, а для этого потребуются дополнительные связи.

Здесь мы не будем останавливаться на реализации этих операций в силу того, что они представляют собой простые упражнения по программированию, в рамках которых используются базовые технологии, описанные в части 1; в силу того, что библиотека STL содержит реализации, которыми мы можем пользоваться, поскольку поддержка сложных структур со множественными указателями в узле не оправдывается в типовых приложениях обработки статических графов; и в силу того, что мы не хотим утонуть в трясине бесконечных уровней абстракций или в мелких деталях поддержки многочисленных указателей при реализации алгоритмов обработки графов, которые без этого не могут их использовать. В главе 22 мы будем изучать реализации подобных структур, играющих важную роль в мощных универсальных алгоритмах, изучение которых будет продолжаться в данной главе.

Для ясности описания и с целью разработки реализаций, представляющих для нас интерес, мы воспользуемся простейшей из подходящих представлений. Вообще говоря, мы стремимся использовать структуры данных, которые лучше всего подходят для решения имеющихся задач. Многие программисты придерживаются этого вида минимализма как нечто само собой разумеющееся, понимая, что соблюдение принципа целостности струк-

тур данных с многочисленными неравноценными компонентами и в самом деле представляет собой трудную задачу.

Мы можем также рассмотреть и альтернативные реализации, которые выполняют модификацию базовых структур данных в процессе настройки рабочих характеристик с целью экономии пространства памяти или времени выполнения при обработке крупных графов (или большого числа графов небольших размеров). Например, мы можем существенно повысить производительность алгоритмов, выполняющих обработку крупных статических графов, представленных списками смежных вершин, отказавшись от представления множества вершин, инцидентных каждой конкретной вершине, в виде связных списков и воспользовавшись их представлением в виде векторов переменной длины. Пользуясь такой технологией, мы можем в конечном итоге представить граф всего лишь $2E$ целыми числами, что меньше V, и V целыми числами, что меньше V^2 (см. упражнения 17.52 и 17.54). Подобные представления особенно привлекательны в условиях обработки крупных статических графов.

Алгоритмы, которые мы рассматриваем, легко приспосабливаются ко всем изменениям, предложенным нами в этом разделе, поскольку в их основу положены несколько высокоуровневых операций, таких как "*выполнить следующую операцию на каждом ребре, связанном с вершиной v*", которые поддерживаются построенным базовым АТД.

В некоторых случаях наши решения относительно структуры алгоритма зависят от некоторых свойств конкретного представления графа. Работа на высоком уровне может отодвинуть на задний план наше осознание этой зависимости. Если мы знаем, что одно представление ведет к снижению производительности алгоритма, а другое нет, мы берем на себя неоправданный риск, если будем планировать применение алгоритма на неправильно выбранном уровне абстракции. Как обычно, наша цель состоит в том, чтобы построить такие реализации, которые позволяли бы давать точную оценку производительности алгоритма. По этой причине мы сохраняем отдельные типы **DenseGRAPH** (насыщенный граф) и **SparseMultiGRAPH** (разреженный мультиграф) для представления графа, соответственно, в виде матрицы смежности и в виде списка смежных вершин с тем, чтобы клиенты могли воспользоваться той из реализаций, которая лучше подходит для решения стоящей перед ними задачи.

В худшем случае простой алгоритм выбора пути, описанный в разделе 17.7, (а также несколько других методов, которые мы будем изучать в граве 18) исследует все E ребер графа. Вхождения в среднем и правом столбцах нижней строки таблицы 17.1 показывают, соответственно, что этот алгоритм может проверить все V^2 вхождений представления графа в виде матрицы смежности и все V заголовков списков и все E узлов в списках в представлении графа в виде списков смежных вершин. Из этого следует, что время счета алгоритма линейно зависит от размера представления графа, в то же время имеются два исключения из этого правила: в худшем случае линейная зависимость времени счета от числа ребер графа нарушается, если мы используем матрицу смежности для представления разреженного графа или для представления очень разреженного графа (граф, который содержит большое число изолированных вершин). Чтобы больше не останавливаться на этих исключениях, в дальнейшем мы полагаем, что *размер представления графа, которое мы используем, пропорционален числу ребер этого графа.* В большинстве практических приложений эта гипотеза подлежит дальнейшим исследованиям, поскольку они производят

обработку крупных разреженных графов и, следовательно, отдают предпочтение представлению графа в виде списков смежных вершин.

Значение в нижней строке левого столбца таблицы 17.1 получено по результатам применения алгоритмов объединения-поиска, описанного в главе 1 (см. упражнение 17.15). Этот метод привлекателен тем, что требует пространство памяти, всего лишь пропорциональное числу вершин V графа, однако его недостаток заключается в том, что он не способен показывать пути. Это вхождение в таблицу 17.1 подчеркивает важность полного и точного описания задач обработки графов.

Даже после того, как все эти факторы будут учтены, одной из наиболее важных и трудных проблем, с которыми нам приходится сталкиваться при разработке практических алгоритмов обработки графов, является оценка того, насколько рабочие характеристики этих алгоритмов, полученные для худших случаев наподобие представленных в таблице 17.1, переоценивают потребности во времени и в пространстве памяти обработки графов, которые встречаются на практике. Большая часть публикаций по алгоритмам на графах описывает рабочие характеристики в переводе на показатели, которые гарантируются в худшем случае, и в то время как эта информация полезна при выявлении алгоритмов, которые могут обладать заведомо неприемлемыми характеристиками, она может оказаться не в состоянии пролить свет на то, какая из нескольких простых программ является наиболее подходящей для заданного применения. Эта ситуация усугубляется трудностями разработки полезных рабочих моделей алгоритмов на графах для обычных условий работы; все что нам остается (возможно, это ненадежный метод) – это проверка в контрольных точках и (возможно, это чрезмерно консервативный метод) характеристик, гарантированных в худшем случае, на которые мы можем опираться на практике. Например, все методы поиска на графах, которые рассматриваются в главе 18, представляют собой эффективные алгоритмы, подчиняющиеся линейной зависимости и предназначенные для поиска пути между двумя заданными вершинами, однако их рабочие характеристики существенно различаются в зависимости от того, какой граф подвергается обработке и как он представлен. При использовании алгоритмов обработки графов на практике мы ведем постоянную борьбу с таким несоответствием гарантированных рабочих характеристик в худшем случае, которые мы можем доказать, и фактических рабочих характеристик, которые мы имеем основания ожидать. С этой темой мы будем периодически соприкасаться на протяжении всей этой книги.

Упражнения

17.39. Разработайте матрицу смежности для представления насыщенных мультиграфов и построить реализацию АТД для программы 17.1, которая его использует.

○ **17.40**. Почему не используется прямое представление графов (структура данных, которая строит точную модель графа с объектами-вершинами, содержащими списки смежных вершин со ссылками на эти вершины)?

▷ **17.41**. Почему программа 17.11 не увеличивает на единицу *оба* значения **deg[v]** и **deg[w]**, когда она обнаруживает, что вершина **v** смежна с **w**?

▷ **17.42**. Добавьте в класс графа, который использует матрицы смежности (программа 17.7), вектор, индексированный именами вершин, в котором содержатся степени каж-

дой вершины. Добавьте в общедоступную функцию-элемент **degree**, которая возвращает степень заданной вершины.

17.43. Выполните упражнение 17.43 для представления графа в виде списков смежных вершин.

▷ **17.44**. Включите в таблицу 17.1 строку, соответствующую задаче определения числа изолированных вершин графа. Дайте обоснование вашего ответа в виде реализаций функции для каждого из трех представлений, указанных в таблице.

○ **17.45**. Включите в таблицу 17.1 строку, соответствующую задаче, определяющей, содержит ли заданный орграф вершину с полустепенью захода V и полустепенью выхода 0. Дайте обоснование вашего ответа в виде реализаций функции для каждого из трех представлений, указанных в таблице. *Примечание*: Вхождение для представления графа в виде матрицы смежности должно составлять V.

17.46. Воспользуйтесь двухсвязными списками смежных вершин с перекрестными связями, в соответствии с изложенным в тексте, для реализации функции **remove**, выполняющей операцию *remove edge* (*удалить ребро*) за постоянное время на АТД графа, для представления которого используются списки смежных вершин (программа 17.9).

17.47. Добавьте функцию **remove**, выполняющую операцию *remove vertex* (*удалить вершину*) в класс двухсвязного графа, представленного в виде списков связных вершин, в соответствии с изложенным в предыдущем упражнении.

○ **17.48**. В соответствии с изложенным в тексте, внесите изменения в решение задачи 17.16, позволяющие пользоваться динамическими хеш-таблицами, с тем, чтобы операции *insert edge* (*вставить ребро*) и *remove edge* (*удалить ребро*) выполнялись за постоянное время.

17.49. Добавьте в класс графа, использующего для своего представления списки смежных вершин (программа 17.9), таблицу символов, позволяющую игнорировать дубликаты ребер, с тем, чтобы он представлял графы, а не мультиграфы. Воспользуйтесь динамическим хешированием при реализации таблицы символов с тем, чтобы полученные вами реализации занимали пространство памяти, пропорциональное E, и выполняли операции, осуществляющие вставку, поиск и удаление ребра за постоянное время (в среднем, с амортизацией расходов на реализацию).

17.50. Разработайте класс мультиграфа на основе представления мультиграфов в виде векторов таблицы символов (по одной таблице символов на каждую вершину, содержащую список смежных ребер). Воспользуйтесь динамическим хешированием при реализации таблицы символов с тем, чтобы полученная реализация использовала пространство памяти, пропорциональное E, и выполняла операции вставки, поиска и удаления ребра за постоянное время (в среднем, с амортизацией расходов на реализацию).

17.51. Разработайте АТД графа, ориентированный на статические графы, основанные на использовании конструктора, который принимает вектор ребер в качестве аргумента и использует базовый АДТ графов для построения графов. (Подобного рода реализация может оказаться полезной при сравнении по производительности с реализациями, построенными в упражнениях 17.52–17.55.)

17.52. Разработайте реализацию конструктора, описанную в упражнении 17.51, которая использует компактное представление графа, основанное на следующих структурах данных:

```
struct node { int cnt; vector <int> edges; };
struct graph { int V; int E; vector <node> adj; };
```

Граф есть совокупность числа вершин, числа ребер и вектора вершин. Вершина содержит число ребер и вектор с одним индексом вершины, соответствующей каждому смежному ребру.

- **17.53**. Добавьте к вашему решению упражнения 17.52 функцию, которая, аналогично упражнению 17.34, удаляет петли параллельные ребра.

○ **17.54**. Разработайте реализацию АДТ статического графа, описанного в упражнении 17.51, которая использует всего лишь два вектора для представления графа: один вектор – это вектор E вершин, второй – вектор V индексов или указателей на первый вектор. Получите реализацию функции **io::show** для этого представления.

- **17.55**. Добавьте к вашему решению упражнения 17.54 функцию, которая удаляет петли и параллельные ребра, как в упражнении 17.34.

17.56. Разработайте интерфейс АТД графа, который связывает координаты (x,y) с каждой вершиной, что позволит работать с чертежами графов. Включите функции **drawV** и **drawE** для нанесения на чертеж, соответственно, вершин V и ребер E.

17.57. Напишите клиентскую программу, которая использует ваш интерфейс из упражнения 17.56 для вычерчивания ребер, добавляемых в граф небольших размеров.

17.58. Разработайте реализацию интерфейса из упражнения 17.56, создающую программу **PostScript**, выходом которой служат чертежи (см. раздел 4.3.).

17.59. Найдите соответствующий графический интерфейс, который позволил бы разработать реализацию интерфейса из упражнения 17.56, способную непосредственно выводить на экран чертежи графов в специальном окне.

- **17.60**. Распространите ваше решение на упражнения 17.56 и 17.59 с таким расчетом, чтобы они содержали функции удаления вершин и ребер и были способны вычерчивать их в различных стилях с тем, чтобы вы могли писать клиентские программы, способные осуществлять динамические графические анимации алгоритмов обработки графов в процессе работы.

17.6. Генераторы графов

Чтобы углубить наше понимание различных свойств графов как комбинаторных структур, мы теперь рассмотрим подробные примеры типов графов, которыми мы позднее воспользуемся для тестирования изучаемых алгоритмов. Некоторые из этих примеров заимствованы из приложений. Другие взяты из математических моделей, которые предназначены как для исследования свойств, с какими мы можем столкнуться в реальных графах, так и для расширения диапазона входных испытаний, которые можно применять для тестирования избираемых нами алгоритмов.

С целью конкретизации примеров мы представим их в виде клиентских функций программы 17.1 с тем, чтобы их можно было непосредственно применять для тестирования реализаций алгоритмов на графах, которые мы намереваемся использовать на практике. Кроме того, мы проведем исследование реализации функции **io::scan** из программы 17.4, которая производит считывание последовательности пар произвольных имен из стандартного ввода и строит граф, составленный из вершины, соответствующих именам, и ребер, соответствующих парам.

Реализации, которые мы рассматриваем в этом разделе, основаны на интерфейсе, содержащемся в программе 17.1, благодаря чему они функционируют должным образом, по крайней мере, теоретически, при любом представлении графа. На практике, однако, некоторые сочетания реализаций и представлений, как мы вскоре убедимся сами, не могут обеспечить приемлемой производительности.

Программа 17.12. Генератор случайных графов (случайные ребра)

Рассматриваемая функция добавляет в граф произвольное (случайное) ребро путем генерации E случайных пар целых чисел, интерпретации целых чисел как вершин графа и пар меток вершин как ребер графа. Решение о том, как поступать с параллельными ребрами и петлями, возлагается на функцию-элемент **Graph**. В общем случае этот метод не подходит для генерации крупных насыщенных графов из-за того, что при генерации ребер появляется значительное количество параллельных ребер.

```
static void randE(Graph &G, int E)
  {
    for (int i = 0; i < E; i++)
      {
        int v = int(G.V()*rand()/
              (1.0+RAND_MAX));
        int w = int(G.V()*rand()/
              (1.0+RAND_MAX));
        G.insert(Edge(v,w));
      }
  }
```

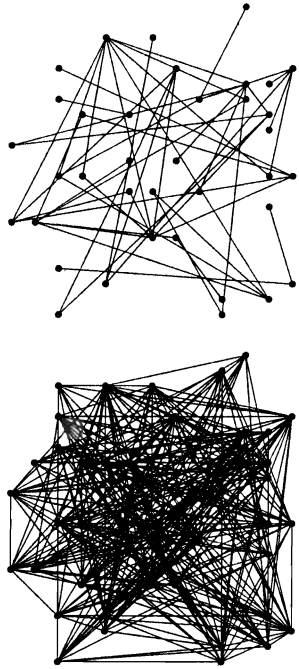


РИСУНОК 17.12. ДВА СЛУЧАЙНЫХ ГРАФА

Оба показанных на рисунке графа содержат по 50 вершин. Разреженный граф в верхней части рисунка содержит 50 ребер, в то время как насыщенный граф в нижней части рисунка — 500 ребер. Разреженный граф не относится к категории связных, поскольку каждая из его вершин соединена только с небольшим числом других вершин; насыщенный граф, несомненно, является связным, ибо каждая его вершина связана в среднем, минимум, с 20 другими вершинами. Эти диаграммы показывают, с какими трудностями сопряжена разработка алгоритмов, которые способны вычерчивать произвольные графы (в рассматриваемом случае вершины размещаются в случайно выбранных местах).

Как обычно, мы заинтересованы в "примерах случайных задач" как в плане выполнения наших задач в условиях произвольных вводов, так и с целью получить представление о том, как будут вести себя программы в реальных приложениях. Что касается графов, то достижение второй цели сопряжено с большими трудностями, чем в других предметных областях, которые мы рассматривали ранее, хотя она все еще оправдывает затрачиваемые усилия. Мы столкнемся с различными моделями случайных величин, начиная со следующих двух.

Случайные ребра. Реализация этой модели довольно проста, в чем легко убедиться, ознакомившись с генератором, представленным программой 17.12. Для заданного числа вершин V мы генерируем произвольные ребра путем выбора случайных чисел в преде-

лах от 0 до $V-1$. Результатом, скорее всего, будет произвольный мультиграф с петлями. Заданной парой может оказаться два идентичных числа (отсюда получаются петли); при этом одна и та же пара может повторяться многократно (отсюда получаются параллельные ребра). Программа 17.12 генерирует ребра до тех пор, пока не станет известно, что число ребер в графе достигло значения E, при этом решение об удалении параллельных ребер остается за реализацией. Если параллельные ребра удалены, то число полученных при генерации ребер должно быть значительно больше, чем число ребер (E), использованных в насыщенных графах (см. упражнение 17.62); в силу отмеченных особенностей этот метод обычно используется для разреженных графов.

Программа 17.13. Генератор случайных графов (случайный граф)

Как и в программе 17.13, рассматриваемая функция генерирует пары случайных целых чисел в диапазоне от 0 до V-1 с тем, чтобы добавлять в граф произвольные ребра, однако она использует другую вероятностную модель, по условиям которой каждое возможное ребро появляется независимо от других с вероятностью p. Значение p вычисляется таким образом, чтобы ожидаемое число ребер ($pV(V$-1)/2) было равно E. Число ребер в каждом конкретном графе, генерируемых этой программой, близко к E, однако маловероятно, что оно будет в точности равно E. Этот метод пригоден в основном применительно к насыщенным графам, поскольку время его выполнения пропорционально V^2.

```
static void randG(Graph &G, int E)
  { double p = 2.0*E/G.V()/(G.V()-1);
    for (int i = 0; i < G.V(); i++)
      for (int j = 0; j < i; j++)
        if (rand() < p*RAND_MAX)
          G.insert(Edge(i, j));
  }
```

Случайный граф. Классическая математическая модель случайных графов должна рассмотреть все возможные ребра и включить в граф каждое ребро с фиксированной вероятностью p. Если мы хотим, чтобы ожидаемое число ребер графа было равно E, мы должны выбрать $p = 2E/V(V-1)$. Программа 17.13 реализует функцию, которая использует эту модель для генерации случайных графов. Эта модель не допускает дубликаты ребер, однако число ребер в графе равно E только *в среднем*. Эта реализация хорошо подходит для насыщенных графов, но отнюдь не для разреженных графов, поскольку за время, пропорциональное $V(V-1)/2$, она генерирует $E = pV(V-1)/2$ ребер. То есть, для разреженных графов время прогона программы 17.13 находится в квадратичной зависимости от размеров графа (см. упражнение 17.68).

Эти модели хорошо изучены, а их реализация не представляет трудностей, однако они не обязательно генерируют графы со свойствами, подобные свойствам графов, наблюдаемым на практике. В частности, графы, которые моделируют карты, электронные схемы, расписания, транзакции, сети и другие реальные ситуации, обычно не только относятся к числу разреженных, но и проявляют свойство *локальности* – вероятность того, что заданная вершина соединена с одной из вершин конкретного множества вершин, выше, чем вероятность соединения с вершиной, не входящей в это множество. Мы можем обсуждать многие различные способы моделирования свойства локальности, как показывают следующие примеры.

***k*-соседний граф**. Граф, изображенный в верхней части рис. 17.13, вычерчен по результатам простых изменений, внесенных в генератор графов со случайными ребрами, в процессе которых мы случайным образом выбирали первую вершину **v**, затем также случайным образом выбирали следующую вершину из числа тех вершин, индексы которых попадали в диапазон, образованный некоторой постоянной k, с центром в **v** (циклический возврат от $V-1$ до 0, когда вершины упорядочены в виде окружности, как показано на рис. 17.13). Генерация таких графов не представляет собой трудностей, они, безусловно, обладают свойством локальности, которое не характерно для случайных графов.

Эвклидов граф с близкими связями. Граф, показанный в нижней части рис. 17.13, вычерчен генератором, который выбирает на плоскости V точек со случайными координатами в диапазоне от 0 до1, а затем генерирует ребра, соединяющие две точки, удаленные друг от друга на расстояние, не превышающее d. Если d невелико, то граф получается разреженным, если d большое, то граф насыщенный (см. упражнение 17.74). Такой граф моделирует типы графов, с которыми мы обычно сталкиваемся при работе с картами, электронными схемами или другими приложениями, в условиях которых вершины привязаны к определенным геометрическим точкам. Их нетрудно представить наглядно, они позволяют подробно отобразить свойства алгоритмов и структурные свойства, которые обнаруживаются в приложениях подобного рода.

Один из возможных недостатков этой модели состоит в том, что разреженные графы с большой долей вероятности могут не быть связными; другая связанная с ними трудность заключается в том, что маловероятно, чтобы они содержали вершины высокой степени, а также в том, что в них нет длинных ребер. При желании мы можем внести в эти модели соответствующие изменения, которые позволят нам найти выход из таких ситуаций, либо мы можем провести исследования многочисленных примеров подобного рода, чтобы попытаться смоделировать другие ситуации (см., например, упражнения 17.72 и 17.73).

Опять-таки, мы можем протестировать наши алгоритмы на реальных графах. В условиях многих приложений

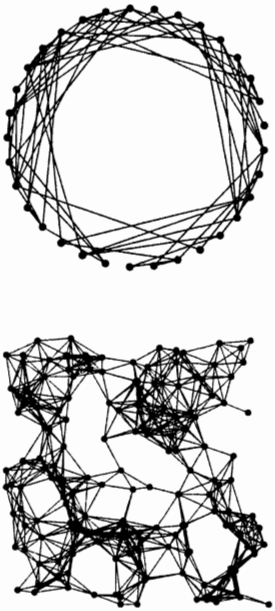


РИСУНОК 17.13. СЛУЧАЙНЫЕ ГРАФЫ С БЛИЗКИМИ СВЯЗЯМИ

Примеры, приводимые на этом рисунке, служат иллюстрацией двух моделей разреженных графов. Граф с близкими связями, изображенный в верхней части рисунка, содержит 33 вершины и 99 узлов, при этом каждое ребро может соединять заданную вершину с другой вершиной, индекс которой отличается от индекса заданной вершины не больше, чем на 10 (по модулю V). Эвклидов граф с близкими связями, показанный в нижней части рисунка, моделирует типы графов, которые мы можем найти в приложениях, в которых вершины привязаны к конкретным геометрическим точкам. Вершины изображены как случайные точки на плоскости; ребра соединяют любую пару вершин, расстояние между которыми не превышает d. *Этот граф относится к категории разреженных (177 вершин и 1001 ребер); варьируя значения* d, *мы можем построить граф любой степени насыщенности.*

фактические данные порождают массу примеров задач, которые мы с успехом можем использовать для тестирования наших алгоритмов. Например, крупные графы, полученные на базе реальных географических данных, встречаются довольно часто, еще два таких примера приводятся в последующих параграфах. Преимущество работы с фактическими данными, а не с моделями случайных графов, заключается в том, что мы можем находить решения реальных задач в процессе совершенствования алгоритмов. Недостаток определяется тем, что мы можем потерять привилегию оценивать производительность разрабатываемых нами алгоритмов, используя для этой цели методы математического анализа. Мы вернемся к этой теме в конце главы 18, когда будем готовы провести сравнение применения нескольких различных алгоритмов к решению одной и той же задачи.

Граф транзакций. На рис. 17.14 показан всего лишь небольшой фрагмент графа, который мы можем обнаружить в компьютерах телефонной компании. В этом графе каждому телефонному номеру определяется собственная вершина, а каждое ребро, соединяющее пару *i* и *j*, наделено свойством, согласно которому *i* производит телефонный звонок *j* в течение некоторого фиксированного промежутка времени. Это множество ребер представляет собой мультиграф огромных размеров. Он, естественно, разрежен, поскольку каждый абонент охватывает своими звонками всего лишь мизерную часть установленных телефонов. Этот граф является представительным для многих приложений. Например, кредитная карточка финансового учреждения и запись в счете торгового агента могут содержать информацию аналогичного характера.

900-435-5100	201-332-4562
415-345-3030	757-995-5030
757-310-4313	201-332-4562
747-511-4562	609-445-3260
900-332-3162	212-435-3562
617-945-2152	408-310-4150
757-995-5030	757-310-4313
212-435-3562	803-568-8358
913-410-3262	212-435-3562
401-212-4152	907-618-9999
201-232-2422	415-345-3120
913-495-1030	802-935-5112
609-445-3260	415-345-3120
201-310-3100	415-345-3120
408-310-4150	802-935-5113
708-332-4353	803-777-5834
413-332-3562	905-828-8089
815-895-8155	208-971-0020
802-935-5115	408-310-4150
708-410-5032	212-435-3562
201-332-4562	408-310-4150
815-511-3032	201-332-4562
301-292-3162	505-709-8080
617-833-2425	208-907-9098
800-934-5030	408-310-4150
408-982-3100	201-332-4562
415-345-3120	905-569-1313
413-435-4313	415-345-3120
747-232-8323	408-310-4150
802-995-1115	908-922-2239

РИСУНОК 17.14. ГРАФ ТРАНЗАКЦИИ

Последовательности пар чисел, аналогичные показанным на этом рисунке, могут представлять список телефонных вызовов в местной телефонной станции, или финансовую операцию между двумя счетами, либо любую ситуацию подобного рода, когда производится транзакция между двумя элементами предметной области, которым присвоены уникальные идентификаторы. Такие графы нельзя рассматривать как число случайные, ибо некоторые телефонные номера используются намного чаще, чем остальные, а некоторые счета характеризуются большей активностью, чем многие другие.

Программа 17.14. Построение графа из пар символов

Данная реализация функции **scan** из программы 17.4 использует таблицу символов для построения графа путем считывания пар символов из стандартного ввода. Функция **index** АТД таблицы символов ставит некоторое целое число в соответствие каждому символу: если поиск в таблице размера **N** закончится неудачно, она добавляет в таблицу символ с привязанным к нему целым числом **N+1**; если поиск завершится успешно, она просто возвращает целое число, которое ранее было ассоциировано с заданным символом. Любой из методов построения таблиц символов, рассмотренных в

части 4, может быть приспособлен для решения этой задачи (см., например, программу 17.15).

```
#include  "ST.cc"
template  <class  Graph>
void  IO<Graph>::scan(Graph  &G)
  { string v, w;
    ST  st;
    while  (cin  >>  v  >>  w)
        G.insert(Edge(st.index(v),  st.index(w)));
  }
```

Граф вызовов функций. Мы можем поставить граф в соответствие любой компьютерной программе, отождествляя функции с вершинами графа, при этом ребро соединяет вершину X с вершиной Y всякий раз, когда функция X вызывает функцию Y. Мы можем снабдить программу средствами построения таких графов (или заставить компилятор делать это). Для нас интерес представляют два абсолютно различных графа: статическая версия, когда мы строим ребра во время компиляции в соответствии с вызовами функций, которые появляются в программном тексте каждой функции; и динамическая версия, когда мы строим ребра во время выполнения программы, т.е. когда эти вызовы фактически происходят. Мы используем статические графы вызовов функций при изучении структуры программы и динамические графы при изучении поведения программы. Как правило, такие графы принимают большие размеры и относятся к категории разреженных.

В приложениях, подобных рассматриваемым в этой книге, мы имеем дело с крупными массивами данных, так что довольно часто мы отдаем предпочтение изучению рабочих характеристик алгоритмов на реальных данных, а не на вероятностных моделях. Мы можем предпринять попытку избежать тупиковой ситуации за счет случайного выбора ребер или за счет введения в наши алгоритмы случайного фактора, который проявляется в процессе принятия решений, однако генерация случайных графов – это совсем другое дело. По существу, изучение свойств различных структур графов уже само по себе представляет несомненных интерес.

Программа 17.15. Символьная индексация имен вершин

Реализация индексации строковых ключей посредством таблицы символов (описание которой дано в пояснении к программе 17.14) завершает эту задачу добавлением поля **index** к каждому узлу TST-дерева (ternary search tree –дерево тернарного поиска) таблицы существования (см. программу 15.8). Индекс, ассоциированный с каждым ключом, хранится в поле индекса узла, соответствующего символу конца его строки.

Как обычно, мы используем символы ключа поиска для перемещения по TST-дереву сверху вниз. Когда мы достигнем конца ключа, мы при необходимости устанавливаем значение его индекса, а также устанавливаем значение приватного элемента данных **val**, которое возвращается вызывающему объекту после того, как все рекурсивные обращения к рассматриваемой функции возвратят соответствующие значения.

```
#include  <string>
class  ST
{ int N, val;
  struct  node
    { int v, d; node* l, *m, *r;
      node(int d)  : v(-1), d(d), l(0), m(0), r(0) {}
    };
```

```
  typedef node* link;
  link head;
  link indexR(link h, const string &s, int w)
   { int i = s[w];
     if (h == 0) h = new node(i);
     if (i == 0)
      {
        if (h->v == -1) h->v = N++;
        val = h->v;
        return h;
      }
     if (i < h->d) h->l = indexR(h->l, s, w);
     if (i == h->d) h->m = indexR(h->m, s, w+1);
     if (i > h->d) h->r = indexR(h->r, s, w);
     return h;
   }
public:
   ST() : head(0), N(0) { }
   int index(const string &key)
    { head = indexR(head, key, 0); return val; }
};
```

В нескольких таких примерах вершины представляют собой естественные имена объектов, а ребра выступают в виде пары именованных объектов. Например, граф транзакций может быть построен на базе последовательности пар телефонных номеров, а эвклидов граф — на базе последовательности пар городов или населенных пунктов. Программа 17.14 представляет собой реализацию функции **scan** из программы 17.14, которой мы можем воспользоваться для построения графов общей ситуации подобного рода. Для удобства клиентских программ она использует некоторое множество ребер в качестве определения графа и вычисляет множество имен вершин графа по мере того, как они появляются в ребрах. В частности, рассматриваемая программа считывает последовательность пар символов из стандартного ввода, использует таблицу символов для того, чтобы поставить в соответствие символам номера вершин в диапазоне от 0 до $V-1$ (где V — число различных символов во вводе), и строит граф путем вставки ребер, как это имело место в программах 17.12 и 17.13. Мы можем приспособить реализацию, использующую таблицу символов, для поддержки средств, необходимых программе 17.14; программа 17.15 представляет собой пример использования TST-деревьев (см. главу 14). Эти программы существенно упрощают задачу тестирования разрабатываемых алгоритмов на реальных графах, которые, по всей видимости, не допускают точного представления с помощью какой бы то ни было вероятностной модели.

Значение программы 17.15 велико еще потому, что она подтвердила наше предположение, которое мы допускали во всех разрабатываемых нами программах и которое заключается в том, что имена вершин являются целочисленными значениями в диапазоне от 0 до $V-1$. Если в нашем распоряжении имеется граф с другим множеством имен вершин, то первым шагом в построении представления графа является выполнение программы 17.15, отображающей имена вершин на целые числа в диапазоне от 0 до $V-1$.

В основе некоторых графов лежат неявные связи между его элементами. Мы не будем рассматривать такие графы, однако подтвердим их существование несколькими следующими примерами и посвятим им несколько упражнений. Столкнувшись с задачей обработки такого графа, мы, естественно, можем написать программу построения явных

графов путем нумерации всех его ребер, в то же время существуют задачи, решение которых не требует, чтобы мы нумеровали все ребра, благодаря чему время их выполнения подчиняется сублинейной зависимости.

Граф со степенями разделения. Рассмотрим некоторую совокупность подмножеств из V элементов. Мы определяем граф таким образом: его вершина соответствует каждому элементу объединения подмножеств, а ребро между двумя вершинами проводится в том случае, если обе вершины встречаются в каком-то одном поднаборе (см. рис. 17.15). При желании этот граф может быть преобразован в мультиграф, в котором метки ребер именуют соответствующие подмножества. Говорят, что все элементы, инцидентные данному элементу **v**, отделены от вершины **v** одной *степенью разделения* (*degree of separation*). В противном случае все элементы, инцидентные какому-либо элементу, который отделен i степенями разделения от вершины **v**, (о которых еще не известно, сколькими степенями разделения они отделены от вершины **v**, i степенями или меньшим числом степеней разделения), обладают $i + 1$ степенями разделения с вершиной **v**. Такая конструкция служила развлечением для многих людей, от математиков (числа Эрдеша (Erdos)) до деятелей киноискусства (отделение от Кевина Бэкона (Kevin Bacon)).

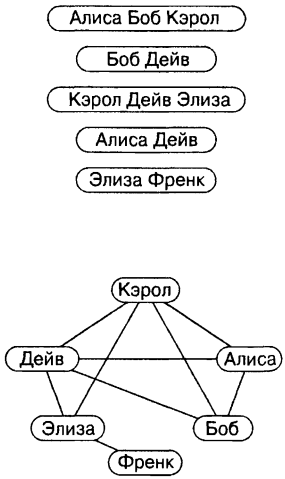


РИСУНОК 17.15. ГРАФ СО СТЕПЕНЯМИ РАЗДЕЛЕНИЯ

Граф в нижней части рисунка определяется группами, показанными в верхней части рисунка, при этом каждому имени соответствует вершина, а ребра соединяют вершины с именами, попадающими в одну и ту же группу. Кратчайшие длины пути в рассматриваемом графе соответствуют степеням разделения. Например, Франк отдален на три степени разделения от Алисы и Боба.

Интервальный граф. Рассмотрим совокупность V интервалов на действительной оси (пары вещественных чисел). Мы определяем граф следующим образом: каждому интервалу ставится в соответствие вершина, ребра между вершинами проводятся в том случае, когда соответствующие интервалы пересекаются (имеют любое число общих точек).

Граф Де-Бруйна. Предположим, что V равно степени 2. Мы определяем орграф следующим образом: каждому неотрицательному целому числу, меньшему V, соответствует одна вершина графа, ребра направлены из каждой вершины i в вершины $2i$ и $(2i + 1) \bmod \lg V$. Эти графы полезны при исследовании последовательностей значений, которые возникают в сдвиговых регистрах фиксированной длины при выполнении последовательностей операций, в рамках которых мы неоднократно сдвигаем все разряды на одну позицию влево, отбрасываем самый левый разряд и заполняем самый правый разряд нулем или единицей. На рис. 17.16 изображены графы Де-Бруйна (de Bruijn) с 8, 16, 32 и 64 вершинами.

Различные типы графов, которые мы рассмотрели в этом разделе, обладают широким набором характеристик. Однако с точки зрения наших программ они выглядят одинаково: это просто множества ребер. Как мы уже убедились в главе 1, чтобы установить простейшие свойства графов, нужно преодолеть сложные вычислительные проблемы. В этой

книге мы рассматриваем замысловатые алгоритмы, которые разрабатывались с целью решения практических задач, связанных со многими типами графов.

На базе нескольких примеров, рассмотренных в данном разделе, мы приходим к заключению, что графы представляют собой сложные комбинаторные объекты, которые намного сложнее тех, что служат основой других алгоритмов, обсуждавшихся в главах 1–4. Во многих случаях графам, с которыми приходится иметь дело в приложениях, трудно или даже невозможно дать более-менее точную характеристику. Алгоритмы, которые хорошо показывают себя на случайных графах, часто не находят широкого применения ввиду того, что зачастую трудно убедиться в том, что случайные графы обладают теми же структурными характеристиками, что и графы, возникающие в приложениях. Обычный способ преодолеть это возражение предусматривает разработку алгоритмов, которые хорошо работают в наихудших случаях. И если этот подход приносит успех в одних случаях, он не оправдывает ожиданий в других (в силу своей консервативности).

В то время как не всегда оправдываются наше предположение, что изучение рабочих характеристик на графах, построенных на базе одной из рассмотренных вероятностных моделей графа, даст нам достаточно точную информацию, которая обеспечила бы возможность прогнозировать производительность реальных графов, генераторы графов, которые мы рассматривали в данном разделе, полезны для тестирования реализаций исследуемых алгоритмов и понимания их сущности. Перед тем, как мы попытаемся спрогнозировать производительность приложения, мы, по меньшей мере, должны подтвердить правильность всех предположений, которые мы, возможно, сделали относительно отношений, связывающих данные приложения с какими бы то ни было моделями или образцами, которыми мы могли воспользоваться. И если такая проверка целесообразна при работе в любой области приложений, она особенно важна при обработке графов, что объясняется широким набором типов графов, с которыми приходится сталкиваться.

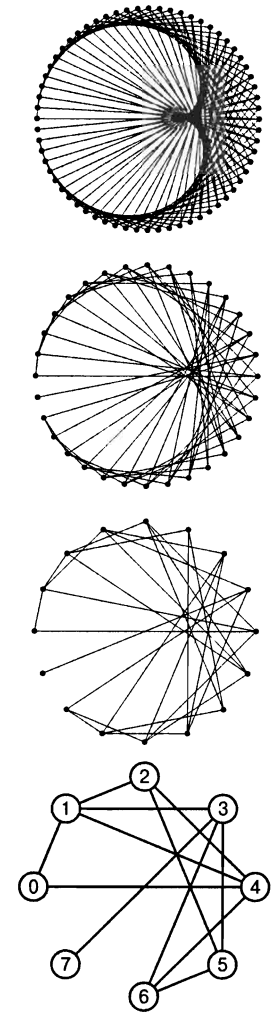


РИСУНОК 17.16. ГРАФЫ ДЕ-БРУЙНА

Орграф Де-Бруйна порядка n содержит 2^n вершин, ребра исходят из вершины i в вершины $2i \bmod n$ и $(2i + 1) \bmod 2^n$ для всех i. На рисунке показаны служащие основой орграфов Де-Бруйна неориентированные графы порядка 6, 5, 4 и 3 (сверху вниз).

Упражнения

▷ **17.61.** в тех случаях, когда для построения случайных графов со степенью насыщенности αV мы используем программу 17.12, какую часть генерируемых ребер составляют петли?

• **17.62**. Вычислите ожидаемое число построенных параллельных ребер при условии, что для генерации случайных графов с V вершинами и насыщенностью **a** мы используем программу 17.12. Воспользуйтесь результатами этих вычислений для построения кривых, показывающих в виде функции от насыщенности **a**, какую часть от общего числа ребер составляют параллельные ребра при $V = 10$, 100 и 1000.

17.63. Воспользуйтесь функцией **map** из библиотеки STL для разработки реализации класса **ST**, альтернативной реализации, представленной программой 17.15.

• **17.64**. Найдите большой неориентированный граф в какой-нибудь динамической предметной области, возможно, граф, основанный на данных о связности узлов сети, либо граф со степенями разделения, определенный соавторами на множестве библиографических списков или актерами на множестве кинофильмов.

▷ **17.65**. Напишите программу, которая генерирует разреженные случайные графы для правильно выбранных значений V и E и печатает, какое пространство памяти используется для представления графа и какое время потребовалось для его построения. Проведите тестирование разработанной программы с использованием класса разреженного графа (программа 17.9) и генератора случайных графов (программа 17.12) с тем, чтобы вы могли применять обоснованные эмпирические критерии к графам, построенным на базе этой модели.

▷ **17.66**. Напишите программу, которая генерирует разреженные случайные графы для удачно подобранных множеств значений V и E и печатает, какое пространство памяти используется для представления графа и какое время потребовалось для его построения. Проведите тестирование программы с использованием класса насыщенного графа (программа 17.7) и генератора случайных графов (программа 17.13) с тем, чтобы вы могли применять обоснованные эмпирические критерии к графам, построенным на базе этой модели.

• **17.67**. Вычислите среднеквадратическое отклонение числа ребер, построенных программой 17.13.

• **17.68**. Напишите программу, которая строит каждый возможный граф с точно такой же вероятностью, что и программа 17.13, но в то же время затрачивает время и пространство памяти, пропорциональное $V+E$, но не V^2. Проведите тестирование полученной программы в соответствии с изложенным в упражнении 17.65.

○ **17.69**. Напишите программу, которая строит с равной вероятностью каждый возможный граф с точно такой же вероятностью, что и программа 17.12, но в то же время затрачивает время и пространство памяти, пропорциональное E, даже если степень насыщенности графа близка к 1. Проведите тестирование полученной программы в соответствии с изложенным в упражнении 17.66.

• **17.70**. Напишите программу, которая строит с равной вероятностью каждый возможный граф с V вершинами и E ребрами (см. упражнение 17.9). Проведите тестирование полученной программы в соответствии с изложенным в упражнении 17.65 (для низких уровней насыщенности) и в соответствии с изложенным в упражнении 17.66 (для высоких уровней насыщенности).

○ **17.71**. Напишите программу, которая строит случайные орграфы путем соединения вершин, упорядоченных в виде решетки размером $\sqrt{V}$ на $\sqrt{V}$, с соседними вершинами (см. рис. 1.2), при этом k дополнительных ребер соединяют каждую вершину со случайно выбранной вершиной назначения (выбор любой вершины назначения рав-

новероятен). Определите, какое значение k следует выбрать, чтобы ожидаемое число ребер было равно E. Проведите тестирование полученной программы в соответствии с изложенным в упражнении 17.65.

○ **17.72**. Напишите программу, которая строит случайные графы путем соединения вершин, упорядоченных в виде решетки размером $\sqrt{V}$ на $\sqrt{V}$, с соседними вершинами, при этом каждое ребро появляется с вероятностью p (см. рис. 1.2). Определите, какое значение p следует выбрать, чтобы ожидаемое число ребер было равно E. Проведите тестирование полученной программы в соответствии с изложенным в упражнении 17.65.

○ **17.73**. Расширьте программу из упражнения 17.72 путем добавления R дополнительных случайных ребер, вычисленных методом, реализованным программой 17.12. Для больших R сожмите решетку настолько, чтобы общее число ребер оставалось примерно равным V.

● **17.74**. Напишите программу, которая генерирует V случайных точек на плоскости, после чего строит граф, состоящий из ребер, соединяющих все пары точек, удаленных друг от друга на расстояние, не превышающее d (см. рис. 17.13 и программу 3.20). Определите, какое значение d следует выбрать, чтобы ожидаемое число ребер было равно E. Проведите тестирование полученной программы в соответствии с изложенным в упражнении 17.65 (для низких уровней насыщенности) и в соответствии с изложенным в упражнении 17.66 (для высоких уровней насыщенности).

● **17.75**. Напишите программу, которая генерирует V случайных интервалов в единичном интервале, каждый длиной d, затем постройте соответствующий интервальный граф. Определите, какое значение d следует выбрать, чтобы ожидаемое число ребер было равно E. Проведите тестирование полученной программы в соответствии с изложенным в упражнении 17.65 (для низких уровней насыщенности) и в соответствии с изложенным в упражнении 17.66 (для высоких уровней насыщенности). *Указание*: Воспользуйтесь BST-деревом (Binary Search Tree – дерево двоичного поиска).

● **17.76**. Напишите программу, которая случайным образом выбирает V вершин и E ребер из реального графа, который вы найдете в упражнении 17.64. Проведите тестирование полученной программы в соответствии с изложенным в упражнении 17.65 (для низких уровней насыщенности) и в соответствии с изложенным в упражнении 17.66 (для высоких уровней насыщенности).

○ **17.77**. Один из способов описания транспортной системы предусматривает использование множества последовательностей вершин, при этом каждая такая последовательность определяет путь, соединяющий соответствующие вершины. Например, последовательность **0-9-3-2** определяет ребра **0-9**, **9-3**, **3-2**. Напишите программу, которая строит граф по данным из входного файла, содержащего в каждой строке одну последовательность, с использованием символических имен. Подготовьте ввод, который позволил бы вам использовать эту программу для построения графа, соответствующего схеме парижского метро.

17.78. Обобщите ваше решение упражнения 17.77 на случай, когда заданы координаты вершин и выполнены условия, сформулированные в упражнении 17.60, что позволило бы вам работать с графическими представлениями графов.

○ **17.79**. Примените преобразования, описанные в упражнениях 17.34–17.37, к различным графам (см. упражнения 17.63–17.76) и сведите в таблицу число вершин и ребер, удаленных при каждом таком преобразовании.

○ **17.80**. Постройте реализацию конструктора для программы 17.1, которая позволяет клиентам строить граф разделения без необходимости вызова функции для каждого неявного ребра. То есть, количество вызовов функции, необходимых клиенту, чтобы построить граф, должно быть пропорционально сумме размеров групп. Разработайте эффективную реализацию этого модифицированного АТД (основанного на структурах данных с использованием групп, но без использования неявных ребер).

17.81. Определите строгую верхнюю границу числа ребер любого графа разделения для N различных групп с k человек в каждой группе.

▷ **17.82**. Сделайте чертеж графа в стиле рис. 17.16, который содержит V вершин, пронумерованных от 0 до $V-1$, и ребра, соединяющие вершину **i** с вершиной **[i/2]** для $V = 8$, 16 и 32.

17.83. Внесите изменения в интерфейс АТД из программы 17.1, который позволил бы клиентам использовать символьные имена вершин и имена ребер в виде пар экземпляров обобщенного типа **Vertex**. Полностью скройте представление, использующее индексацию именами вершин и АТД с таблицей символов, от клиентских программ.

17.84. Добавьте в интерфейс АТД функцию из упражнения 17.83, которая поддерживает операцию *объединить* (*join*) на графах, и постройте реализации, генерирующие представления графов в виде матрицы смежности и списков смежных вершин. *Примечание*: Любая вершина или ребро любого графа должны быть указаны в операции *join*, но вершины, которые фигурируют в обоих графах, в операции *join* должны быть указаны только один раз, кроме того, вы должны удалить параллельные ребра.

17.7. Простые, эйлеровы и гамильтоновы пути

Первый рассмотренный нами нетривиальный алгоритм обработки графов решает фундаментальные задачи, касающиеся поиска путей в графах. Для решения этих задач мы вводим принцип общей рекурсии, который будем широко применять на протяжении всей книги, они служат иллюстрацией того, что на первый взгляд очень похожие задачи существенно различаются по трудности их решения.

Эти задачи уводят нас от локальных свойств, таких как существование конкретных ребер или определение степеней вершин, к глобальным свойствам, которые могут многое сказать о структуре графа. Фундаментальное свойство графа говорит нам, связаны ли какие-либо две вершины графа или нет. Если они связаны, то возникает следующий вопрос, который нас интересует: как найти простейший путь, который их связывает.

Простой путь. Если заданы две какие-либо вершины графа, существует ли путь, который их соединяет? В условиях некоторых приложений нам вполне достаточно знать, существует или не существует такой путь, но данном случае наша задача заключается в том, чтобы найти конкретный путь.

Программа 17.16 предлагает решение этой задачи. В ее основу положен *поиск в глубину* (*depth-first search*) — фундаментальный принцип обработки графов, на котором мы кратко останавливались в главах 3 и 5 и который подробно будем изучать в граве 18. Этот алгоритм построен на базе приватной функции-элемента, которая определяет, существует ли простой путь из вершины **v** в вершину **w** путем проверки для каждого ребра **v-t** инцидентного **v**, существует ли простой путь из **t** в **w**, который не проходит через **v**. Он использует вектор, индексированный именами вершин, с целью пометить **v**, чтобы ни при каком рекурсивном вызове путь, проходящий через **v**, не проверялся.

Программа 17.16. Поиск простого пути

Этот класс использует рекурсивную функцию поиска в глубину **searchR**, чтобы найти простой путь, соединяющий две заданных вершины графа, и предоставляет клиенту функцию-элемент **exist**, чтобы клиентская программа могла проверить, существует ли путь между этими вершинами. В случае, когда заданы две вершины **v** и **w**, функция **searchR** проверяет каждое ребро **v-t**, смежное с **v**, может ли оно быть первым ребром на пути к **w**. Вектор **visited**, индексированный именами вершин, предотвращает повторный заход на любую вершину, так что обеспечивается прохождение только простых путей.

```
template <class Graph> class sPATH
{ const Graph &G;
   vector <bool> visited;
   bool found;
   bool searchR(int v, int w)
    {
      if (v == w) return true;
      visited[v] = true;
      typename Graph::adjIterator A(G, v);
      for (int t = A.beg(); !A.end(); t = A.nxt())
        if (!visited[t])
          if (searchR(t, w)) return true;
      return false;
    }
public:
  sPATH(const Graph &G, int v, int w) :
    G(G), visited(G.V(), false)
    { found = searchR(v, w); }
  bool exists() const
    { return found; }
};
```

Программа 17.16 просто проверяет, существует тот или иной путь. Как следует расширить ее с тем, чтобы она могла печатать ребра, составляющие путь? Рекурсивный подход предлагает простое решение:

- Добавить оператор печати ребра **v-t** сразу же после того, как рекурсивный вызов в функции **searchR** находит путь из **t** в **w**.
- В вызове функции **searchR** из конструктора поменять местами **v** и **w**.

Первое изменение приводит к тому, что путь из **v** в **w** печатается в обратном порядке: если в результате вызов **searchR(t, w)** находит путь из **t** в **w** (и печатает ребра, составляющие этот путь, в обратном порядке), то печать пути **t-v** завершает решение задачи поиска пути из **v** в **w**. Второе изменение меняет порядок: чтобы распечатать ребра, составляющие путь из **v** в **w**, мы распечатываем путь из **w** в **v** в обратном порядке. (Этот прием работает только в случае неориентированных графов.) Мы могли бы применить эту стратегию для реализации функции АТД, которая вызывает функцию, переданную клиентом, для каждого ребра, составляющего путь (см. упражнение 17.88).

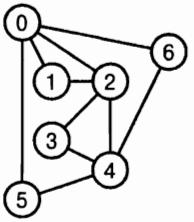
Рисунок 17.17 представляет собой пример динамики рекурсии. Как и в случае любой другой рекурсивной программы (в принципе, любой программы с вызовами функций), такую трассировку нетрудно вывести на печать: чтобы внести соответствующие изменения в программу 17.16, мы можем добавить переменную **depth**, значение которой увеличивается на 1 при входе и уменьшается на 1 при выходе, чтобы отслеживать глубину ре-

курсии, затем поместить в начало рекурсивной функции программный код, осуществляющий **depth** пропусков при печати, за которыми следует соответствующая информация (см. упражнения 17.86 и 17.87).

Свойство 17.2. *Мы можем найти путь, соединяющий две заданных вершины графа, за линейное время.*

Рекурсивная функция поиска в глубину, реализованная в программе 17.16, представляет собой доказательство методом индукции того факта, что функция АТД определяет, существует искомый путь или нет. Такое доказательство легко можно расширить с целью установки, что в худшем случае программа 17.16 проверяет все элементы матрицы смежности в точности один раз. Аналогично, мы можем показать, что подобного рода программы, ориентированные на списки смежных вершин, проверяют в худшем случае все ребра графа в точности два раза (по разу в каждом направлении). ■

Мы используем термин *линейный (linear)* в контексте алгоритмов на графах, подразумевая под этим, что некоторое количественное значение не превосходит величины $V + E$ (т.е. размеры графа), умноженной на соответствующий постоянный коэффициент. Как мы отметили в конце раздела 17.5, такое значение обычно не превосходит размера представления графа, умноженного на соответствующий постоянный коэффициент. Свойство 17.2 сформулировано таким образом, что оно позволяет использовать представление разреженных графов в виде списков смежных вершин и представление насыщенных графов в виде матрицы смежности, что, собственно говоря, и является нашей обычной практикой. Термин "линейный" не принято применять для описания алгоритма, который использует матрицу смежности и выполняется за время, пропорциональное V^2 (даже если он линеен по отношению к размерам представления графа), *если* рассматриваемый граф не принадлежит к числу насыщенных. В самом деле, если мы используем представление графа в виде матрицы смежности, мы не можем подобрать алгоритм, линейный в смысле времени исполнения, для любой задачи обработки графа, которая требует анализа каждого ребра.




```
2-0 searchR(G, 0, 6)
  0-1 searchR(G, 1, 6)
    1-0
    1-2
  0-2
  0-5 searchR(G, 5, 6)
    5-0
    5-4 searchR(G, 4, 6)
      4-2
      4-3 searchR(G, 3, 6)
        3-2
        3-4
      4-6 searchR(G, 6, 6)
```

РИСУНОК 17.17. ТРАССИРОВКА ПОИСКА ПРОСТОГО ПУТИ

*Данная трассировка показывает, как работает рекурсивная функция из программы 17.16 на примере вызова **searchR(G, 2, 6)** с целью поиска простого пути из вершины **2** в вершину **6** на графе, который показан в верхней части рисунка. Для каждого исследуемого ребра в трассе отводится отдельная строка с отступом на один уровень для каждого рекурсивного вызова. Чтобы проверить ребро **2-0**, мы осуществляем вызов **searchR(G, 0, 6)**. Этот вызов заставляет нас проверить ребра **0-1**, **0-2** и **0-5**. Чтобы проверить ребро **0-1**, мы делаем вызов **searchR(G, 1, 6)**, который заставляет нас проверить ребра **1-0** и **1-2**, которые не приводят к рекурсивным вызовам, поскольку вершины **0** и **2** уже помечены. В этом примере рассматриваемая функция обнаруживает путь **2-0-5-4-6**.*

Мы проведем подробные исследования алгоритма поиска в глубину в его более общей формулировке в следующем разделе, там же мы рассмотрим несколько других алгоритмов связности. Например, несколько более общая версия программы 17.16 предлагает нам способ обхода всех ребер графа путем построения вектора, индексированного именами вершин, который позволяет клиенту проверить за постоянное время, существует ли путь, соединяющий какие-либо две вершины.

Свойство 17.2 дает существенно завышенную оценку фактического времени прогона программы 17.16, поскольку она может найти путь после исследования всего лишь нескольких ребер. В данный момент все, что нас интересует, это освоение метода, который гарантированно обеспечивает определение пути, соединяющего любые пары вершин любого графа за линейное время. В отличие от этого, другие задачи, которые на первый взгляд мало чем отличаются от рассматриваемой проблемы, намного труднее поддаются решению. Например, рассмотрим следующую задачу, в рамках которой мы осуществляем поиск пути, соединяющих пары вершин, но при этом добавляется условие, согласно которому эти пути проходят через все остальные вершины графа.

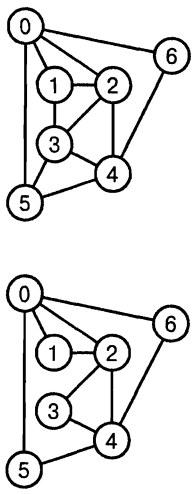


РИСУНОК 17.18. ГАМИЛЬТОНОВ ПУТЬ

*В графе, помещенном в верхней части рисунка, имеется гамильтонов путь **0-6-4-2-1-3-5-0**, который заходит в каждую вершину в точности один раз и возвращается в ту вершину, в которой он начинался; в то же время, граф в нижней части рисунка такого пути не содержит.*

Гамильтонов путь. Пусть даны две вершины, существует ли простой путь, соединяющий эти вершины, который проходит через каждую вершину графа в точности один раз? Если этот путь приводит в ту же вершину, из которой он вышел, то эта задача известна как задача поиска гамильтонова цикла. Существует ли цикл, который проходит через каждую вершину в точности один раз?

На первый взгляд, кажется, что эта задача решается просто: достаточно внести некоторые простые изменения в рекурсивную часть класса, осуществляющего поиск пути, который показан в программе 17.16. Однако, эта программа, по-видимому, не может быть пригодной для всех графов, поскольку время ее прогона в худшем случае находится в экспоненциальной зависимости от числа вершин в графе.

Программа 17.17. Гамильтонов путь

Данная рекурсивная функция отличается от используемой в программе 17.16 всего лишь в двух отношениях: во-первых, она принимает длину искомого пути в качестве третьего аргумента и возвращает значение, означающее успех, только в случае, если находит путь длины **V**; во-вторых, она переустанавливает значение маркера **visited**, прежде чем возвратит значение, означающее неуспех.

Если мы заменим рекурсивную функцию, используемую программой 17.16, на приводимую ниже и добавим третий аргумент **G.V()-1** в вызов функции **searchR** из функции **search**, то **search** будет выполнять поиск гамильтонова пути. *Однако не рассчитывайте, что поиск будет выполнен до конца, если только он не проводится на графах совсем небольших размеров (см. рассуждения по тексту раздела).*

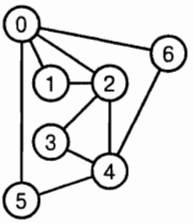
```
bool searchR(int v, int w, int d)
  {
    if (v == w) return (d == 0);
    visited[v] = true;
    typename Graph::adjIterator A(G, v);
    for (int t = A.beg(); !A.end(); t = A.nxt())
      if (!visited[t])
        if (searchR(t, w, d-1)) return true;
    visited[v] = false;
    return false;
}
```

Свойство 17.3. *Рекурсивный поиск гамильтонова цикла может потребовать экспоненциального времени.*

Доказательство: Рассмотрим граф, у которого вершина **V-1** изолирована, а ребра, связывающие остальные $V-1$ вершин, образуют полный граф. Программа 17.17 никогда не найдет гамильтонова пути, в то же время, применив метод индукции, легко убедиться в том, что она исследует все $(V-1)!$ путей в полном графе, каждый из которых требует $V-1$ рекурсивных вызовов. Следовательно, общее число рекурсивных вызовов равно $V!$ или примерно $(V/e)^V$, что больше, чем любая константа в V-той степени. ■

Полученные нами реализации – программа 17.16, осуществляющая поиск простых путей, и программа 17.17, выполняющая поиск гамильтоновых путей, – очень похожи друг на друга. Если таких путей не существует, выполнение обеих программ прекращается, когда всем элементам вектора **visited** установлено значение **true**. Почему времена прогона этих программ столь разительно отличаются друг от друга? Прогон программы 17.16 гарантированно выполняется за короткое время, поскольку она устанавливает, по меньшей мере, один элемент вектора **visited** в 1 всякий раз, когда вызывается функция **searchR**. С другой стороны, программа 17.17 может устанавливать элементы вектора **visited** снова в 0, так что мы не можем гарантировать, что ее выполнение завершится скоро.

Выполняя поиск простых путей с помощью программы 17.16, мы знаем, что если существует путь из **v** в **w**, мы найдем его, выбирая одно из ребер **v-t**, исходящее из **v**; то же можно сказать и о гамильтоновых путях. Но на этом сходство заканчивается. Если мы не можем найти простой путь из **t** в **w**, то приходим к выводу, что простого пути из **v** в **w**, проходящего через **t**, не существует; однако такая ситуация в процессе поиска гамильтонова пути не возникает. Может случиться так, что в графе нет гамильтонова пути в вершину **w**, который начинается с ребра **v-t**, но есть путь, который начинается с **v-x-t** и проходит через вершину **x**. Мы должны выполнять рекур-




```
0-1
  1-2
    2-3
      3-4
        4-5
        4-6
    2-4
      4-3
      4-5
      4-6
0-2
  2-1
  2-3
    3-4
      4-5
      4-6
  2-4
    4-3
    4-5
    4-6
0-5
  5-4
    4-2
      2-1
      2-3
    4-3
      3-2
        2-1
    4-6
0-6
```

РИСУНОК 17.19. ТРАССИРОВКА ПОИСКА ГАМИЛЬТОНОВА ПУТИ

Эта трассировка показывает ребра, проверенные программой 17.17 для того, чтобы убедиться, что граф, приведенный в верхней части рисунка, не имеет гамильтонова цикла. Для краткости ребра, входящие в помеченные вершины, опущены.

сивные вызовы из t, соответствующие каждому пути, который ведет в нее из вершины v. Короче говоря, нам, возможно, придется проверить каждый путь в графе.

Полезно задуматься над тем, насколько медленно действующим является алгоритм, время выполнения которого пропорционально факториалу количества вершин. Допустим, что мы можем обработать граф с 15 вершинами за 1 секунду, тогда обработка графа с 19 вершинами будет длиться целые сутки, более года, если граф содержит 21 вершину, и 6 столетий, если граф содержит 23 вершины. Увеличение быстродействия компьютера мало чем может помочь. Если повысить быстродействие компьютера в 200000 раз, то для решения рассматриваемой задачи с 23 вершинами ему потребуется больше суток. Стоимость обработки графа с 100 или 1000 вершинами такова, что страшно подумать, не говоря уже о графах, с которыми нам приходится сталкиваться на практике. Потребуются многие миллионы страниц этой книги, чтобы только записать число веков, в течение которых будет длиться обработка графа, содержащего миллионы вершин.

В главе 5 мы провели исследование некоторого числа простых рекурсивных программ, которые по характеру подобны программе 17.17, однако качество этих исследований можно было существенно повысить за счет применения нисходящего динамического программирования. Данная рекурсивная программа по своему характеру полностью от них отличается: количество промежуточных результатов, которые требуется сохранять в памяти, возрастает по экспоненте. Несмотря на то что многие исследователи затрачивают громадные усилия на решение этой задачи, ни одному из них еще не удалось найти алгоритм, который обеспечивал приемлемую производительность при обработке графов больших (и даже средних) размеров.

Теперь предположим, что мы изменили начальные условия, и требование обязательного обхода всех *вершин* заменяется требованием обхода всех *ребер*. Является ли эта задача такой же легкой, как и нахождение простых путей, или безнадежно трудной проблемой, подобной поиску гамильтоновых путей?

Эйлеров путь. Существует ли путь, соединяющий две заданных вершины, который проходит через каждое *ребро* графа в точности один раз? Путь не обязательно должен быть простым – вершины можно посещать многократно. Если путь начинается из некоторой вершины и возвращается в ту же вершину, мы имеем задачу поиска *эйлерова цикла (Euler tour)*. Существует ли циклический путь, который проходит через каждое ребро графа в точности один раз? В следствии к свойству 17.4 мы докажем, что задача поиска такого пути эквивалентна задаче поиска цикла в графе, построенного путем добавления в граф ребра, соединяющего две соответствующие вершины. На рис. 17.20 приводятся два небольших примера.

Первым эту классическую задачу исследовал Л.Эйлер (L.Euler) в 1736 г. Некоторые математики склонны считать, что начало исследованию графов и теории графов

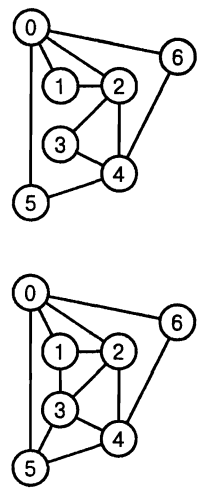


РИСУНОК 17.20. ПРИМЕРЫ ЭЙЛЕРОВЫХ ЦИКЛОВ И ЦЕПЕЙ

*Граф, показанный в верхней части рисунка, содержит эйлеров цикл **0-1-2-0-6-4-3-2-5-0**, который использует все ребра в точности один раз. Граф, изображенный в нижней части рисунка, не содержит таких циклов, однако содержит эйлеров путь **0-2-0-1-3-4-2-3-5-4-6-0-5**.*

положило появление в печати работы Эйлера, посвященной решению этой проблемы, известная как *задача о Кенигсбергских мостах* (см. рис. 17.21). В немецком городе Кенигсберг (с 1946 года – Калининград, входящий в состав России) семь мостов соединяли берега реки и острова, и жители этого города обнаружили, что они, по-видимому, не могут перейти все семь мостов, не пройдя по одному из них дважды. От этой задачи берет начало проблема поиска эйлерова цикла.

Эти задачи будоражили воображение энтузиастов. Их рассматривают как головоломки, по условиям которых вы должны вычертить заданную фигуру, не отрывая карандаша от бумаги, возможно, при условии, что вы должны начать и вернуться в заданную точку. Перед нами встает естественная задача поиска эйлеровых путей при разработке алгоритмов обработки графов, поскольку эйлеровы пути является эффективным представлением графов (упорядочение ребер графа определенным образом), которые могут служить основой разработки эффективных алгоритмов.

Эйлер нашел легкий способ определить, существует ли такая цепь, для этого достаточно определить степень каждой вершины. Это свойство нетрудно сформулировать и применять, однако его доказательство в теории графов требует немалой изобретательности.

Свойство 17.4. *Граф содержит эйлеров цикл тогда и только тогда, когда он связный и все его вершины имеют четную степень.*

Доказательство: Чтобы упростить доказательство, допустим существование петель и параллельных ребер, хотя совсем нетрудно изменить доказательство таким образом, чтобы показать, что это свойство справедливо и для простых графов (см. упражнение 17.94).

Если в графе имеется эйлеров цикл, то он должен быть связным в силу того, что цикл определяет путь, соединяет каждую пару вершин. Кроме того, степень любой заданной вершины графа **v** должна быть четной, поскольку, когда мы проходим цикл (начало которого находится в какой-то другой вершине), то мы входим в вершину **v** через одно ребро и выходим из нее через другое ребро (ни то, ни другое больше в цикл не входят); следовательно, число ребер, инцидентных вершине **v**, должно быть равно удвоенному числу наших посещений вершины **v** при прохождении эйлерова цикла, т.е. должно быть равно четному числу.

Чтобы доказать достаточность, воспользуемся методом индукции по количеству ребер. Это утверждение заведомо выполняется для графов, у которых нет ребер. Рассмотрим любой связный граф, в котором содержится более одного ребра, при этом степени всех вершин четные. Предположим, что начиная с произвольной вершины, мы продвигаемся по любому ребру, после чего его удаляем. Мы продолжаем этот процесс до тех пор, пока не окажемся в вершине, у которой нет ребер. Этот процесс должен когда-нибудь завершиться, поскольку мы удаляем по одному ребру на каждом шаге, однако каким может стать результат этой процедуры? На рис. 17.22 приводятся примеры. Нам сразу становится ясно, что этот процесс должен закончиться на вершине **v** тогда и только тогда, когда ее степенью было число нечетное в момент, когда этот процесс начинался.

Одна из возможностей состоит в том, что мы пройдем по всему циклу; если это так, то доказательство завершено. В противном случае, все вершины графа, который при этом остался, имеют четные степени, но тогда граф может оказаться несвязным. Однако в соответствии с индуктивным предположением, каждая его связная компонен-

та содержит эйлеров цикл. Более того, только что удаленный циклический путь связывает эти циклы в эйлеров цикл исходного графа: пройдите по этому циклическому пути, делая обход эйлерова цикла каждой соединенной с ним компоненты. Каждый такой обход по существу представляет собой эйлеров цикл, заканчивающийся в вершине, с которой он начинался. Обратите внимание на тот факт, что каждый такой обход многократно касается удаленного цикла (см. упражнение 17.99). В каждом таком случае мы делаем подобный обход только один раз (например, когда мы впервые с ним сталкиваемся). ■

Следствие. *Граф содержит эйлеров цикл тогда и только тогда, когда он связный и в точности две его вершины имеют нечетную степень.*

Доказательство: Эта формулировка эквивалентна формулировке свойства 17.4 для графа, построенного путем добавления ребра, соединяющих две вершины нечетной степени (на концах пути). ■

Отсюда следует, например, что никто не может пройти через все мосты Кенигсберга так, чтобы не пройти по одному мосту дважды, поскольку в соответствующем графе все четыре составляющие его вершины обладают нечетными степенями (см. рис. 17.21).

В соответствии с изложенным в разделе 17.5, мы можем найти все степени вершин за время, пропорциональное E для представления графа в виде списков смежных вершин или в виде множества ребер, либо за время, пропорциональное V^2 для представления графа в виде матрицы смежности, либо мы можем поддерживать вектор, индексированный именами вершин, в котором степени вершин являются частью представления графа (см. упражнение 17.42). При наличии такого вектора мы можем проверить, выполняется ли свойство 17.4 за время, пропорциональное V. Программа 17.18 реализует эту стратегию и показывает, что проверка заданного графа на наличие в нем эйлерового цикла представляет собой достаточно простую вычислительную задачу. Этот факт важен в силу того, что до сих пор мы не были уверены в том, что эта задача проще, чем определение гамильтонова пути в заданном графе.

Теперь предположим, что мы и в самом деле хотим найти эйлеров цикл. Мы идем по тонкому льду, поскольку прямая рекурсивная реализация (поиск пути за счет проверки ребра, после которой выполняется рекурсивный вызов с целью определения пути в остальной части графа) обеспечивает ту же производительность, пропорциональную факториалу числа вершин, что и программа 17.17. Мы не можем смириться с такой производительностью, по-

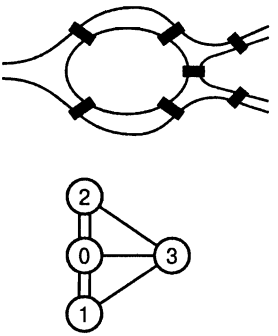


РИСУНОК 17.21. МОСТЫ КЕНИГСБЕРГА

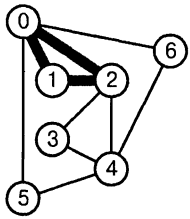
*Широко известная задача, изучавшаяся еще Эйлером, связана с городом Кенигсберг, в котором в устье реки Преголь расположены острова, соединенные с берегами семью мостами. Существует ли путь, который позволяет во время непрерывной прогулки по городу, обойти все семь мостов и ни на одном из них не побывать дважды? Если мы обозначим остров через **0**, берега реки через **1** и **2** еще один остров через **3**, а также определим ребра, соответствующие каждому мосту, мы получим мультиграф, показанный в нижней части рисунка. Суть задачи заключается в том, чтобы найти такой путь, который использует каждое ребро в точности один раз.*

скольку довольно легко проверить существование такого пути, поэтому мы намерены отыскать более приемлемый алгоритм. Есть возможность избежать зависимости от факториала числа вершин в условиях фиксированной стоимости проверки, можно ли использовать то или иное ребро (в отличие от неизвестной стоимости в условиях рекурсивных вызовов), однако мы оставляем этот подход на самостоятельную проработку (см. упражнения 17.96 и 17.97).

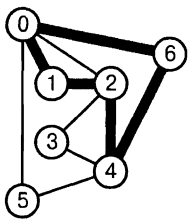
Другой подход вытекает из доказательства свойства 17.4. Проследуем по циклическому пути, стирая все ребра и помещая в стек все вершины, которые нам встретятся, так что (*i*) мы можем проследить свой путь, распечатав его ребра, и (*ii*) проверить каждую вершину на наличие боковых путей (который может быть включен в главный путь). Этот процесс иллюстрируется рисунком 17.23.

Сначала программа добавляет в искомый цикл ребро **0-1** и удаляет его из списков смежных вершин (из двух мест) (диаграмма в верхнем левом углу, списки слева от диаграммы). Затем она добавляет ребро **1-2** в искомый цикл тем же способом (диаграмма слева, вторая сверху). Далее, она поворачивает назад в **0**, но при этом продолжает строить цикл **0-5-4-6-0**, возвращаясь в вершину **0**, при этом ребер, инцидентных вершине **0**, не остается (справа, вторая диаграмм сверху). Затем она выталкивает из стека изолированные вершины **0** и **6**, так что вверху стека остается вершина **4**, и начинает цикл с вершины **4** (справа, третья диаграмма сверху), проходит через вершины **3, 2** и возвращается в **4**, вследствие чего из стека выталкиваются все теперь уже изолированные вершины **4, 2, 3** и т.д. Последовательность вершин, вытолкнутых из стека, определяет эйлеров цикл **0-6-4-2-3-4-5-0-2-1-0** для всего графа.

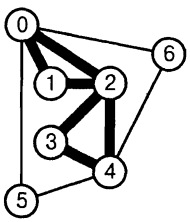
Программа 17.19 представляет собой реализацию изложенного в этих строках. Она предполагает, что эйлеров цикл существует, при этом она уничтожает локальную копию графа; таким образом, важно, чтобы в класс **Graph**, который использует рассматриваемая программа, входил конструктор копирования, который создает полностью автономную копию графа. Программный код достаточно сложен, новичкам полезно освоить алгоритмы обработки графов, рассмотренные в нескольких следующих главах, прежде чем предпринимать попытку разобраться в нем. Мы включили ее в этот раздел с тем, чтобы показать, что хорошие алгоритмы и умелая их реализация позволяют исключительно эффективно решать некоторые задачи обработки графов.



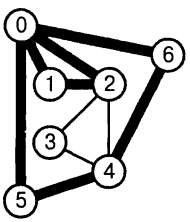
1-0-2-1



6-0-1-2-4-6



2-0-1-2-3-4-2



0-6-4-5-0-2-1-0

РИСУНОК 17.22. ЧАСТИЧНЫЕ ЦИКЛЫ

Путь вдоль ребер, начинающийся в любой из вершин графа, в котором имеется эйлеров цикл, всегда приводит нас в ту же вершину, как показано в находящихся на рисунке примерах. Цикл не обязательно проходит через все ребра в графе.

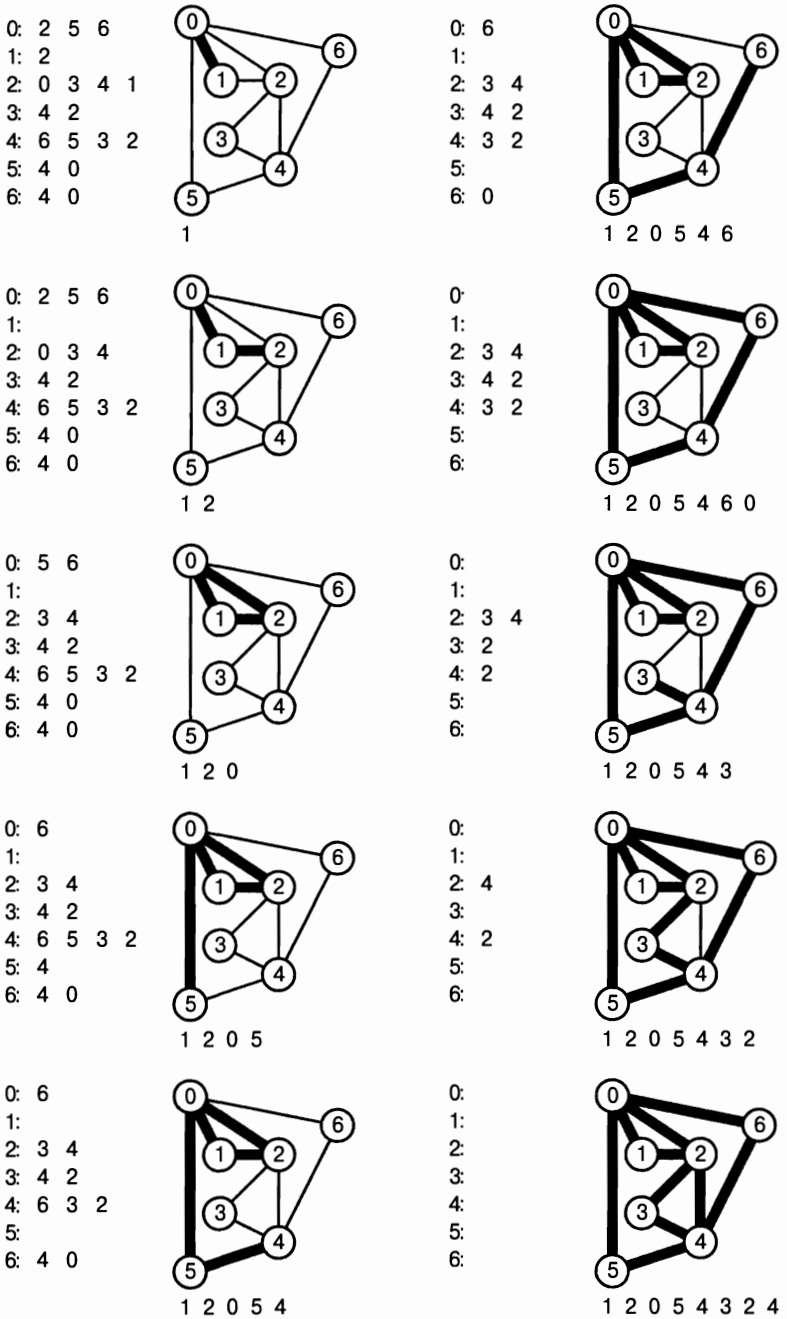


РИСУНОК 17.23. ПОИСК ЭЙЛЕРОВА ПУТИ МЕТОДОМ УДАЛЕНИЯ ЦИКЛОВ

*Этот рисунок служит иллюстрацией того, как программа 17.19 находит эйлеров цикл из вершины **0** в вершину **0** на простом графе. В этот цикл входят жирные черные ребра, содержимое стека показано под каждой диаграммой, список смежных вершин, не входящих в искомый цикл, находится слева от диаграммы, а списки смежных вершин ребер, не попавших в цикл, показаны слева от каждой диаграммы.*

Программа 17.18. Существование эйлерова цикла

Имея в своем распоряжении этот класс, клиентские программы могут проверить существование эйлерова цикла в графе. Вершины **v** и **w** рассматриваются как приватные элементы данных, благодаря чему клиентские программы могут воспользоваться функцией-элементом **show** (которая использует приватную функцию-элемент **tour**) для печати пути (см. программу 17.19).

В основу этого теста, использующего программу 17.11, положено следствие свойства 17.4. На его выполнение затрачивается время, пропорциональное V, причем сюда не входит время, затрачиваемое на предварительную обработку, в рамках которой производится проверка связности и построение таблицы степеней вершин в классе **DEGREE**.

```
template <class Graph> class ePATH
{ Graph G;
   int v, w;
   bool found;
   STACK <int> S;
   int tour(int v);
public:
   ePATH(const Graph &G, int v, int w) :
     G(G), v(v), w(w)
     { DEGREE<Graph> deg(G);
         int t = deg[v] + deg[w];
         if ((t % 2) != 0) { found = false; return; }
         for (t = 0; t < G.V(); t++)
           if ((t != v) && (t != w))
             if ((deg[t] % 2) != 0)
                { found = false; return; }
         found = true;
     }
   bool exists() const
     { return found; }
   void show();
};
```

Программа 17.19. Поиск эйлерова пути с линейным временем выполнения

Данная реализация функции **show** для класса из программы 17.18 выводит на печать эйлеров путь между двумя заданными вершинами, если таковой имеется. В отличие от других наших реализаций, в основу рассматриваемого программного кода положена реализация АТД **Graph**, в которой имеется соответствующий конструктор копирования, поскольку он строит копию графа и уничтожает эту копию, удаляя ребра из графа во время печати этого пути. При условии, что реализация функции **remove** выполняется за постоянное время (см. упражнение 17.46), функция **show** выполняется за линейное время. Приватная функция-элемент **tour** проходит по ребрам, удаляет их из цикла и помещает вершины в стек, чтобы выявить наличие боковых циклов (*см. рассуждения по тексту раздела*). Главный цикл продолжает вызывать функцию **tour** до тех пор, пока существуют боковые циклы.

```
template <class Graph>
int ePATH<Graph>::tour(int v)
 {
  while (true)
    { typename Graph::adjIterator A(G, v);
        int w = A.beg(); if (A.end()) break;
        S.push(v);
```

```
            G.remove(Edge(v, w));
            v = w;
        }
      return v;
    }
template <class Graph>
void ePATH<Graph>::show()
    {
        if (!found) return;
        while (tour(v) == v && !S.empty())
          { v = S.pop(); cout << "-" << v; }
        cout << endl;
    }
```

Свойство 17.5. *Мы можем обнаружить в графе эйлеров цикл, если таковой существует, за линейное время.*

Полное доказательство этого свойства методом индукции мы оставляем на самостоятельную проработку (см. упражнение 17.100). По существу, после первого вызова функции **path** в стеке содержится путь от **v** до **w**, и оставшаяся часть графа (после удаления изолированных вершин) состоит из связных компонент меньших размеров (имеющих, по меньшей мере, одну общую вершину с найденным на текущий момент путем), которые также содержат эйлеровы циклы. Мы выталкиваем изолированные вершины из стека и применяем функцию **path** для поиска тем же способом эйлеровых циклов, которые содержат неизолированные вершины. Каждое ребро графа заталкивается в стек (и выталкивается из него) в точности один раз, в силу чего общее время выполнения пропорционально E. ■

Несмотря на то что эйлеровы циклы допускают систематический обход всех ребер и вершин, мы довольно редко используем их на практике в силу того обстоятельства, что лишь немногие графы содержат такие циклы. Вместо этого для исследования графов мы обычно применяем метод поиска в глубину, который подробно рассматривается в главе 18. В самом деле, как мы убедимся позже, поиск в глубину на неориентированном графе эквивалентен вычислению *двухпроходного эйлерова цикла* (*two way Euler tour*) — пути, который проходит по каждому ребру в точности *два раза*, по одному в каждом направлении.

Подводя итог сказанному в этом разделе, отметим, что поиск простых путей в графах не представляет трудностей, что еще легче определить, сможем ли мы обойти через все *ребра* крупного графа, не проходя ни по одному из них дважды (достаточно всего лишь убедиться в том, что степени всех вершин суть четные числа), и что даже существует "умный" алгоритм, способный найти такой цикл, но в то же время практически невозможно узнать, можем ли мы обойти все *вершины* графа, не посетив ни одну из них дважды. В нашем распоряжении имеются рекурсивные решения всех этих задач, однако высокая вероятность экспоненциального возрастания времени выполнения делает эти решения практически бесполезными. Другие решения позволяют получить быстродействующие алгоритмы, удобные для практического применения.

Такой разброс похожих на первый взгляд задач по трудности решения, иллюстрацией которого могут послужить эти примеры, характерен для обработки графов и является фундаментальным свойством в теории вычислений. Из краткого анализа, проводимого в

разделе 17.8, и более подробных исследований, выполняемых в части 8, следует, что мы должны признать то, что представляется нам непреодолимым барьером между задачами, для решения которых, по-видимому, требуется экспоненциальное время (такие как задача поиска гамильтонова пути и многие другие реальные задачи), и задачами, о которых нам известно, что алгоритмы их решения гарантированно выполняются за полиномиальное время (такие как задача поиска эвклидова пути и многие другие практические задачи). В данной книге основной нашей целью является разработка эффективных алгоритмов решения задач второго из указанных выше классов.

Упражнения

▷ **17.85**. В стиле упражнения 17.17 покажите трассировку рекурсивных вызовов (и пропущенные вершины), когда программа 17.16 производит поиск пути из вершины **0** в вершину **5** в графе

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

17.86. В соответствии с изложенным в тексте, внесите изменения в рекурсивную функцию, используемую в программе 17.16, с тем, чтобы она могла распечатывать трассировки, аналогичные показанным на рис. 17.17, используя для этой цели глобальную переменную.

17.87. Выполните упражнение 17.86, добавив аргумент рекурсивной функции, позволяющий отслеживать глубину рекурсии.

○ **17.88**. Используя метод, описанный в тексте, постройте реализацию класса **sPATH**, обеспечивающего выполнение общедоступной функции-элемента, которая вызывает клиентскую функцию для каждого ребра на пути из **v** в **w**, если такой путь существует.

○ **17.89**. Внесите такие изменения в программу 17.16, которые позволили бы ей принимать третий аргумент **d** и проверять существование пути, соединяющего вершины **u** и **v**, длина которого больше **d**. В частности, значение **search(v, v, 2)** должно быть ненулевым в том и только том случае, когда **v** содержится в некотором цикле.

• **17.90**. Проведите эксперименты с целью определения эмпирическим путем вероятности того, что программа 17.16 найдет путь между двумя наудачу выбранными вершинами в различных графах (см. упражнения 17.63–17.76) и вычислите среднюю длину пути, найденного для различных типов графов.

○ **17.91**. Рассмотрим графы, заданные следующими четырьмя наборами ребер:

0-1 0-2 0-3 1-3 1-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
0-1 0-2 0-3 1-3 0-3 2-5 5-6 3-6 4-7 4-8 5-8 5-9 6-7 6-9 8-8
0-1 1-2 1-3 0-3 0-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8
4-1 7-9 6-2 7-3 5-0 0-2 0-8 1-6 3-9 6-3 2-8 1-5 9-8 4-5 4-7

Какие из этих графов содержат эйлеровы циклы? Какие из них содержат гамильтоновы циклы?

○ **17.92**. Сформулируйте такие необходимые и достаточные условия, что ориентированный граф, удовлетворяющий этим условиям, содержит (ориентированный) эйлеров цикл.

17.93. Докажите, что каждый связный неориентированный граф содержит двухпроходный эйлеров цикл.

17.94. Внесите изменение в доказательство свойства 17.4 с тем, чтобы оно работало и в случае графов с параллельными ребрами и петлями.

▷ **17.95**. Покажите, что если добавить еще один мост, то задача о Кенигсбергских мостах получит решение.

• **17.96**. Докажите, что в связном графе имеется эйлеров путь из **v** в **w** только в том случае, когда он содержит ребро, инцидентное **v**, удаление которого не нарушает связности графа (если не учитывать возможности, что после этого вершина **v** станет изолированной).

• **17.97**. Воспользуйтесь упражнением 17.96, чтобы разработать эффективный рекурсивный метод поиска эйлерова цикла в графе, в котором есть такой цикл. Помимо базовых функций АТД графа, вы можете воспользоваться классами, рассматриваемыми в данной главе, которые определяют степени вершин (см. программу 17.11) и проверяют, существует ли путь между двумя заданными вершинами (см. программу 17.16). Постройте реализацию и протестируйте полученную программу как на разреженных, так и на насыщенных графах.

▷ **17.98**. Приведите пример, в котором граф, оставшийся после первого обращения к функции **path** из программы 17.19, будет несвязным (в графе, содержащем эйлеров цикл).

▷ **17.99**. Опишите, какие изменения следует внести в программу 17.19, с расчетом, чтобы ее можно было использовать для определения за линейное время, имеется ли в заданном графе эйлеров цикл.

17.100. Дайте полное доказательства методом индукции утверждения, что алгоритм поиска эйлерова пути, выполняемый за линейное время, который описан в тексте книги и реализован в программе 17.19, правильно находит эйлеров цикл.

○ **17.101**. Найдите число графов с V вершинами, которые содержат эйлеров цикл, для максимального числа V, для которого вы способны выполнить реальные вычисления.

• **17.102**. Выполните эксперименты для различных графов с тем, чтобы эмпирическим путем определить среднюю длину пути, найденного при первом вызове функции **path** в программе 17.19 (см. упражнения 17.63–17.76). Вычислите вероятность того, что этот путь является циклом.

○ **17.103**. Напишите программу, которая вычисляет последовательность из $2^n + n - 1$ бит, в которой никакие две последовательности из n следующих подряд битов не совпадают. (Например, для $n = 3$ последовательность 0001110100 обладает таким свойством.) *Примечание*: Найти эйлеров цикл в орграфе Де-Бруйна.

▷ **17.104**. Покажите в стиле рис. 17.19 трассу рекурсивных вызовов (и пропущенные вершины), когда программа 17.16 находит гамильтонов цикл в графе

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

○ **17.105**. Внесите в программу 17.17 изменения с тем, чтобы она могла распечатать гамильтонов цикл, если она его найдет.

• **17.106**. Найдите гамильтонов цикл в графе

1-2 2-5 4-2 2-6 0-8 3-0 1-3 3-6 1-0 1-4 4-0 4-6 6-5 2-6
6-9 9-0 3-1 4-3 9-2 4-9 6-9 7-9 5-0 9-7 7-3 4-5 0-5 7-8,

либо показать, что такового не существует.

●● **17.107**. Определите, сколько графов с V вершинами содержится в гамильтоновом цикле для максимально большого значения V, для которого еще можно выполнить реальные вычисления.

17.8 Задачи обработки графов

Вооружившись базовыми инструментальными средствами, которые были разработаны в настоящей главе, мы рассмотрим в главах 18–22 самые разнообразные алгоритмы решения задач обработки графов. Эти алгоритмы относятся к категории фундаментальных и могут оказаться полезными во многих приложениях, но, тем не менее, их можно рассматривать как введение в область алгоритмов на графах. Разработано множество интересных и полезных алгоритмов, которые выходят за рамки данной книги, исследовано множество интереснейших задач, для решения которых все еще не удалось найти хороших алгоритмов.

Как и в случае любой другой предметной области, первая серьезная проблема, с которой нам приходится сталкиваться при решении новой задачи обработки графов, касается определения, насколько трудной для решения является эта новая задача. Применительно к обработке графов, это решение может оказаться намного более трудным, чем мы можем это себе представить, даже для задач, решение которых, на первый взгляд, не представляет трудностей. Более того, наша интуиция часто оказывается бессильной и не может отличить легкой задачи от трудной или от нерешенной на данный момент. В этом разделе мы кратко опишем классические задачи и проверим, что мы о них знаем.

Имея перед собой новую задачу обработки графов, с какого вида трудностями нам придется столкнуться при разработке реализации ее решения? Неприглядная правда состоит в том, что не существует подходящего метода, чтобы ответить на этот вопрос применительно к любой задаче, с которыми нам приходится сталкиваться, в то же время мы способны дать общее описание сложности решения различных классических задач обработки графов. В этом смысле мы разобьем эти задачи примерно по следующим категориям сложности их решения:

- Легкие
- Поддаются решению
- Трудно решаемые
- Неизвестно, существует ли решение.

Такая классификация предназначена для того, чтобы получать информацию о каждой такой категории и чтобы иметь представление о текущем уровне знаний в области алгоритмов на графах.

Как показывает эта терминология, основная причина, побудившая ввести подобную классификацию задач, заключается в том, что существует множество задач на графах, подобных задаче поиска гамильтонова цикла, при этом никто не знает, можно ли найти их эффективное решение. В конечном итоге мы узнаем (в части 8), как наполнить это заявление содержанием в точном техническом смысле; однако на данном этапе мы, по меньшей мере, получаем предостережение о том, с какими серьезными препятствиями нам придется столкнуться во время написания программ, решающих эти задачи.

Мы отложим подробное описание многих задач обработки графов до дальнейших разделов данной книги. Здесь мы ограничимся краткими и понятными формулировками с тем, чтобы ознакомить читателя с проблемой классификации задач обработки графов по трудности их решения.

Легкая задача обработки графа – это задача, которая может быть решена путем применения некоторых видов компактных, элегантных и эффективных программ, к которым мы уже успели привыкнуть на протяжении глав 1–4. Достаточно часто для выполнения этих программ требуется в худшем случае линейные затраты времени, либо зависимость этих затрат ограничивается полиномами низких степеней по числу вершин или числу ребер. В общем случае, как это имело место во многих других предметных областях, мы можем установить, что проблема относится к числу легких, путем разработки решения "в лоб", которое, будучи слишком медленным для графов крупных размеров, допустимо для графов небольших, а иногда и средних размеров. Затем, зная, что задача имеет легкое решение, мы предпринимаем попытки найти эффективные решения, которыми мы смогли воспользоваться на практике, и стремимся выбрать из них наилучшее. Задачу поиска эйлерова цикла, которую мы рассматривали в разделе 17.7, может служить наиболее ярким примером легких задач, в главах 18–22 мы рассмотрим множество других таких задач, и в первую очередь следующие.

Простая связность. Обладает ли заданный граф свойством связности? Иначе говоря, существуют ли пути, соединяющие каждую пару его вершин? Существует ли цикл в графе или он представляет собой лес? Содержатся ли в цикле две заданные вершины? Впервые мы столкнулись с необходимостью найти ответ на эти основные вопросы, касающиеся обработки графов, в главе 1. Мы будем искать решения указанных задач в главе 18. Некоторые из них легко решаются при линейных затратах времени; чтобы решить другие задачи за линейное время, требуется разработка изощренных алгоритмов, которые, в свою очередь, требуют серьезных исследований.

Сильная связность в орграфах. Существует ли ориентированная цепь, соединяющая каждую пару вершин орграфа? Соединены ли две заданные вершины графа ориентированной цепью в обоих направлениях (входит ли они в какой-либо направленный цикл)? Реализация эффективного решения этих задач намного сложнее, чем соответствующая задача простой связности в неориентированных графах, на их изучение отводится большая часть главы 19. Несмотря на все хитроумные приемы, применяемые при их решения, мы относим эти проблемы к категории легких, поскольку мы можем написать компактную эффективную и полезную реализацию.

Транзитивное замыкание. Какое множество вершин может быть получено, если следовать по направлению ориентированных графов из каждой вершины орграфа? Эта задача имеет прямое отношение к задаче сильной связности и другим фундаментальным вычислительным задачам. В главе 19 мы будем изучать классические решения, которые сводятся всего лишь к нескольким строкам программного кода.

Минимальное остовное дерево. Во взвешенном графе необходимо найти множество ребер с минимальным весом, которые соединяет все вершины. Это одна из старейших и хорошо изученных задач обработки графов; глава 20 целиком отводится для изучения различных классических алгоритмов ее решения. Исследователи продолжают поиски более быстродействующих алгоритмов решения этой задачи.

Поиск кратчайших путей из одного истока. Каким будет кратчайший путь, соединяющий заданную вершину v с каждой другой вершиной во взвешенном орграфе (сети)? Вся глава 21 посвящена изучению этой задачи, которая исключительно важна для многочисленных применений. Эту задачу никоим образом нельзя отнести к категории легко решаемых, если веса могут принимать отрицательные значения.

Задача обработки графов, принадлежащая к категории поддающейся решению, – это задача, алгоритм решения которой известен, а требования к которой в смысле затрат времени и пространства памяти на ее реализацию ограничиваются полиномиальной функцией от размеров графа ($V + E$). Все легкие задачи поддаются решению, однако мы проводим различия между ними, поскольку для многих задач, поддающихся решению, характерно то, что разработка эффективных программ их решения, применение которых в оправдано в практических целях, представляет собой исключительно трудную, если не невозможную, проблему. Решения могут оказаться слишком сложным, чтобы приводить их в данной книге, поскольку их реализации могут содержать сотни и даже тысячи строк программного кода. Далее следуют два примера наиболее актуальных задач этого класса.

Планарность. Можно ли начертить заданный граф таким образом, чтобы никакие линии, представляющие ребра, не пересекались? Мы свободны поместить вершины в любое место, так что мы можем решить эту проблему для множества одних графов, но ее нельзя решить для множества других графов. Замечательный классический результат, известный как *теорема Куратовского* (*Kuratovski's theorem*), позволяет легко проверить, является ли граф планарным (плоским). Эта теорема утверждает: графы, которые не могут быть представлены на чертеже без пересечения ребер, суть графы, содержащие некоторый подграф, который после удаления из него вершин степени 2, становится изоморфным одному из графов, изображенных на рис. 17.24. Бесхитростная реализация этого теста, даже если не принимать во внимание вершин степени 2, работает недопустимо медленно на крупных графах (см. упражнение 17.110), но в 1974 г. Роберту Тарьяну (R.Tarjan) удалось получить оригинальный (но в то же время довольно замысловатый) алгоритм, способный решать эту задачу за линейное время с использованием схемы поиска в глубину, которая является расширением схем, рассматриваемых в главе 18. Алгоритм Тарьяна не обязательно позволяет получить чертеж, пригодный для применения в практических целях, он просто констатирует тот факт, что такой чертеж существует. Как уже было отмечено в разделе 17.1, построение наглядного чертежа графа для приложений, в котором вершины графа не обязательно соответствуют реалиям внешнего мира, превращается в довольно сложную исследовательскую задачу.

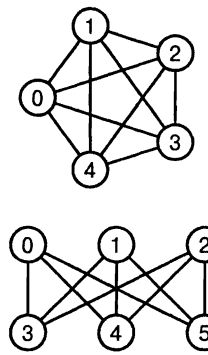


РИСУНОК 17.24. ЗАПРЕЩЕННЫЕ ПОДГРАФЫ ПЛАНАРНЫХ ГРАФОВ

Ни один из изображенных здесь графов нельзя начертить на плоскости без того, чтобы его ребра не пересекались, это утверждение справедливо для всех графов, которые содержат любой из этих графов в качестве подграфа (после того, как мы удалим вершины степени 2); однако все остальные графы допускают подобное отображение на плоскости.

Сочетания. Каким является наибольшее подмножество ребер графа, обладающего тем свойством, что никакие два ребра из подмножества не связаны с одной и той же

вершиной? Известно, что эта классическая задача имеет решение, при этом она решается за время, пропорциональное полиномиальной функции от V и E, однако исследователям никак не удается получить быстродействующий алгоритм, пригодный для работы с крупными графами. Эту задачу проще решить при наличии разнообразных ограничений. Например, задача поиска соответствия студентов вакантным должностям в различных общественных организациях есть задача *двудольного сочетания* (*bipartite matching*): мы имеем два различных типа вершин (студенты и организации), при этом нас интересуют только те ребра, которые соединяют вершину одного типа с вершиной другого типа. Решение этой задачи можно найти в главе 22.

Решения некоторых задач, поддающихся решению, никогда не были представлены в виде программ либо требуют для своего выполнения такое большое время, что вопрос об их практическом применении отпадает сам собой. Приводимый далее пример принадлежит к числу задач этого типа. Он также демонстрирует капризный характер математической реальности обработки графов.

Четные циклы в орграфах. Имеются ли в заданном орграфе цикл четной длины? На первый взгляд кажется, что на этот вопрос нетрудно ответить, поскольку нетрудно ответить на этот вопрос в случае неориентированных графов (см. раздел 18.4), равно как и на вопрос, имеется ли в орграфе цикл нечетной длины. Тем не менее, в течение многих лет мы никак не можем изучить эту задачу настолько, чтобы хотя бы ответить на вопрос, существует или нет алгоритм ее решения (*см. раздел ссылок*). Теорема, устанавливающая существование эффективного алгоритма, была доказана в 1999 г., однако метод доказательства настолько сложен, что никакой математик или программист не возьмется за ее реализацию.

Одна из основных тем, рассматриваемых в главе 22, заключается в том, что многие задачи на графах, поддающиеся решению, решаются наилучшим образом за счет применения алгоритмов, которые могут решать весь класс этих задач в общей постановке. *Алгоритмы поиска кратчайшего маршрута* (*shortest path algorithm*), рассматриваемые в главе 21, *алгоритмы на транспортных сетях* (*network flow algorithm*), исследуемые в главе 22, а также мощный *сетевой симплексный алгоритм* (*network simplex algorithm*), рассматриваемый в главе 22, способны решать многие задачи на графах, которые в противном случае превращаются в трудно преодолимые проблемы. В числе таких задач отметим следующие.

Задача распределения. Эта задача, известная еще как задача *двудольного взвешенного сочетания* (*bipartite weighed matching*), заключается в том, чтобы найти в двудольном графе совершенное сочетание с минимальным весом. Она легко решается за счет применения алгоритмов, разработанных для транспортных сетей. Известны специальные методы, которые решают данную проблему непосредственно, но при ближайшем рассмотрении было обнаружено их большое сходство с решениями, обеспечиваемыми алгоритмами транспортных сетей.

Общая связность. Какое минимальное число ребер нужно удалить из графа, чтобы он распался на две несвязных части (реберная связность)? Каково минимальное число вершин, удаление которых разобьет граф на две несвязных части? Как мы узнаем из главы 22, эти две задачи, будучи трудно разрешимыми непосредственно, достаточно просто решаются при помощи алгоритмов транспортных сетей.

Задача почтальона. На заданном графе необходимо найти цикл с минимальным числом ребер, в котором каждое ребро графа используется, *по меньшей мере*, раз (при этом разрешается использовать ребра многократно). Эта задача намного сложнее, чем задача поиска эйлерова пути, в то же время она намного легче, чем задача поиска гамильтонова пути.

Этап перехода от убеждения в том, что задача имеет решение, до того, как будет получено программное обеспечение, позволяющее решать эту задач на практике, может оказаться весьма продолжительным. С одной стороны, доказав, что задача допускает реализацию, исследователи не могут устоять перед соблазном не останавливаться подробно на многочисленных деталях, с которыми приходится иметь дело при разработке реализации; с другой стороны, они должны уделить внимание различным возможным ситуациям, которые в практических условиях не обязательно могут возникать. Этот разрыв между теорией и практикой ощущается особенно остро, когда исследования касаются алгоритмов на графах, что можно объяснить тем, что математические исследования заполнены глубокими результатами, описывающими ошеломляющее разнообразие структурных свойств, которые мы должны учитывать при обработке графов, а также тем, что связь между полученными результатами и свойствами графа, возникающая на практике, мало понятна. Разработка общих схем, таких как, например, сетевой симплексный алгоритм, представляет собой исключительно эффективный подход к решению подобного рода задач.

Задача обработки графов, *неподдающаяся решению (intractable)*, – это задача, для которой не известен алгоритм, гарантирующий ее решение за приемлемый промежуток времени. Для многих таких задач характерно то, что для ее решения мы можем использовать метод решения "в лоб", в рамках которого можно попытаться воспользоваться любыми возможностями, чтобы найти решение путем вычислений, а неподдающимися решению мы их считаем в силу того факта, что таких возможностей слишком много. Это очень широкий класс задач, он включает в себя многие из важных задач, решение которых мы хотели бы знать. Для описания задач этого класса применяется термин *NP-трудный* (*NP-hard*, NP – non-linear polynomial – класс комбинаторных задач с нелинейной полиномиальной оценкой числа итераций). Многие специалисты убеждены в том, что эффективных алгоритмов решения этих задач не существует. В части 8 мы более подробно рассмотрим, что послужило причиной для такого рода убеждений и что стало основой для употребления этого термина. Задача поиска гамильтонова цикла, которую мы рассматривали в разделе 17.7, представляет собой прекрасный пример NP-трудной задачи обработки графа, равно как и указанные в приводимом ниже списке.

Самый длинный путь. Какой путь, соединяющий две заданных вершины графа, является самым длинным? Несмотря на все сходство этой задачи с задачей поиска кратчайшего пути, эта задача, тем не менее, есть версия задачи поиска гамильтонова пути, и посему, NP-трудная.

Задача окраски. Существует ли такой способ закрашивания каждой вершины графа одним из k цветов, чтобы ни одно ребро не соединяло вершин одного и того же цвета? Эта классическая задача легко решается для $k = 2$ (см. раздел 18.4), но она относится к числу NP-трудных задач уже при $k = 3$.

Множество независимых вершин. Каким будут размеры наибольшего подмножества вершин графа, обладающих тем свойством, что никакие две из них не соединены ребром? Точно так же, как и в случае сопоставления задач нахождения эйлерова и гамильтонова путей, мы убеждаемся, что несмотря на кажущееся подобие проблеме сочетаний, которая решается за полиномиальное время, эта задача принадлежит к числу NP-трудных задач.

Клика. Каков размер максимальной клики (полного подграфа) в заданном графе? Эта задача обобщает часть задачи планарности, ибо если наибольшая клика состоит из более четырех узлов, граф не может быть планарным.

Эти задачи сформулированы как задачи существования – мы должны определить, существует или не существует подграф конкретного типа. Целью ряда других задач является определение размера наибольшего подграфа конкретного типа, что мы можем сделать, сводя задачу существования к проверке существования подграфа размера k, который обладает интересующим нас свойством, затем используем метод двоичного поиска для выявления наибольшего из них. На практике, однако, мы, по существу, часто стремимся отыскать полное решение, которое в общем случае найти гораздо труднее. Например, известная *теорема четырех красок* (*four color theorem*) утверждает, что можно воспользоваться четырьмя цветами для раскраски всех вершин планарного графа таким образом, что ни одно ребро не будет соединять две вершины одного и того же цвета. Однако теорема ничего не говорит нам о том, как это сделать для конкретного плоского графа: наше знание о том, что такая раскраска существует, ничем не может нам помочь в поиске полного решения этой задачи. Другой известный пример – это задача *коммивояжера* (*traveling salesperson*), по условиям которой требуется определить минимальный путь обхода вершин взвешенного графа. Эта задача относится к тому же классу задач, что и задача поиска гамильтонова цикла, при этом она нисколько не легче: если мы не можем найти эффективного решения задачи поиска гамильтонова пути, мы не можем рассчитывать на то, что найдем решение задачи коммивояжера. Как правило, сталкиваясь с трудными задачами, мы работаем с простейшими ее версиями, которые в состоянии решить. Задачи существования по духу соответствуют этому правилу, но они играют важную роль в теории, как мы убедимся, изучив материал части 8.

Перечисленные выше задачи – это всего лишь небольшая часть из нескольких тысяч NP-трудных задач, которые были выявлены. Они возникают во всех типах вычислительных приложений, как мы узнаем из части 8. Особенно много таких задач возникает при обработке графов, так что мы должны учитывать их существование на протяжении всей книги.

Обратите внимание, что мы настаиваем на том, чтобы предлагаемые нами алгоритмы гарантировали эффективность для худшего случая. Возможно, вместо этого мы должны ориентироваться на алгоритмы, которые эффективно работают на типовых входных данных (не обязательно для худшего случая). Аналогично, многие задачи требуют *оптимизации*. Возможно, мы должны преследовать цель найти длинный путь (не обязательно самый длинный) или большую клику (но не обязательно максимальную). С точки зрения обработки графов, может быть, легко найти хороший ответ для графов, которые встречаются на практике, и нас, скорее всего, не интересует алгоритм, который может найти оптимальное решение на некотором выдуманном графе, с которым никогда не доведется

иметь дела. В самом деле, для решаемых задач могут быть применены прямые или универсальные алгоритмы, подобные программе 17.17, которые, несмотря на то, что время их выполнения в худшем случае экспоненциально, позволяют быстро найти решение (или приемлемое приближение) для многих конкретных примеров соответствующих реальных задач. Мы можем отказаться от использования программы, которая на некоторых входных данных может дать неверные результаты или аварийно завершиться, однако временами мы прибегаем к помощи программ, для которых характерно экспоненциальное время выполнения на некоторых вводах. Мы будем изучать подобного рода ситуации в части 8.

Результаты многих исследований показывают, что многие трудно решаемые задачи так и остаются трудно решаемыми, даже если ослабить некоторые ограничения. Более того, существует множество практических задач, которые мы не можем решить, поскольку ничего не известно о существовании достаточно быстродействующего алгоритма. В этой части книги мы будем относить эти задачи, при столкновении в ними, к категории NP-трудных задач, и будем трактовать этот термин, по меньшей мере, как указание на то, что мы не надеемся отыскать эффективный алгоритм их решения и что не будем предпринимать попыток найти их решение без применения современных технологий, подобных тем, что рассматриваются в части 8 (за исключением, возможно, применения методов решения "в лоб" для решения небольших задач).

Существуют задачи обработки графов, о которых не известно, насколько они трудны для решения (их трудность *неизвестна*). Неизвестно, существует ли алгоритм, обеспечивающий их эффективное решение, неизвестно также, принадлежат ли они к категории NP-трудных задач. Вполне возможно, что по мере того, как наши знания алгоритмов обработки графов и свойств графов увеличиваются, окажется, что некоторые из этих задач окажутся в категории решаемых и даже легких задач. Приводимая ниже важная естественная задача, с которой нам уже приходилось сталкиваться (см. рис. 17.2), является наиболее известной задачей такого класса.

Изоморфизм графов. Можно ли сделать два графа идентичными, переименовав соответствующим образом их вершины? Известно, что существуют эффективные алгоритмы решения этой задачи для специальных типов графов, но вопрос о трудности решения задачи для общего случая остается открытым.

Количество важных задач, для которых свойственная им трудность решения неизвестна, небольшое и не идет ни в какое сравнение с другими категориями задач, которые мы рассмотрели выше, благодаря интенсивным исследованиям, проводившимся в этой области за последние несколько десятилетий. Некоторые задачи этого класса, такие как изоморфизм классов, представляют собой огромный практический интерес; другие задачи этого класса получили известность главным образом в силу того обстоятельства, что они не поддаются классификации.

Таблица 17.2. Трудность классификации задач обработки графов

В данной таблице обобщены приведенные в тексте результаты обсуждения относительной трудности решения различных классических задач обработки графов, в процессе которых сравнение задач проводилось с субъективных точек зрения. Эти примеры не только показывают сложность задач, но и то, что сама классификация конкретной задачи может оказаться довольно-таки трудной проблемой.

	E	T	I	?
Неориентированные графы				
Связность	*			
Полная связность		*		
Эвклидов цикл	*			
Гамильтонов цикл			*	
Двудольное сочетание	*			
Максимальное сочетание		*		
Планарность		*		
Максимальная клика			*	
Раскраска в 2 цвета	*			
Раскраска в 3 цвета			*	
Кратчайшие пути	*			
Самые длинные пути			*	
Вершинное покрытие			*	
Изоморфизм				*
Орграфы				
Транзитивное замечание	*			
Сильная связность	*			
Цикл нечетной длины	*			
Цикл четной длины		*		
Взвешенные графы				
Минимальное остовое дерево	*			
Задача коммивояжера			*	
Сети				
Кратчайшие пути (неотрицательные веса)	*			
Кратчайшие пути (отрицательные веса)			*	
Максимальный поток	*			
Распределение		*		
Поток минимальной стоимости		*		

Обозначения:

E Легкая – известен эффективный классический алгоритм решения (см. справку)

T Решаемая – решение существует (трудно получить реализацию)

I Нерешаемая – эффективное решение неизвестно (NP-трудная задача)

? Неизвестно, существует ли решение

Что касается класса легких алгоритмов, то мы применяем принцип сравнения алгоритмов с различными характеристиками для худших случаев и пытаемся дать оценку производительности алгоритма путем анализа и применения эмпирических критериев. В слу-

чае обработки графов, решение таких задач сопряжено с особыми трудностями ввиду сложности определения характеристик всех типов графов, которые могут встретиться на практике. К счастью, многие важные классические алгоритмы обладают оптимальными или почти оптимальными рабочими характеристиками для худшего случая, либо время их выполнения зависит только от числа вершин и ребер, а не от структуры графа. В силу этого обстоятельства мы можем сконцентрировать свои усилия на выборе оптимальных реализаций и при этом сохранять способность правильно оценивать производительность алгоритмов и доверять этой оценке.

Короче говоря, известно, что существует широкий спектр задач и алгоритмов обработки графов. В таблицу 17.2 заключена некоторая обсуждавшаяся выше информация. Каждая задача представлена в различных вариантах для различных типов графов (ориентированные, взвешенные, двудольные, планарные разреженные, насыщенные), при этом существуют тысячи задач и алгоритмов, заслуживающих изучения. Разумеется, мы не можем рассчитывать на то, что решим любую задачу, с которыми то и дело приходится сталкиваться, при этом некоторые задачи, кажущиеся на первый взгляд легкими для решения, все еще приводят экспертов в замешательство. Несмотря на естественное априорное ожидание, что у нас не возникнет никаких проблем при проведении различия между легкими задачами и задачами, не имеющих решения, многие из рассмотренных нами примеров показывают, что размещение той или иной задачи в какую-либо из указанных выше весьма приблизительную категорию уже само по себе является сложной исследовательской задачей.

По мере того как наши знания о графах и алгоритмах на графах расширяются, конкретная задача может переходить из одной категории в другую. Несмотря на всплеск исследовательской деятельности в семидесятые годы прошлого столетия и интенсивную работу многих исследователей в последующий период, сохраняется определенная вероятность того, что все обсуждаемые нами задачи в один прекрасный день будут классифицироваться как "легкие" (т.е. решаемые посредством применения компактного эффективного и, возможно, оригинального алгоритма).

Подготовив данный контекст, мы теперь основное свое внимание уделим множеству полезных алгоритмов обработки графов. Задачи, которые мы способны решать, возникают часто, алгоритмы на графах, изучением которых мы занимаемся, хорошо работают в самых разнообразных приложениях. Эти алгоритмы служат основой для решения других многочисленных задач, обязательного решения которых от нас требует жизнь, даже если мы не можем гарантировать получение эффективных решений.

Упражнения

- **17.108**. Докажите, что ни один из графов, изображенных на рис. 17.24, не может быть планарным.

17.109. Разработайте АТД графа для клиентской программы, которая выясняет, содержит ли заданный граф хотя бы один из графов, показанных на рис. 17.24. Для этой цели воспользуйтесь алгоритмом решения "в лоб", в котором вы проверяете все возможные подмножества из пяти вершин для клики и все возможные подмножества из шести вершин для полного двудольного графа. *Примечание*: Этой проверки не достаточно, чтобы показать, что граф планарный, поскольку она игнорирует условие, со-

гласно которому удаление вершин степени 2 в некоторых подграфах может дать один из двух запрещенных подграфов.

17.110. Начертите графы

3-7 1-4 7-8 0-5 5-2 3-0 2-9 0-6 4-9 2-6
6-4 1-5 8-2 9-0 8-3 4-5 2-3 1-6 3-5 7-6,

в котором нет пересекающихся ребер, либо докажите, что такой чертеж невозможен.

17.111. Найдите такой способ выбрать 3 цвета вершинам графа

3-7 1-4 7-8 0-5 5-2 3-0 2-9 0-6 4-9 2-6
6-4 1-5 8-2 9-0 8-3 4-5 2-3 1-6 3-5 7-6,

чтобы ни одно ребро не соединяло вершины одного и того же цвета, либо покажите, что это сделать невозможно.

17.112. Решите задачу независимого множества для графа

3-7 1-4 7-8 0-5 5-2 3-0 2-9 0-6 4-9 2-6
6-4 1-5 8-2 9-0 8-3 4-5 2-3 1-6 3-5 7-6.

17.113. Каков размер максимальной клики в графе Де-Бруйна порядка n?

Глава 18

Поиск на графе

Мы часто изучаем сжойства графа, систематически исследуя каждую из его вершин и каждое ребро. Определение некоторых простых свойств – например, вычисление степеней всех его вершин – выполняется просто, если мы исследуем каждое ребро (в любом порядке). Многие другие свойства графа имеют отношение к путям на графах, таким образом, естественный порядок их изучения состоит в переходе от одной вершине к другой вдоль ребер графа. Почти все алгоритмы обработки графов, которые мы применяем, используют эту базовую абстрактную модель. В данной главе мы рассматриваем фундаментальные алгоритмы *поиска на графах (graph search)*, которые используются для перемещения по графам, с изучением по мере продвижения его структурных свойств.

Поиск на графе в таком виде эквивалентен исследованию лабиринта. Точнее, коридоры в лабиринте соответствуют ребрам графа, а точки, в которых коридоры в лабиринте пересекаются, соответствуют взршинам графа. Когда программа меняет значение переменной при переходе от вершины **v** к взршине **w**, что обусловлено наличием ребра **v-w**, мы отождествляем его с передвижением человека из точки **v** в точку **w**. Мы начинаем эту главу с изучения методов систематического исследования лабиринтов. В соответствии с этим процессом, мы наглядно представляем себе, как базовые алгоритмы поиска по графу проходит через каждое ребро и каждую вершину графа.

В частности, рекурсивный алгоритм *DFS* (*depth-first search – поиск в глубину*) точно соответствует конкретной стратегии исследования лабиринта, избранной в главе 18. Поиск в глубину представляет собой классический гибкий алгоритм, который применяется для решения задачи связности и множества других задач обработки графов. Возможны две

реализации этого базового алгоритма: одна в виде рекурсивной процедуры и другая – с использованием явно заданного стека. Замена стека очередью FIFO (First in First Out – первым пришел, первым обслужен) приводит к другому классическому алгоритму – к алгоритму *BFS* (*breadth-first search – поиск в ширину*), который используется для решения других задач обработки графов, связанных с нахождением кратчайших путей.

Основной темой, рассматриваемой в данной главе, являются алгоритмы поиска в глубину, поиска в ширину и другие связанные с ними алгоритмы, а также их применение при обработке графов. Краткий обзор алгоритмов поиска в глубину и поиска в ширину было проведено в главе 5, тогда как здесь мы их рассматриваем на основании первого принципа, в контексте классов, выполняющих обработку графов на основе поиска, и используем их, чтобы показать, какие отношения устанавливаются между различными алгоритмами обработки графов. В частности, мы рассмотрим общий подход к поиску на графах, который охватывает некоторое множество классических алгоритмов обработки графов, в том числе алгоритмов поиска в глубину и поиска в ширину.

В качестве иллюстрации применения этих базовых методов поиска на графах для решения более сложных задач мы будем рассматривать алгоритмы поиска связных компонент, двусвязных компонент, остовных деревьев и кратчайших путей, а также алгоритмы решения множества других задач по обработке графов. Эти реализации являют собой примеры подхода, который мы намерены использовать для решения более трудных задач в главах 19–22.

Мы завершим эту главу исследованием основных проблем, сопряженных с анализом алгоритмов на графах в контексте конкретного случая, сравнивая несколько различных алгоритмов определения числа связных компонент в конкретном графе.

18.1. Исследование лабиринта

Процесс поиска на графах становится поучительным, если представлять его в терминах эквивалентной задачи, которая имеет долгую и интересную историю (*см. раздел ссылок*), – задачи поиска выхода из лабиринта, который состоит из перекрестков, соединенных коридорами. В этом разделе представлен подробный анализ базового метода исследования каждого коридора в заданном лабиринте. Для некоторых лабиринтов конкретная задача может быть решена с применением простого правила, однако для большей части лабиринтов требуется более сложная стратегия (см. рис. 18.1). Использование терминов *лабиринт* (*maze*) вместо *граф*, *коридор* (*passage*) вместо *ребро* и *перекресток* (*intersection*) вместо *вершина* есть ни что иное, как просто семантическое различие, однако на данной стадии переход на другую терминологию поможет глубже прочувствовать задачу.

Один из приемов, применявшийся для исследования лабиринта без риска заблудиться и известный еще с античных времен (берет начало, по меньшей мере, от легенд о Тесее и Минотавре), заключается в том, чтобы разматывать клубок по мере продвижения вглубь лабиринта. Нить гарантирует, что мы всегда сможем выбраться из лабиринта, однако мы заинтересованы в том, чтобы исследовать каждую часть лабиринта и не хотим проходить по уже пройденному пути, пока в этом не появится необходимость. Для достижения этих целей нам нужно некоторое средство, чтобы помечать те места, в которых мы уже были. Разумеется, можно с этой целью использовать и нить, однако мы воспользуемся другим методом, который с большой точностью соответствует компьютерной реализации.

Мы полагаем, что на каждом перекрестке установлены лампы, которые в исходном состоянии выключены, и двери в обоих концах каждого коридора, которые в исходном состоянии закрыты. Далее мы полагаем, что в двери встроены окна, а источники света достаточно мощные и коридоры достаточно прямые, так что мы, открыв дверь, можем определить, освещен или нет перекресток на другом конце коридора (если даже дверь на другом конце коридора закрыта). Наша цель заключается в том, чтобы зажечь все лампы и открыть все двери. Для достижения этой цели мы должны иметь в своем распоряжении набор правил, которым будем систематически следовать. Следующая стратегия исследования лабиринта, которую мы будем называть *исследованием Тремо (Tremaux exploration)*, известна, по меньшей мере, с девятнадцатого столетия (*см. раздел ссылок*):

(*i*) Если на текущем перекрестке нет закрытых дверей, переходите к шагу (*iii*). В противном случае откройте любую дверь любого коридора, ведущую из текущего перекрестка (и оставьте ее открытой).

(*ii*) Если вы видите, что лампа, установленная на перекрестке на другом конце этого коридора уже включена, попробуйте открыть другую дверь на текущем перекрестке (шаг (*i*)). Иначе (если вы видите, что перекресток на другом конце соответствующего коридора не освещен), проследуйте по коридору к этому перекрестку, разматывая при этом нить, включите свет и переходите к шагу (*i*).

(*iii*) Если все двери на текущем перекрестке открыты, проверьте, не находитесь ли вы в отправной точке. Если да, то процесс окончен. Если нет, воспользуйтесь нитью, чтобы двигаться назад вдоль коридора, который привел вас в этот перекресток в первый раз, разматывая нить по мере продвижения, и ищите другую замкнутую дверь уже там (т.е. вернитесь к шагу (*i*).

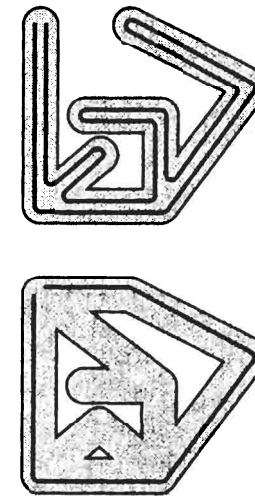


РИСУНОК 18.1. ИССЛЕДОВАНИЕ ЛАБИРИНТА

Мы можем исследовать простой лабиринт, пройдя по каждому коридору и руководствуясь простым правилом типа "держитесь правой рукой за стену". Следуя этому правилу, в лабиринте, изображенном в верхней части рисунка, мы исследуем весь лабиринт, проходя по каждому коридору один раз в каждом направлении. Но если мы будем соблюдать это правило при обходе лабиринта, содержащего цикл, мы вернемся в начальную точку, так и не обследовав весь граф, в чем нетрудно убедиться на примере лабиринта в нижней части рисунка.

На рисунках 18.2 и 18.3 представлен пример обхода графа и показано, что в рассматриваемом случае все лампы зажжены и все двери открыты. Эти рисунки отображают один из множества возможных успешных исходов исследования, поскольку нам предоставлена возможность открывать любую дверь в любом порядке на каждом перекрестке. Чтобы убедиться в том, что этот метод всегда остается эффективным, воспользуйтесь методом математической индукции; для вас это будет очень полезным упражнением.

Свойство 18.1. *Когда мы проводим исследование Тремо некоторого лабиринта, мы зажигаем все лампы и открываем все двери в лабиринте и завершаем обход там, где его начинали.*

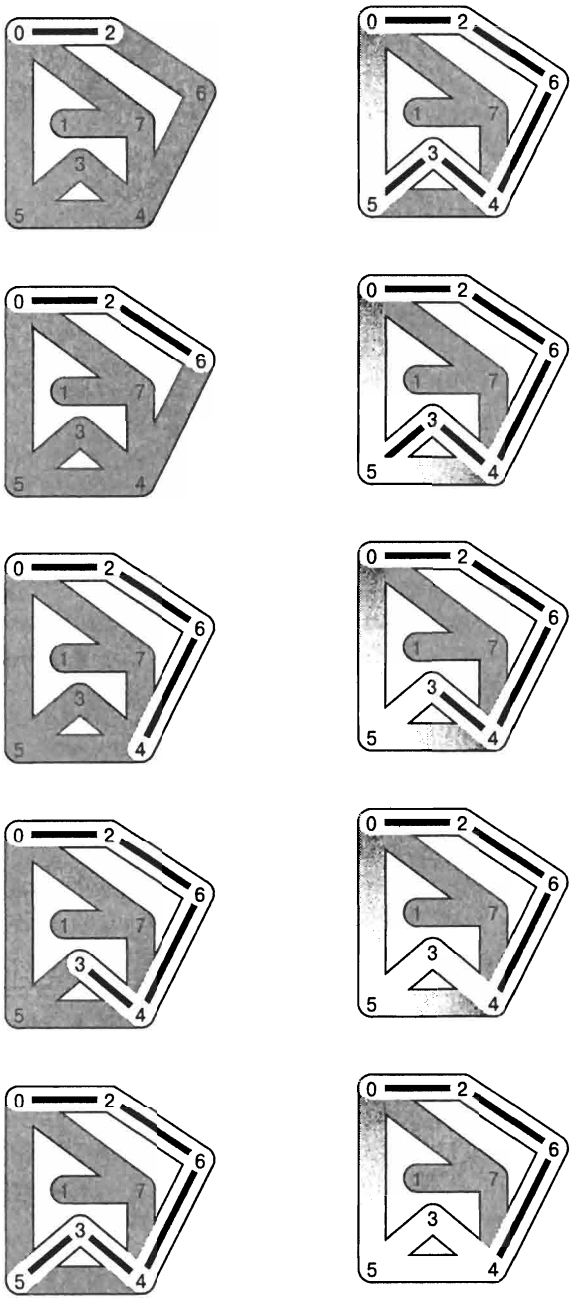
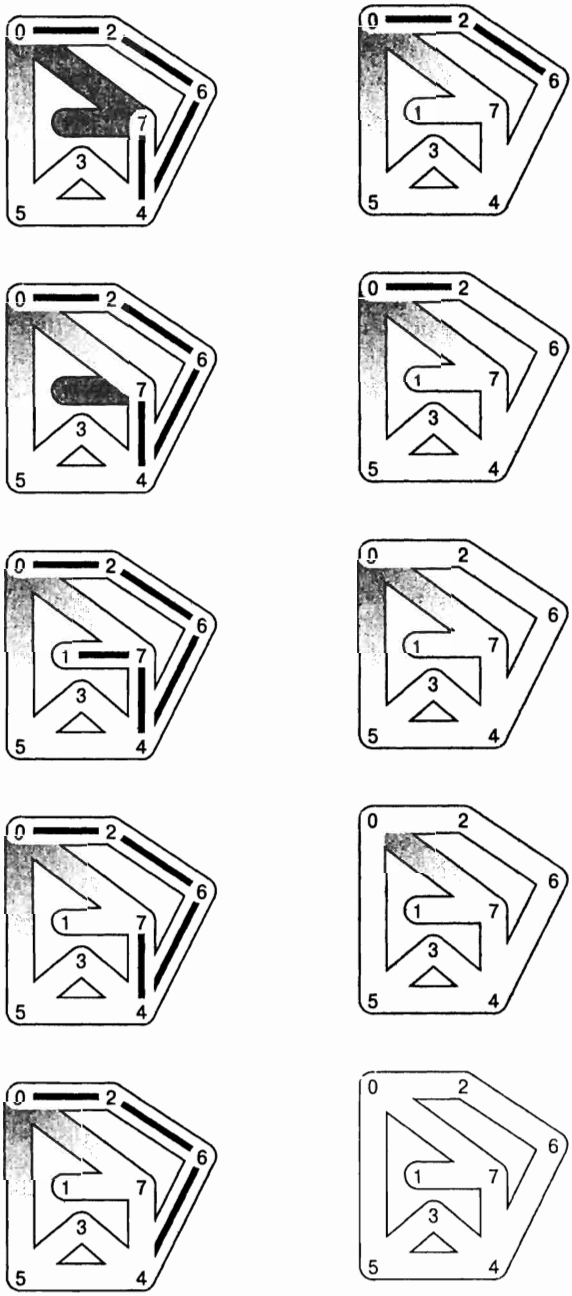


РИСУНОК 18.2. ПРИМЕР ИССЛЕДОВАНИЯ ТРЕМО КОНКРЕТНОГО ЛАБИРИНТА

На этой диаграмме места, которые мы еще не посетили, заштрихованы (темные), а те места, в которых мы уже были, не заштрихованы (светлые). Мы полагаем, что на перекрестках горит свет, и что когда мы открываем двери с обоих концов светлого коридора, этот коридор освещен. Исследование лабиринта мы начинаем с перекрестка ***0*** *и выбираем коридор к перекрестку* ***2*** *(вверху слева). Далее мы продвигаемся по маршруту* ***6***, ***4***, ***3*** *и* ***5***, *открывая двери в коридоры и зажигая свет на перекрестках по мере продвижения и разматывая нить (слева). Открывая дверь, которая ведет из* ***5*** *в* ***0***, *мы замечаем, что перекресток* ***0*** *освещен, поэтому мы игнорируем этот коридор (вверху справа). Аналогично, мы пропускаем коридор от* ***5*** *к* ***4*** *(справа, вторая диаграмма сверху), а здесь нам не остается ничего другого как вернуться из* ***5*** *в* ***3*** *и далее в* ***4***, *наматывая нить на клубок. Когда мы откроем дверь коридора, ведущего из* ***4*** *в* ***5***, *мы видим через открытую дверь на другом конце коридора, что перекресток* ***5*** *освещен, и в силу этого обстоятельства пропускаем этот коридор (справа внизу). Мы ни разу не прошли по коридору, соединяющему перекрестки* ***4*** *и* ***5***, *но мы осветили его, открыв двери с обоих концов.*

РИСУНОК 18.3. ПРИМЕР ИССЛЕДОВАНИЯ ТРЕМО КОНКРЕТНОГО ЛАБИРИНТА (ПРОДОЛЖЕНИЕ)

*Далее мы продвигаемся к перекрестку **7** (слева вверху), открываем дверь и видим, что перекресток **0** освещен (слева, вторая диаграмма сверху), а затем проходим к перекрестку **1** (слева, третья диаграмма сверху). В этой точке, когда большая часть лабиринта уже пройдена, мы используем нить, чтобы вернуться в начало пути, двигаясь от **1** до **7**, далее до **4**, до **6**, до **2** и до **0**. Вернувшись на перекресток **0**, мы завершаем наше исследование, проверяя коридоры, ведущие к перекрестку **5** (справа, вторая диаграмма снизу) и к перекрестку **7** (внизу справа), после чего все коридоры и перекрестки становятся освещенными. И снова оба коридора, соединяющие перекрестки **0** с **5** и **0** с **7**, освещены, поскольку мы открыли двери с обоих концов, однако мы через них не проходим.*



Доказательство: Чтобы доказать это утверждение методом индукции, мы сначала отметим, что оно выполняется в тривиальном случае, т.е. для лабиринта, который содержит один перекресток и ни одного коридора — мы просто включаем свет. Для любого лабиринта, который содержит более одного перекрестка, мы полагаем, что это свойство справедливо для всех лабиринтов с меньшим числом перекрестков. Достаточно показать, что мы посетили все перекрестки, поскольку мы открываем все двери на каждом посещенном перекрестке. Теперь рассмотрим первый коридор, который мы выбираем на первом перекрестке, и разделим все перекрестки на два подмножества: (*i*) те, которые мы можем достичь, выбрав этот коридор и не возвращаясь в отправную точку, и (*ii*) те, которые мы *не можем* достичь, не вернувшись в отправную точку. По индуктивному предположению мы знаем, что посетили все перекрестки в (*i*) (игнорируя все коридоры, возвращающие на начальный перекресток, который освещен) и вернулись на начальный перекресток. Следовательно, применяя индуктивное предположение еще раз, мы знаем, что посетили все перекрестки (игнорируя коридоры, ведущие из отправной точки на перекрестки в (*ii*), которые освещены). ■

Из подробного примера, представленного на рис. 18.2 и 18.3, мы видим, что существуют четыре различных ситуаций, которые возникают при выборе очередного коридора и которые мы должны учитывать, принимая одно из возможных решений:

(*i*) Коридор не освещен, следовательно, мы его выбираем.

(*ii*) Коридор уже был использован (в нем мы размотали нить), следовательно, мы выбираем его (и сматываем нить в клубок).

(*iii*) Дверь на другом конце коридора закрыта (но сам перекресток освещен), в силу этого обстоятельства мы пропускаем этот коридор.

(*iv*) Дверь на другом конце коридора открыта (а перекресток освещен), в силу этого обстоятельства мы пропускаем этот коридор.

Первая и вторая ситуации относятся к любому коридору, который мы обходим, сначала с одного его конца, а затем с другого. Третья и четвертая ситуация относятся к любому коридору, который мы пропускаем, сначала с одного его конца, а затем с другого. Далее мы увидим, как этот план исследования лабиринта преобразуется непосредственно в поиск на графе.

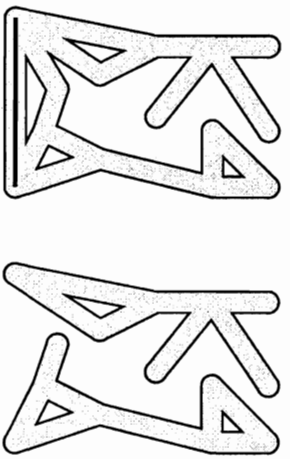


РИСУНОК 18.4. РАЗЛОЖЕНИЕ ЛАБИРИНТА НА ЧАСТИ

Для доказательства методом индукции, что исследование Тремо приводит в любую точку лабиринта (верхний рисунок), мы разбиваем его на две части меньших размеров посредством удаления всех ребер, соединяющих первый перекресток с любым другим перекрестком, который можно достичь из первого коридора без необходимости возврата через первый перекресток (рисунок внизу).

Упражнения

▷ **18.1**. Предположим, что из лабиринта, показанного на рис. 18.2 и 18.3, удалены перекрестки **6** и **7** (а также все ведущие к ним коридоры), зато добавлен коридор, который соединяет перекрестки 1 и 2. В стиле рис. 18.2 и 18.3 покажите, как протекает исследование Тремо полученного лабиринта.

○ **18.2**. Какая из представленных ниже последовательностей не может служить последовательностью включения освещения коридоров в процессе проведения исследования Тремо на лабиринте, представленном на рис. 18.2 и 18.3?

0-7-4-5-3-1-6-2
0-2-6-4-3-7-1-5
0-5-3-4-7-1-6-2
0-7-4-6-2-1-3-5

● **18.3**. Сколько существует путей обхода лабиринта, показанного на рис. 18.2 и 18.3, при проведении исследования Тремо?

18.2. Поиск в глубину

Наши интерес к исследованиям Тремо объясняется тем обстоятельством, что этот метод непосредственно приводит к классической рекурсивной функции обхода графов: посетив конкретную вершину, мы помечаем ее специальной меткой, свидетельствующей о том, что она посещена, затем, в режиме рекурсии, мы посещаем все смежные с нею вершины, которые еще не были помечены. Этот метод, краткий анализ которого проводился в главах 3 и 5 и который использовался для решения задачи нахождения путей в разделе 17.7, носит название *метода поиска в глубину (DFS – depth-first search)*. Это один из наиболее важных алгоритмов, с которыми доведется столкнуться. Метод DFS обманчиво прост, поскольку в его основе лежит знакомая идея и его реализация не вызывает трудностей; фактически это очень гибкий и мощный алгоритм, который применяется для решения множества трудных задач обработки графов.

Программа 18.1 представляет класс DFS, который посещает все вершины и исследует все ребра связного графа. Подобно функциям поиска простого пути, которые рассматривались в разделе 17.7, он базируется на рекурсивной функции и позволяет поддерживать приватный вектор, в котором отмечаются пройденные вершины. В этой программной реализации **ord** представляет собой вектор целых чисел, в котором вершины записываются в порядке их посещения. Рисунок 18.5 отражает трассировку, которая показывает, в каком порядке программа 18.1 производит обход ребер и вершин для примера, показанного на рис. 18.2 и 18.3 (см. также и рис. 18.17); в рас-

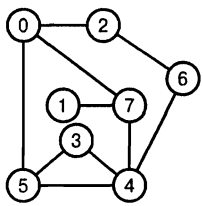


РИСУНОК 18.5. ТРАССИРОВКА DFS

Эта трассировка показывает, в каком порядке алгоритм поиска в глубину проверяет ребра и вершины графа, представленного в виде матрицы смежности и соответствующего примеру, изображенному на рис. 18.2 и 18.3 (вверху), и отслеживает содержимое вектора ***ord*** *(справа) по мере продвижения поиска (звездочки означают -1 для непосещенных вершин). Для каждого ребра графа отводится две строки, по одной на каждое направление. Величина отступа определяет уровень рекурсии.*

сматриваемом случае используется реализация **DenseGRAPH** графа в виде матрицы смежности из раздела 17.3. На рис. 18.6 дается графическое представление процесса исследования лабиринта, для которого используются стандартные чертежи графа.

Указанные рисунки служат иллюстрацией динамики рекурсивного метода DFS и его соответствия с исследованием лабиринта методом Тремо. Во-первых, вектор, проиндексированный вершинами, соответствует освещению перекрестков: когда мы обнаруживаем ребро, ведущее к вершине, на которой мы только что побывали (т.е. "видим свет в конце коридора"), мы не производим рекурсивных вызовов, дабы следовать вдоль этого ребра (т.е. идти вдоль этого коридора). Во-вторых, механизм вызова-возврата функции этой программы является аналогом нити в лабиринте: когда мы обработаем все ребра, инцидентные некоторой вершине (исследуем все коридоры, исходящие от соответствующего перекрестка), мы возвращаемся (в обоих смыслах этого слова).

Программа 18.1. Поиск в глубину связной компоненты

Класс DFS соответствует исследованию Тремо. Конструктор помечает как посещенные все вершины той же связной компоненты, в которой содержится **v**, через вызов рекурсивной функции **searchC**, которая обходит все вершины, смежные с **v**, подвергая их всех проверке, и вызывая саму себя для каждого ребра, которое ведет из **v** в непомеченную вершину. Клиентские программы могут воспользоваться функцией **count** для вычисления числа посещенных вершин и перегруженным оператором [], чтобы узнать, в какой последовательности алгоритм поиска посещал вершины.

```
#include <vector>
template <class Graph> class cDFS
{ int cnt;
   const Graph &G;
   vector <int> ord;
   void searchC(int v)
    {
        ord[v] = cnt++;
        typename Graph::adjIterator A(G, v);
        for (int t = A.beg(); !A.end(); t = A.nxt())
           if (ord[t] == -1) searchC(t);
    }
public:
   cDFS(const Graph &G, int v = 0) :
      G(G), cnt(0), ord(G.V(), -1)
      { searchC(v); }
   int count() const { return cnt; }
   int operator[](int v) const { return ord[v]; }
};
```

Подобно тому, как при обходе лабиринта мы дважды сталкиваемся с каждым коридором (по одному разу с каждого конца), так и в графе мы дважды сталкиваемся с каждым ребром (по одному разу в каждой его вершине). При проведении исследования Тремо мы открываем двери в обоих концах коридора. Применяя поиск в глубину к неориентированному графу, мы проверяем оба представления каждого ребра. Если мы встречаем ребро **v-w**, то либо совершаем рекурсивный вызов (если вершина **w** не помечена), либо пропускаем это ребро (если **w** помечена). Когда мы встретим это же ребро во второй раз, на этот раз с противоположной ориентацией, то есть **w-v**, мы его игнорируем, поскольку вершину назначения **v** мы уже определенно посещали (первый раз, когда сталкивались с этим ребром).

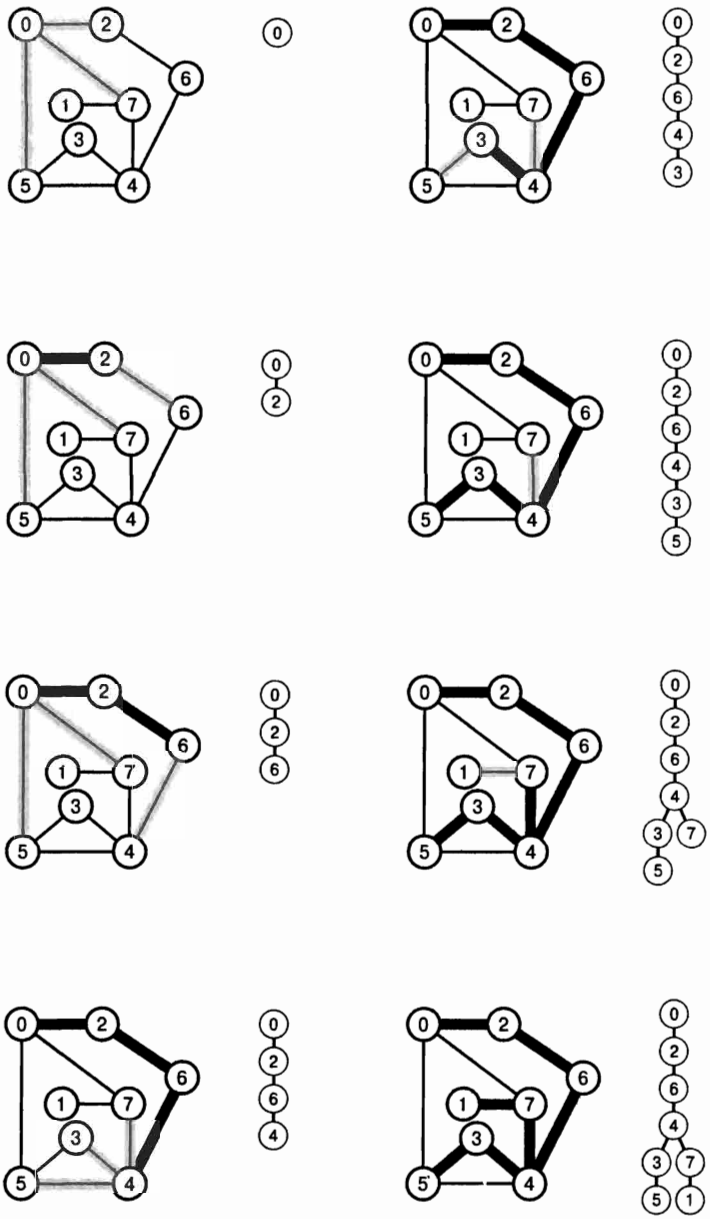


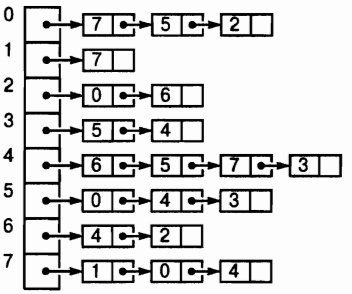
РИСУНОК 18.6. ПОИСК В ГЛУБИНУ

Данные диаграммы представляют собой графическое представление процесса, изображенного на рис. 18.5, на котором показано дерево рекурсивных вызовов поиска в глубину в его развитии. Ребра графа, выделенные толстыми жирными линиями, соответствуют ребрам в дереве DFS, показанном справа от каждой диаграммы графа. Заштрихованные ребра – это ребра, намеченные для включения в дерево на следующих шагах. На ранних стадиях (слева) выполнения алгоритма дерево растет вниз в виде прямой линии, что соответствует рекурсивным вызовам для вершин ***0****,* ***2****,* ***6*** *и* ***4*** *(слева). Затем мы выполняем рекурсивные вызовы для вершины* ***3****, затем для вершины* ***5*** *(справа, две верхних диаграммы), после чего возвращаемся из этих вызовов, чтобы сделать очередной рекурсивный вызов для вершины* ***7*** *из вершины* ***4*** *(справа, вторая снизу) и для* ***1*** *из* ***7*** *(справа снизу).*

Одно из различий между поиском в глубину, каким он реализован в программе 18.1, и исследованием Тремо, каким оно представлено на рис. 18.2 и 18.3, заслуживает того, чтобы потратить некоторое время на его изучение, хотя во многих контекстах оно не играет никакой роли. Когда мы перемещаемся из вершины **v** в вершину **w**, мы не проверяем никаких элементов в матрице связности, которые соответствуют ребрам, ведущим из вершины **w** в другие вершины графа. В частности, мы знаем, что существует ребро из **v** в **w** и оно будет проигнорировано, когда мы на него выйдем (поскольку **v** помечена как посещенная вершина). Подобное решение приходится принимать в моменты, которые отличаются от ситуаций, имеющих место во время проведения исследований Тремо, когда мы открываем дверь, соответствующую ребру, идущему из **v** в **w**, когда мы попадаем в вершину **w** из **v** в первый раз. Если бы мы *закрывали* эти двери во время входа и открывали во время выхода (обозначив соответствующий коридор за счет протягивания вдоль него нити), то в этом случае мы бы имели точное соответствие между поиском в глубину и исследованием Тремо.

На рис. 18.6 показано дерево рекурсивных вызовов, соответствующее рис. 18.5, в динамике его развития. Это дерево рекурсивных вызовов, известное как *дерево DFS*, обеспечивает структурное описание процесса поиска. Как следует из раздела 18.4, дерево DFS, если оно наращивается должным образом, представляет собой полное описание динамики поиска в дополнение к только что описанной структуре вызовов.

Порядок обхода вершин зависит не только от графа, но и от его представления и реализации АТД. Например, на рис.18.7 показана динамика поиска при использовании класса **SparseMultiGRAPH** из раздела 17.4 для реализации представления графа в виде списков смежных вершин. В случае представления графа в виде матрицы смежности мы проводим исследование ребер, инцидентных каждой вершине, в цифровой последовательности. Что же касается представления графа в виде списков смежных вершин, то мы проводим их исследование в том порядке, в каком они выступают в списках. Это различие приводит к совершенно другой динамике рекурсивных поисков, поскольку отличаются последовательности, в которых ребра появляются в списках (что имеет место, например, когда построение одного и того же графа производится путем вставки ребер в различном порядке). Обратите внимание на то обстоятельство, что наличие параллельных ребер не име-




```
                    01234567
0-0                 0*******
  0-7               0******1
    7-1             02*****1
      1-7
    7-0
    7-4             02**3**1
      4-6           02**3*41
        6-4
        6-2         025*3*41
          2-0
            2-6
        4-5         025*3641
          5-0
          5-4
          5-3       02573641
            3-5
            3-4
      4-7
      4-3
  0-5
  0-2
```

РИСУНОК 18.7 ТРАССИРОВКА DFS (СПИСКИ СМЕЖНЫХ ВЕРШИН)

Эта трассировка показывает, в каком порядке алгоритм поиска в глубину проверяет ребра и вершины графа, представленного в виде списков смежных вершин и показанного на рис. 18.5.

ет существенного значения для поиска в глубину, ибо любое ребро, параллельное ребру, которое уже было пройдено, игнорируется, поскольку его вершина назначения уже была отмечена как посещенная.

Несмотря на все эти возможности, критичным остается тот факт, что алгоритм поиска в глубину посещает все ребра и все вершины, связанные с исходной вершиной, *независимо от того, в какой последовательности* он проверяет ребра, инцидентные каждой вершине. Этот факт есть прямое следствие свойства 18.1, поскольку доказательство этого свойства не зависит от порядка, в каком двери открываются на любом заданном перекрестке. Все алгоритмы, в основу которых положен поиск в глубину и которые мы изучаем, обладают этим очень важным свойством. Несмотря на то что динамика их выполнения существенно различается в зависимости от представления графа и деталей реализации поиска, рекурсивная структура дает возможность сделать соответствующие выводы о самом графе независимо от того, как он представлен, и от того, какой будет выбран порядок исследования ребер, инцидентных каждой вершине.

Упражнения

▷ **18.4**. Добавьте в программу 18.1общедоступную функцию-элемент, которая возвращает размер связной компоненты, отыскиваемой конструктором.

18.5. Напишите клиентскую программу, подобную программе 17.6, которая просматривает стандартный ввод с целью выявления графа, использует программу 18.1, осуществляющую поиск из каждой вершины, и выполняет распечатку представление каждого остовного леса в виде родительских связей. Воспользуйтесь реализацией АТД **DenseGRAPH** графа из раздела 17.3.

▷ **18.6**. В стиле рис. 18.5 представьте трассировку вызовов рекурсивной функции, выполненных во время построения объекта **DFS<DenseGRAPH>** для графа

0-2 0-5 1-2 3-4 4-5 3-5.

Сделайте чертеж дерева рекурсивных вызовов, соответствующего DFS.

18.7. В стиле рис. 18.6 покажите, как протекает процесс поиска для примера из упражнения 18.6.

18.3. Функции АТД поиска на графе

Поиск в глубину и другие методы поиска на графах, которые будут рассматриваться позже в этой главе, предусматривают перемещение по ребрам графа при переходе от вершины к вершине с целью систематического обхода каждой вершины и каждого ребра графа. Однако переход от вершины к вершине вдоль ребер может провести через все вершины только в рамках одной и той же связной компоненты, которой принадлежит исходная вершина. Разумеется, в общем случае графы не обязательно должны быть связными, следовательно, мы должны вызывать функцию поиска для каждой связной компоненты. Обычно мы используем функции поиска на графе, которые, пока все вершины графа не будут помечены как посещенные, выполняют следующие действия:

- Найти непомеченную вершину (отправная вершина).
- Посетить (и пометить как посещенные) все вершины в связной компоненте, которая содержит отправную вершину.

Метод маркирования не задан в этом описании, но в большинстве случаев мы применяем тот же метод, который был использован в реализациях поиска в глубину в разделе 18.2: мы инициализируем все элементы приватного вектора, индексированного именами вершин, отрицательным целым числом и помечаем вершины, присваивая соответствующим компонентам вектора неотрицательные значения. Использование этой процедуры сводится к установке значения всего лишь одного разряда (знакового) метки; в условиях большинства реализаций в векторе хранится также и другая информация, имеющая отношение к помеченным вершинам (например, в реализации из раздела 18.2 это порядок, в котором маркируются вершины). Не определен также метод поиска вершин в следующей связной компоненте, но мы чаще всего применяем просмотр этого вектора в порядке возрастания индекса.

Программа 18.2. Поиск на графе

Данный базовый класс предназначен для обработки графов, которые могут оказаться несвязными. Производные классы должны дать определение функции **searchC**, которая, будучи вызванной с петлей вершины **v** в качестве второго аргумента, устанавливает **ord[t]** в **cnt++** для каждой вершины **t**, содержащейся в той же связной компоненте, что и вершина **v**. Чаще всего конструкторы в производных классах вызывают функцию **search**, которая, в свою очередь, вызывает **searchC** один раз для каждой связной компоненты графа.

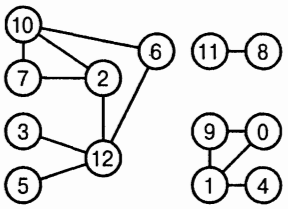
```
template <class Graph> class SEARCH
{
  protected:
    const Graph &G;
    int cnt;
    vector <int> ord;
    virtual void searchC(Edge) = 0;
    void search()
    { for (int v = 0; v < G.V(); v++)
        if (ord[v] == -1) searchC(Edge(v, v)); }
  public:
    SEARCH (const Graph &G) : G(G),
      ord(G.V(), -1), cnt(0) { }
    int operator[](int v) const { return ord[v]; }
};
```

Мы передаем в функцию поиска *ребро* (используя фиктивную петлю в первом вызове для каждой связной компоненты) вместо того, чтобы задавать ей его вершину назначения, ибо ребро говорит нам, как выйти в эту вершину. Если мы знаем ребро, то это равносильно знанию, какой коридор ведет к конкретному перекрестку в лабиринте. Эта информация полезна во многих классах DFS. Если мы просто отслеживаем, в какие вершины мы нанесли визит, то от этой информации мало толку, в то же время более интересные задачи требуют знания места, откуда мы пришли.

Программа 18.2 представляет собой реализацию, которая служит иллюстрацией этих возможностей. На рис. 18.8 представлен пример, который показывает, как отражается посещение каждой вершины на состоянии вектора **ord** любого производного класса. Как правило, рассматриваемые производные классы также исследуют все ребра, инцидентные каждой посещенной вершине. В таких случаях знание того факта, что мы посетили все вершины, говорит нам, что мы также посетили все ребра, как и в случае обхода Тремо.

Программа 18.3 представляет собой пример, который показывает, как можно получить базовый класс DFS для вычисления остовного леса на основе базового класса **SEARCH** из программы 18.2. Мы включаем приватный вектор **st** в производный класс с целью хранения представления дерева в виде родительских связей, которое мы инициализируем в конструкторе. Кроме того, мы даем определение функции **searchC**, которая во всем совпадает с функцией **searchC** из программы 18.1 за исключением того, что она принимает ребро **v-w** в качестве аргумента и устанавливает **st[w]** в **v**. Наконец, мы вводим общедоступную функцию-элемент, которая дает клиентам возможность определять родителя любой вершины. Остовные леса являются предметом интереса для многих приложений, но в этой главе они интересуют нас главным образом в силу того, что они важны для понимания динамического поведения поиска в глубину, что является темой обсуждений раздела 18.4.

В связном графе конструктор из программы 18.2 вызывает функцию **searchC** всего один раз, а именно, для ребра **0-0**, после чего выясняет, что все другие вершины помечены. В графе, состоящем из более чем одной связной компоненты, конструктор просматривает все связные компоненты непосредственно. Поиск в глубину является первым из нескольких методов, который будет применяться для просмотра связной компоненты графа. Независимо от того, какой метод (при этом не играет роли, каково представление графа) был выбран, программа 18.2 является эффективным методом просмотра всех вершин графа.



	0	1	2	3	4	5	6	7	8	9	10	11	12
	*	*	*	*	*	*	*	*	*	*	*	*	*
0-0	0	0	*	*	0	*	*	*	*	0	*	*	*
2-2	0	0	0	0	0	0	0	0	*	0	0	*	0
8-8	0	0	0	0	0	0	0	0	0	0	0	0	0

РИСУНОК 18.8. ПОИСК НА ГРАФЕ

*Таблица в нижней части рисунка содержит метки вершин (содержимое вектора **ord**) во время обычного поиска на графе, который показан в верхней части рисунка. Сначала функция **GRAPHsearch** из программы 18.2 отменяет старые метки всех вершин, устанавливая метки -1 всем вершинам (в таблице проставлены звездочки (*)). Затем она вызывает функцию **search** для фиктивного ребра **0-0**, которая помечает все вершины, содержащиеся в той же компоненте, что и **0** (второй ряд таблицы), назначая им неотрицательные значения (в таблице проставлены **0**). В рассматриваемом примере она помечает вершины **0**, **1**, **4** и **9** значениями от **0** до **3** в этом же порядке. Далее она производит просмотр слева направо, в результате которого находит неотмеченную вершину 2 и вызывает функцию **search** для фиктивного ребра **2-2** (третий ряд таблицы). Упомянутая функция **search** помечает семь вершин, содержащиеся в той же компоненте, что и **2**. Продолжая просмотр слева направо, она вызывает **search** для ребра **8-8**; в результате этого вызова помечаются вершины **8** и **11** (нижний ряд таблицы). И, наконец, функция **GRAPHsearch** завершает поиск, убедившись в том, что все вершины от **9** до **12** помечены.*

Программа 18.3. Производный класс поиска в глубину

Данный программный код демонстрирует, как строится производный класс DFS остового дерева на основе базового класса, определение которого дано в разделе 18.2. Конструктор выполняет построение представления леса в векторе **st** (родительские связи) и в векторе **ord** (из базового класса). Клиенты могут использовать объект **DFS** для поиска родителя любой заданной вершины леса (**ST**) или позиции любой заданной вершины при прямом обходе леса (перегруженный оператор **[]**). Свойства таких лесов и их представления изучаются в разделе 18.4.

```
template <class Graph>
class DFS : public SEARCH<Graph>
{ vector<int> st;
  void searchC(Edge e)
    { int w = e.w;
      ord[w] = cnt++; st[e.w] = e.v;
      typename Graph::adjIterator A(G, w);
      for (int t = A.beg(); !A.end(); t = A.nxt())
        if (ord[t] == -1) searchC(Edge(w, t));
    }
public:
  DFS(const Graph &G) : SEARCH<Graph>(G),
    st(G.V(), -1) { search(); }
  int ST(int v) const { return st[v]; }
};
```

Свойство 18.2. *Функция поиска на графе проверяет каждое ребро и помечает каждую вершину заданного графа тогда и только тогда, когда функция поиска, которую использует граф, помечает каждую вершину и проверяет каждое ребро связной компоненты, содержащей исходную вершину.*

Доказательство: Проводится методом индукции по числу связных компонент. ■

Функции поиска на графе представляют собой систематический способ обработки каждой вершины и каждого ребра графа. В общем случае программные реализации разрабатывались нами с расчетом их выполнения за линейное или за примерно линейное время благодаря тому, что обработка каждого ребра требует выполнения фиксированного числа операций. Мы докажем этот факт для поиска в глубину, попутно отметив, что тот же метод доказательства работает и в отношении других стратегий поиска.

Свойство 18.3: *Поиск в глубину на графе, представленном матрицей смежности, требует времени, пропорционального V^2.*

Доказательство: Рассуждения, аналогичные доказательству свойства 18.1, показывают, что функция **searchC** не только маркирует все вершины, связанные с исходной вершиной, но и вызывает сама себя в точности один раз для такой вершины (чтобы ее пометить). Рассуждения, аналогичные доказательству свойства 18.2, показывает, что вызов функции **search** приводит к вызову функции **searchC** для каждой вершины в точности один раз. В функции **searchC** итератор проверяет каждый элемент строки вершины в матрице смежности. Другими словами, поиск проверяет каждый элемент матрицы смежности в точности один раз. ■

Свойство 18.4: *Поиск в глубину на графе, представленного списками смежных вершин, требует времени, пропорционального $V + E$.*

Доказательство: Из приведенных выше рассуждений следует, что мы обращаемся к рекурсивной функции в точности V раз (отсюда происходит слагаемое V), кроме того, мы также проверяем каждое вхождение в списке смежных вершин (отсюда происходит слагаемое E). ■

Основной вывод, который следует из свойства 18.3 и 18.4, заключается в том, что время выполнения поиска в глубину линейно зависит от размеров структуры данных, используемой для представления графа. В большинстве ситуаций вполне оправданно считать, что

время выполнения поиска в глубину линейно зависит от размеров самого графа: если речь идет о насыщенном графе (число ребер которого пропорционально V^2), то любое представление дает этот результат; если речь идет о разреженном графе, то мы предпочитаем пользоваться представлением в виде списков смежных вершин. В самом деле, обычно мы полагаем, что время выполнения поиска в глубину линейно зависит от E. Это утверждение формально неверно, если для представления разреженных графов не используются матрицы смежности, или для исключительно разреженных графов, для которых $E << V$ и большая часть вершин изолирована; однако мы обычно избегаем второй ситуации за счет удаления изолированных вершин (см. упражнение 17.34).

Как будет показано далее, все эти рассуждения применимы к любому алгоритму, для которого характерны несколько основных свойств поиска в глубину. Если алгоритм помечает каждую вершину и проверяет все вершины, инцидентные рассматриваемой вершине (и проделывает другую работу, на выполнение которой затрачивается время, ограниченное некоторой константой), то эти свойства характерны и для него. В более общей формулировке это звучит так: если время, затрачиваемое на просмотр каждой вершины, ограничено некоторой функцией $f(V,E)$, то есть гарантия того, что время поиска в глубину пропорционально $E + f(V,E)$. В разделе 18.8 мы увидим, что поиск в глубину является одним из алгоритмов из семейства, которому свойственны эти характеристики; в главах 19–22 мы убедимся в том, что алгоритмы этого семейства служат основой достаточно большого числа программ, которые рассматриваются в данной книге.

Большая часть изучаемых нами программ обработки графов представляет собой программы реализации АТД для некоторых конкретных задач, в рамках которых мы разрабатываем класс, выполняющий базовый поиск для вычисления структурированной информации в других векторах, проиндексированных именами вершин. Мы можем построить такой класс на основе программы 18.2 или, в более простых случаях, просто реализовать поиск повторно. Многие из построенных нами классов обработки графов обладают этими свойствами, поскольку, как привило, при выполнении поиска на графе мы получаем представление о его структуре. Обычно мы расширяем код функции поиска, которая выполняется, когда каждая вершина помечена, вместо того, чтобы работать с алгоритмом поиска более общего характера (например, с алгоритмом поиска, который вызывает некоторую специальную функцию каждый раз, когда встречается помеченная вершина) только из желания сделать программный код модульным и более компактным. Построение для клиентских программ механизма АТД более общего характера, обеспечивающего обработку всех вершин с помощью функций, предоставляемых самими клиентами, несомненно заслуживает всяческого внимания (см. упражнения 18.13 и 18.14).

В разделах 18.5 и 18.6 мы проведем исследование многочисленных функций обработки графов, которые основаны на использовании алгоритма DFS. В разделах 18.5 и 18.6 мы рассмотрим другие реализации функции **search** и ряда других функций обработки графов, в основе которых лежат реализации этой функции. Хотя мы и не встроили этот уровень абстракции в наш программный код, мы, тем не менее, позаботимся о том, чтобы было понятно, какая основная стратегия поиска на графе лежит в основе разрабатываемого алгоритма. Например, мы применяем термин *класс DFS* для ссылки на любую реализацию, основанную на рекурсивной схеме поиска в глубину. Программы, определяющие класс поиска простого пути (программа 17.16) и класс основного леса (программа 18.3) могут служить примерами классов DFS.

Многие функции обработки графов основаны на использовании векторов, индексируемых именами вершин. Обычно такие векторы включаются в реализации классов как приватные элементы данных и содержат информацию о структуре графов (которая накапливается в нем во время поиска), что содействует решению непрерывно возникающих задач. Примерами таких векторов могут служить вектор **deg** в программе 17.11 и вектор **ord** в программе 18.1. Некоторые из реализаций, которые будут рассмотрены ниже, используют различного рода векторы для изучения сложных структурных свойств.

В качестве соглашения будем полагать, что все векторы, индексированные именами вершин, инициализируются в функциях поиска на графе минус единицей (-1), а все компоненты этого вектора, соответствующие посещенным вершинам, в функциях поиска на графе принимают неотрицательные значения. Любой такой вектор может использоваться в качестве вектора **ord** (маркирование вершин как посещенных) в программах 18.2 и 18.3. Если функция поиска на графе основана на использовании или вычислении вектора, проиндексированного именами вершин, то обычно, осуществляя поиск, мы используем этот вектор для маркирования вершин, а не для построения производного класса от базового класса **SEARCH** или для поддержки вектора **ord**.

Конкретные результаты поиска на графе зависят не только от природы функции поиска, но и от представления графа и даже от порядка, в каком функция **search** осуществляет просмотр вершин. В силу специфики примеров и упражнений, в данной книге употребляется термин *стандартный DFS по спискам смежных вершин* (*standard adjacency-lists DFS*) для обозначения процесса вставки последовательности ребер в АТД графа, реализованного на основе построения представления графа в виде списков смежных вершин (программа 17.9) с последующим выполнением поиска в глубину, например, через программу 18.3. В случае представления графа в виде матрицы смежности порядок вставки ребер не влияет на динамику поиска, тем не менее, мы будем пользоваться параллельным термином *стандартный DFS по матрице смежности* (*standard adjacency-matrix DFS*) для обозначения процесса вставки последовательности ребер в АТД графа, реализованного на основе построения представления графа в виде матрицы смежности (программа 17.9) с последующим выполнением DFS, например, с помощью программы 18.3.

Упражнения

18.8. В стиле рис. 18.5 покажите трассировку вызовов рекурсивной функции, выполняемых для стандартного DFS по матрице смежности графа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

18.9. В стиле рис. 18.7 покажите трассировку вызовов рекурсивной функции, выполняемых для стандартного DFS по спискам смежных вершин графа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

18.10. Внесите такие изменения в реализацию АТД графа, представленного в виде матрицы смежности (программе 17.7), чтобы можно было использовать фиктивную вершину, связанную со всеми другими вершинами графа. Постройте реализацию упрощенного алгоритма поиска в глубину, учитывающего эти изменения.

18.11. Выполните упражнение 18.10 для реализации АДТ графа, представленного в виде списка смежных вершин (Программа 18.2).

- **18.12**. Существует 13! перестановок вершин в графе, показанном на рис. 18.8. Какая часть этих перестановок может служить описанием очередности, в которой вершины графа посещаются программой 18.2?

18.13. Постройте программную реализацию клиентской функции АТД графа, которая вызывает клиентскую функцию для каждой вершины графа.

18.14. Постройте программную реализацию клиентской программы АТД графа, которая вызывает клиентскую функцию для каждого *ребра* графа. Такая функция может оказаться приемлемой альтернативой функции **GRAPHedges** (см. программу 17.2).

18.4. Свойства лесов DFS

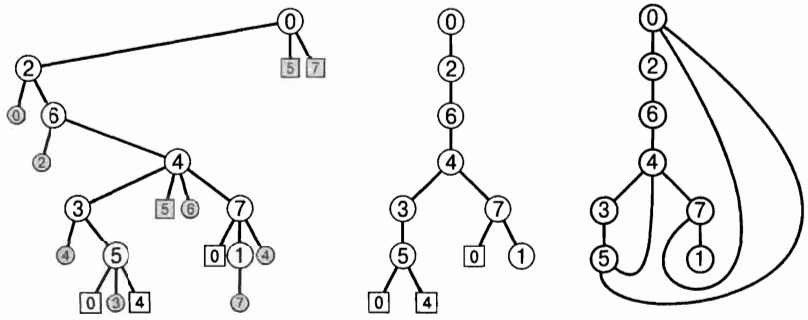
Как уже говорилось в разделе 18.2, деревья, которые описывают рекурсивную структуру вызовов функции DFS, дают ключ для понимания того, как выполняется поиск в глубину. В данном разделе мы будем изучать свойства этого алгоритма на примере исследования свойств деревьев DFS.

Если добавить дополнительные узлы в дерево DFS для фиксации моментов, когда мы пропускаем рекурсивные вызовы для уже посещенных вершин, то мы получим компактное представление о динамике поиска в глубину, иллюстрацией которой может служить рис. 18.9. Это представление заслуживает подробного изучения. Дерево такого типа является представлением графа, при этом каждой вершине дерева соответствует вершина графа, а каждому ребру – ребро графа. Можно выбрать два представления ребра, которое подвергается обработке (по одному в каждом направлении), как показано в левой части рис. 18.9, или только одно представление, как показано в центральной и правой частях рисунка. Первое из них полезно для понимания того факта, что рассматриваемый алгоритм обрабатывает любое и каждое ребро, второе же полезно для понимания, что дерево DFS есть ни что иное, как еще одно представление графа. Обход внутренних узлов дерева в прямом порядке располагает вершины в последовательности, в которой поиск в глубину их обходит; более того, порядок, в котором происходит посещение *ребер* деревьев при обходе дерева в прямом порядке, совпадает с порядком, в котором поиск в глубину осуществляет просмотр *ребер* в графе.

В самом деле, дерево DFS, показанное на рис. 18.9, содержит ту же информацию, что и трассировка на рис. 18.5 или пошаговая иллюстрация обхода Тремо на рис. 18.2 и 18.3. Ребра, ведущие ко внутренним узлам, представляют собой ребра (коридоры), ведущие к непосещенным вершинам (перекресткам), а заштрихованные узлы представляют собой ребра, ведущие к вершинам, для которых в данный момент выполняется рекурсивный поиск в глубину (в момент, когда мы открываем дверь в коридор, в котором дверь на противоположном конце уже открыта). В условиях такой интерпретации прямой обход дерева говорит нам то же, что и подробный сценарий обхода лабиринта.

Чтобы изучить более сложные свойства графа, проведем классификацию ребер графа в соответствии с той ролью, которую они играют в поиске. Имеются два четко определенных класса ребер:

- ребра, представляющие рекурсивные вызовы (*древесные* ребра – *tree* edges);
- ребра, соединяющие конкретную вершину с ее предшественником в дереве DFS, который не является ее родителем (*обратные* ребра – *back* edges)).



	0	1	2	3	4	5	6	7
ord	0	7	1	4	3	5	2	6
st	0	7	0	4	6	3	2	4

РИСУНОК 18.9. РАЗЛИЧНЫЕ ПРЕДСТАВЛЕНИЯ ДЕРЕВА DFS

*Если расширить дерево рекурсивных вызовов поиска в глубину таким образом, чтобы в него были включены ребра, которые мы проверяем, но не проходим, то получим полное описание процесса поиска в глубину (слева). У каждого узла дерева имеется потомок, представляющий каждый смежный с ним узел в том порядке, в каком они рассматривались алгоритмом DFS, а обход в прямом порядке дает ту же информацию, что и рис. 18.5: сначала мы проходим путь **0-0**, затем **0-2**, далее мы пропускаем **2-0**, затем мы проходим по **2-6**, пропускаем **6-2**, затем проходим по **6-4**, **4-3** и так далее. Вектор **ord** определяет последовательность, в которой мы посещаем вершины дерева во время обхода в прямом порядке и который аналогичен последовательности посещения вершин графа алгоритмом DFS. Вектор **st** является представлением в виде родительских связей дерева рекурсивных вызовов поиска в глубину (см. рис. 18.6).*

Для каждого ребра графа в этом дереве имеются две связи, по одной на каждый из двух раз, когда встречается это ребро. Первая связь ведет в незаштрихованный узел и соответствует либо выполнению рекурсивного вызова (если связь ведет во внутренний узел), либо пропуску рекурсивного вызова, поскольку она направлена в предшествующий узел, для которого в данный момент выполняется рекурсивный вызов (если она ведет во внутренний узел). Вторая связь ведет в заштрихованный внешний узел и всегда соответствует пропуску рекурсивного вызова либо в силу того, что она ведет назад к родителю (кружки), либо в силу того, что ведет к потомку родителя, для которого в данный момент выполняется рекурсивный вызов (квадратики). Если удалить заштрихованные узлы, то получится другой чертеж рассматриваемого графа (справа).

При исследовании деревьев DFS орграфов в главе 19 мы будем изучать другие типы ребер, когда в расчет принимаются не только направления ребер, ибо в графе возможны такие ребра, которые идут вразрез со структурой дерева, соединяя узлы, не являющиеся в рассматриваемом дереве ни предками, ни потомками.

Поскольку существуют два представления каждого ребра графа, и каждое ребро соответствует связи в дереве DFS, разделим связи дерева на четыре класса, воспользовавшись *номерами предпорядка* (*preorder number*) и родительскими связями (соответственно, в массивах **ord** и **st**), которые вычисляет наша программа поиска в глубину. Мы рассматриваем связь между вершинами **v** и **w** в дереве DFS, которая представляет ребро дерева, как

- *древесная* связь (*tree* link), если **v** не помечена;
- *родительская* связь (*parent* link), если **st[w]** есть **v**,

а связь от **v** к **w**, которая представляет обратное ребро как

- связь *назад* (*back* link), если **ord[w]** < **ord[v]**;
- связь *вниз* (*down* link), если **ord[w]** > **ord[v]**.

Каждое древесное ребро в графе соответствует древесной связи и родительской связи в дереве DFS, а каждое обратное ребро в графе соответствует связи назад и связи вниз в дереве DFS.

В графическом представлении поиска в глубину, показанном на рис. 18.9, древесные связи указывают на незаштрихованные квадратики, родительские связи – на незаштрихованные кружки, а связи вниз – на заштрихованные квадратики. Каждое ребро графа представлено либо одной древесной связью и одной родительской связью, либо одной связью вниз и одной связью назад. Такая классификация достаточно сложна и заслуживает внимательного изучения. Например, обратите внимание, что даже если родительские связи и связи назад одновременно указывают на предков в дереве, они различны: родительская связь всего лишь другое представление древесной связи, в то время как связь назад дает нам новую информацию о структуре графа.

Данные выше определения предоставляют достаточную информацию, чтобы провести различие между связью древовидной структуры, родительской связью, связью назад и связью вниз в реализации класса DFS. Обратите внимание на то обстоятельство, что для родительских связей и связей назад выполняется условие **ord[w]** < **ord[v]**, т.е., чтобы знать, что **v-w** представляет собой связь назад, мы должны также знать, что **st[w]** не есть **v**. На рис.18.10 представлена распечатка результатов классификации связей древесной структуры DFS для каждого ребра графа по мере того, как эти ребра встречаются в рассматриваемом примере процесса поиска в глубину. Это еще одно полное представление базового процесса поиска, которое может служить промежуточным этапом между рисунками 18.5 и 18.9.

Четыре описанных выше типа связей в древесных структурах соответствуют четырем различным видам трактовки ребер в процессе выполнения поиска в глубину; об этом шла речь (в терминах обхода лабиринта) в конце раздела 18.1. Древесная связь соответствует случаю, когда поиск в глубину сталкивается с первыми двумя представлениями ребра дерева, что приводит к очередному рекурсивному вызову (для исследования еще не просмотренных вершин); роди-

```
0-0 древесная связь
  0-2 древесная связь
    2-0 родительская связь
    2-6 древесная связь
      6-2 родительская связь
      6-4 древесная связь
        4-3 древесная связь
          3-4 родительская связь
          3-5 древесная связь
            5-0 связь назад
            5-3 родительская связь
            5-4 связь назад
        4-5 связь вниз
        4-6 родительская связь
        4-7 древесная связь
          7-0 связь назад
          7-1 древесная связь
            1-7 родительская связь
          7-4 родительская связь
  0-5 связь вниз
  0-7 связь вниз
```

РИСУНОК 18.10. ТРАССИРОВКА ПОИСКА В ГЛУБИНУ (КЛАССИФИКАЦИЯ СВЯЗЕЙ В ДРЕВЕСНЫХ СТРУКТУРАХ)

Данный вариант рис. 18.5 отображает классификацию связей в дереве DFS, соответствующую представлению каждого ребра графа. Древесные ребра (которые соответствуют рекурсивным вызовам) представлены как древесные связи (такими их видит поиск в глубину при первой встрече с ними) и как родительские связи (такими их видит поиск в глубину при второй встрече). Обратные ребра суть связи назад при первой встрече с ними и связи вниз при второй встрече.

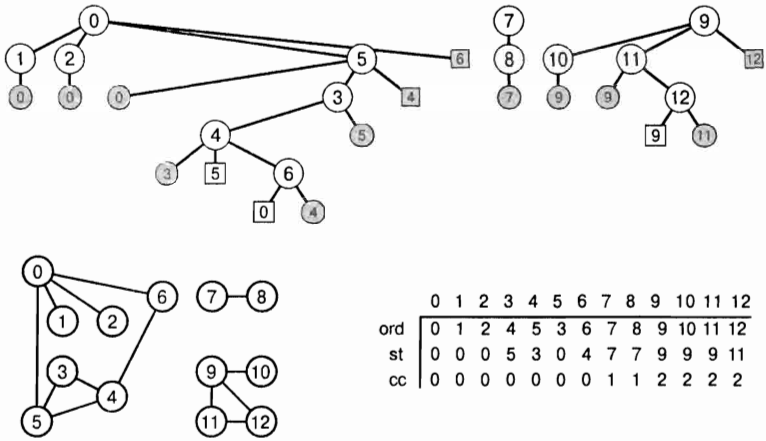
тельская связь соответствует случаю, когда поиск в глубину сталкивается с другим представлением древесного ребра (при просмотре списка смежности при первом таком рекурсивном вызове) и игнорирует это ребро. Связь назад соответствует случаю, когда поиск в глубину сталкивается с первым из двух возможных представлений обратного ребра, указывающего на вершину, для которой рекурсивный поиск еще не закончен. Связь назад соответствует случаю, когда поиск в глубину выходит на вершину, для которой рекурсивный поиск закончен в тот момент, когда поиск в глубину сталкивается с таким ребром. На рис. 18.9 древесные связи и связи назад соединяют незаштрихованные узлы, что означает первый выход поиска в глубину на соответствующее ребро, и являются представлением графа; родительские связи и связи вниз ведут в заштрихованные узлы и означают вторую встречу с соответствующим ребром.

Мы подробно рассматривали это древовидное представление динамических характеристик рекурсивного алгоритма DFS не только из-за того, что оно являет собой полное и лаконичное описание работы этого алгоритма, но и в силу того, что оно представляет собой основу для понимания множества важных алгоритмов обработки графов. В оставшейся части данной главы и в нескольких последующих главах мы рассмотрим целый ряд примеров задач обработки графов, в рамках которых делаются выводы относительно структуры графа на базе дерева DFS.

Поиск на графе является обобщением обхода дерева. Примененный к поиску на дереве, алгоритм DFS в точности эквивалентен рекурсивному обходу дерева; его использование применительно к графу соответствует обходу дерева, которое отображает этот граф и построение которого производится по мере продвижения поиска. Как мы уже имели возможность убедиться, конкретный обход дерева зависит от представления графа. Поиск в глубину соответствует обходу дерева в прямом порядке. В разделе 18.6 мы познакомимся с алгоритмом поиска на графе, аналогичным обходу дерева по уровням, и выясним, как он соотносится с алгоритмом DFS; в разделе 18.7 мы изучим общую схему, которая охватывает практически все методы обхода.

При обходе графов с помощью поиска в глубину мы использовали вектор **ort** для присвоения вершинам номеров в прямом порядке, т.е. для присвоения номеров в той последовательности, в какой мы приступаем к их обработке. Мы будем также присваивать вершинам номера в *обратном порядке* (*postorder numbers*), т.е. номера в той последовательности, в какой мы *завершаем* их обработку (непосредственно перед выходом из функции рекурсивного поиска). В процессе обработки конкретного графа мы делаем больше, нежели просто обходим вершины – как можно будет убедиться позже, нумерация в прямом и в обратном порядке предоставляет сведения о глобальных свойствах графа, что помогает справиться с решением текущей задачи. Нумерации в прямом порядке оказывается вполне достаточно для реализации алгоритмов, рассматриваемых в данной главе, а нумерация в обратном порядке будет использоваться в последующих главах.

Сейчас мы опишем динамику поиска в глубину применительно к неориентированным графам общего вида, представленным в виде *леса DFS (DFS forest)*, в котором каждое дерево DFS представляет одну несвязную компоненту графа. Пример леса DFS показан на рис. 18.11.



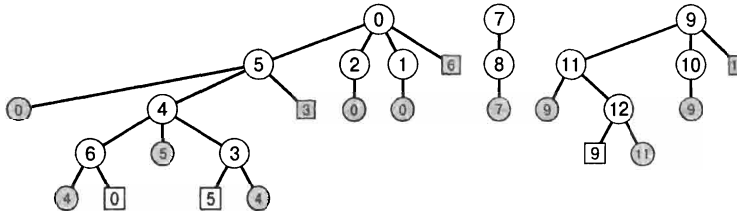
	0	1	2	3	4	5	6	7	8	9	10	11	12
ord	0	1	2	4	5	3	6	7	8	9	10	11	12
st	0	0	0	5	3	0	4	7	7	9	9	9	11
cc	0	0	0	0	0	0	0	1	1	2	2	2	2

РИСУНОК 18.11. ЛЕС DFS

Лес DFS, изображенный в верхней части рисунка, служит иллюстрацией поиска в глубину на графе, представленном в виде матрицы связности и показанном в нижней правой части диаграммы. Рассматриваемый граф состоит из трех связных компонент, в соответствии с этим фактом лес содержит три дерева. Вектор **ort** *представляет прямой порядок нумерации узлов в дереве (порядок, в котором они подвергаются исследованию со стороны DFS), а вектор* **st** *есть представление леса в виде родительских связей. Вектор* **cc** *связывает каждую компоненту с индексом связной компоненты (см. программу 18.4). Как и в случае рис. 18.9, ребра, ведущие к кружками, – это древесные ребра, а ребра, ведущие к квадратикам, – обратные ребра; заштрихованные узлы показывают, что инцидентное ребро было обнаружено раньше, когда поиск проводился в обратном направлении.*

В условиях представления графа в виде списков смежных вершин обход ребер, связанных с каждой вершиной, производится в очередности, отличной от порядка, рассчитанного на представление графа в виде матрицы связности, следовательно, мы получаем другой лес DFS, как показано на рис. 18.12. Деревья и леса DFS суть представления графов, которые описывают не только динамику поиска в глубину, но также и внутреннее представление графов. Например, считывая потомков любого узла на рис. 18.12 слева направо, мы видим порядок, в каком они появляются в списке вершин, смежных с вершиной, соответствующей этому узлу. Для одного и того же графа может существовать множество лесов – каждое расположение узлов в списках смежных вершин приводит к появлению другого леса.

Особенности структуры конкретного леса позволяют понять, как ведет себя DFS на том или ином графе, но большая часть свойств DFS, которые представляют для нас интерес, зависят от свойств графа, которые, в свою очередь, не зависят от структуры леса. Например, оба леса, показанные на рис. 18.11 и 18.12, содержат три дерева (как и любой другой лес DFS одного и того же графа), поскольку они всего лишь различные представления одного и того же графа, состоящего из трех связных компонент. В самом деле, прямое следствие основного доказательства того, что поиск в глубину посещает все узлы и ребра конкретного графа (см. свойства 18.2–18.4), заключается в том, что число связных компонент графа равно числу деревьев в лесе DFS. Этот пример служит иллюстрацией положения, используемого в качестве основы для поиска на графе на протяжении этой книги: разнообразие реализаций классов обработки графов основано на изучении свойств графа путем обработки конкретного его представления (лес, соответствующий поиску).



	0	1	2	3	4	5	6	7	8	9	10	11	12
ord	0	6	5	4	2	1	3	7	8	9	12	10	11
st	0	0	0	4	5	0	4	7	7	9	9	9	11
cc	0	0	0	0	0	0	0	1	1	2	2	2	2

РИСУНОК 18.12. ДРУГОЙ ЛЕС DFS

Данный лес описывает поиск в глубину на том же графе, что и изображенный на рис. 18.11, но в этом случае используется представление графа в виде списков смежных вершин, в силу чего меняется порядок поиска, поскольку в рассматриваемом случае он определяется порядком, в котором узлы появляются в списках смежных вершин. В самом деле, лес сам подсказывает нам этот порядок: это порядок, в каком перечисляются потомки в каждом узле дерева. Например, узлы в списке смежных вершин для вершины ***0*** *расположены в порядке* ***5 2 1 6****, узлы в списке смежных вершин для вершины* ***4*** *— в порядке* ***6 5 3*** *и т.д. Как и раньше, все вершины и ребра графа просматриваются во время поиска способом, который был точно описан с помощью обхода дерева в прямом порядке. Векторы* ***ord*** *и* ***st*** *зависят от представления графа и динамики поиска, вследствие чего они отличаются от приводимых на рис. 18.11, в то же время вектор* ***cc*** *зависит только от свойств графа, благодаря чему остается неизменным.*

В принципе, можно проводить анализ древесных структур DFS, имея перед собой цель повысить производительности алгоритма. Например, нужно ли предпринимать попытки повысить быстродействие алгоритма путем переделки списков смежных вершин перед началом поиска? Для многих важных классов алгоритмов, использующих поиск в глубину, ответ на этот вопрос отрицательный, поскольку они оптимальны — время их выполнения в худшем случае не зависит ни от структуры графа, ни от последовательности, в какой ребра появляются в списках смежных вершин (по существу, они проводят обработку каждого ребра в точности один раз). Тем не менее, леса DFS обладают характерной структурой, которая заслуживает изучения уже за то, что она отличает их от другой фундаментальной схемы, которая будет рассматриваться далее в этой главе.

На рис. 18.13 показано дерево DFS графа больших размеров, которое служит иллюстрацией базовых характеристик динамики поиска в глубину. Это дерево высокое и тонкое, оно демонстрирует несколько свойств просматриваемого графа и процесса поиска в глубину.

- Существует, по меньшей мере, один длинный путь, который соединяет существенную часть узлов.
- Во время поиска большая часть вершин имеет, по меньшей мере, одну смежную вершину, которую мы еще не видели.
- Мы редко делаем более одного рекурсивного вызова с одного узла.
- Глубина рекурсии пропорциональна числу вершин графа.

Подобное поведение типично для поиска в глубину, хотя эти характеристики и не гарантируются для всех графов. Проверка наличия фактов этого рода для моделей графов, представляющих интерес, и различных типов графов, возникающих на практике, требует подробных исследований. В то же время этот пример позволяет выработать интуитивное восприятие алгоритмов, использующих поиск в глубину, которое часто подтверждается на практике. Рисунок 18.13 и аналогичные иллюстрации других алгоритмов поиска на графе (см. рис. 18.24 и 18.29) помогают лучше понять различия в их поведении.

Упражнения

18.15. Сделайте чертеж леса DFS, который получается в результате применения стандартного DFS к графу, заданному в виде матрицы смежности:

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

18.16. Сделайте чертеж леса DFS, который получается в результате применения стандартного DFS к графу, заданному в виде списков смежных вершин:

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

18.17. Составьте программу трассировки поиска в глубину, которая в стиле упражнения 18.10 производит классификацию каждого из двух представлений любого ребра графа на соответствие древесной связи, родительской связи, связи назад или связи вниз в дереве DFS.

○ **18.18.** Напишите программу, которая вычисляет представление полного дерева DFS в виде родительских связей (включая внешние узлы) с использованием вектора из E целых чисел, принимающих значения в диапазоне от 0 до $V-1$. *Указание*: Первые две компоненты этого вектора должны быть такими же, что и компоненты вектора **st**, описание которого дано в тексте.

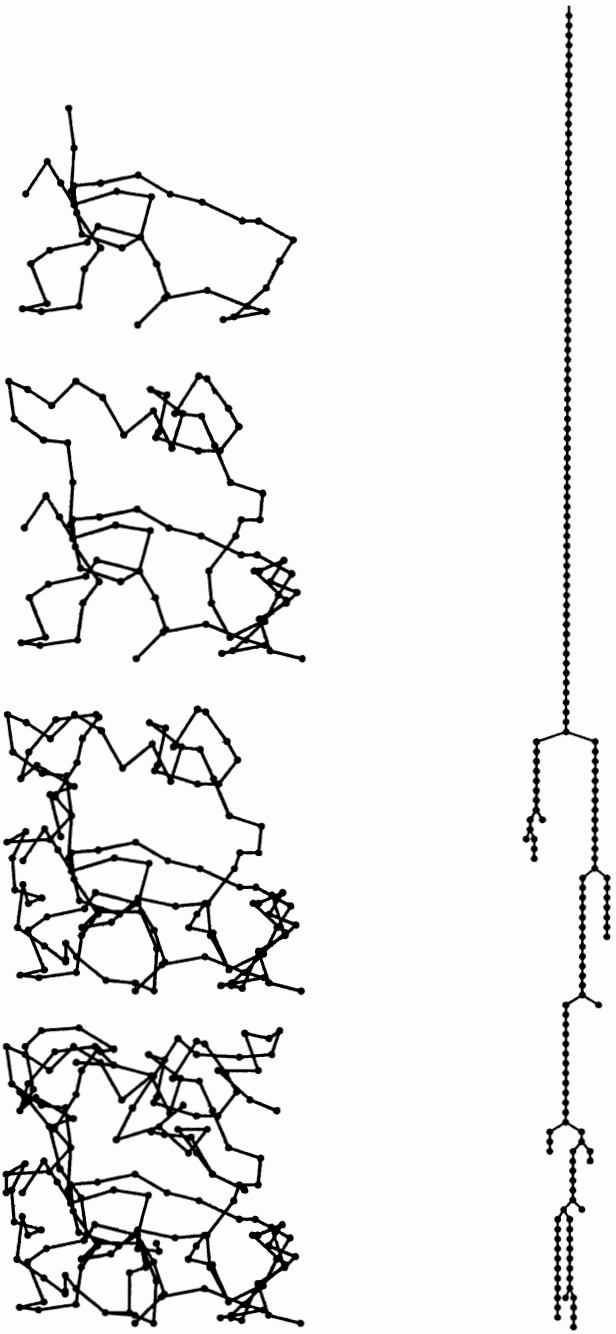
○ **18.19.** Снабдите класс DFS остовного дерева (программа 18.3) функциями-элементами (и соответствующими элементами данных), которые подсчитывают высоту самого высокого дерева леса, число обратных ребер и процент ребер, которые необходимо обработать, чтобы просмотреть каждую вершину.

● **18.20.** Проведите эксперименты с целью определения эмпирическим путем средних значений количественных величин, описанных в упражнении 18.19, для графов различных размеров, полученных для графов различных моделей (см. упражнения 17.64–17.76).

● **18.20.** Напишите функцию, выполняющую построение графа за счет вставки ребер, выбранных случайным образом из заданного вектора и вставки в первоначально пустой граф. Используя эту функцию вместе с реализацией АТД графа, представленного в виде списков смежных вершин, проведите эксперименты с целью определения эмпирическим путем свойств *распределения* количественных величин, описанных в упражнении 18.19, для всех представлений примеров крупных графов различных размеров в виде списков смежных вершин и различных моделей (см. упражнения 17.64–17.76).

РИСУНОК 18.13.
ПОИСК В ГЛУБИНУ

Этот рисунок показывает, как протекает процесс поиска в глубину в случайном эвклидовом графе с близкими связями (слева). На рисунке показаны вершины и ребра дерева DFS в графе по мере того, как процедура поиска просматривает 1/4, 1/2, 3/4 и все вершины графа (сверху вниз). Дерево DFS (только древесные ребра) показано справа. Из этого примера следует, что для дерева поиска в глубину для графов рассматриваемого типа (как и для многих других типов графов, часто встречающихся на практике) характерна узкая продолговатая форма. Обычно мы находим где-то рядом вершину, которая еще не подвергалась просмотру.



18.5 Алгоритмы DFS

Независимо от структуры и представления графа, любой лес DFS позволяет нам определять, какие ребра являются древесными, а какие – обратными, и содействует правильному пониманию структуры графа, которое, в свою очередь, позволяет использовать поиск в глубину в качестве основы для решения многочисленных задач обработки графов. В разделе 17.7 мы уже ознакомились с основными примерами, имеющими отношение к поиску путей. В этом разделе мы рассмотрим реализации функции АТД, использующие DFS, позволяющие решать эту и многие другие типовые задачи. В остальной части данной главы и в нескольких последующих главах мы рассмотрим различные подходы к решению намного более сложных задач.

Обнаружение циклов. Существуют ли в заданном графе какие-либо циклы? (Является ли рассматриваемый граф лесом?) Эта задача легко решается с помощью поиска в глубину, поскольку любое обратное ребро дерева DFS принадлежит циклу, состоящему из этого ребра плюс путь в дереве, соединяющий два узла (см. рис. 18.9). Таким образом, можно немедленно воспользоваться поиском в глубину с целью выявления циклов: граф является ациклическим тогда и только тогда, когда во время выполнения поиска в глубину не встречаются обратные (или прямые!) ребра. Например, для проверки этого условия в программу 18.1 мы просто добавляем предложение **else** в оператор **if** с целью проверки равенства **t** и **v**. Если равенство установлено, это означает, что мы столкнулись с родительской связью **w-v** (второе представление ребра **v-w**, которое привело нас в **w**). Если равенство не установлено, то **w-t** завершает цикл в дереве DFS проверкой ребер из **t** в **w**. Более того, нет необходимости проверять все ребра: мы знаем, что должны найти цикл или завершить поиск, не обнаружив его, прежде чем проверим V ребер, поскольку любой граф с V или большим числом ребер должен содержать в себе цикл. Следовательно, мы можем проверить, является ли рассматриваемый граф ациклическим или за время, пропорциональное V в случае его представления в виде списков смежных вершин, хотя будет затрачено время, пропорциональное V^2 (чтобы найти ребра), если граф задан в виде матрицы смежности.

Простой путь. Существует ли путь в графе, который связывает две заданных вершины? В разделе 17.7 мы видели, что нетрудно построить класс DFS, который способен решить эту задачу за линейное время.

Простая связность. Как следует из исследований, проведенных в разделе 18.3, мы за линейное время определяем, является ли граф связным, всякий раз, когда пользуемся поиском в глубину. В самом деле, выбранная нами стратегия основана на вызове функции поиска для каждой связной компоненты. При проведении поиска в глубину граф относится к категории связных тогда и только тогда, когда функция поиска на графе вызывает рекурсивную функцию DFS всего лишь один раз (программа 18.2). Число связных компонент в графе в точности равно числу вызовов рекурсивной функции из функции **GRAPHsearch**, следовательно, число связных компонент графа можно определить путем простого подсчета числа таких вызовов.

В общем случае программа 18.4 служит иллюстрацией класса DFS, который поддерживает выполнение запросов, касающихся связности, за постоянное время после этапа препроцессорной обработки в конструкторе. Он поддерживает тот же порядок посещения вершин, что и программа 18.3. Вместо ребра рекурсивная функция в качестве свое-

го второго аргумента использует вершину, поскольку ей не нужно знать, кто является ее родителем. Каждому дереву леса DFS соответствует связная компонента графа, так что мы быстро можем определить, содержатся ли две вершины в одной и той же компоненте, включив в представление графа вектор, проиндексированный именами вершин, для последующего его заполнения через поиск в глубину и доступа со стороны запросов о связности. В рекурсивной функции DFS мы присваиваем текущее значение счетчика компоненты элементу вектора, соответствующему каждой посещенной вершине. Далее, мы знаем, что две вершины находятся в одной компоненте графа тогда и только тогда, когда соответствующие им элементы этого векторе равны. Опять-таки, обратите внимание на тот факт, что данный вектор отображает структурные свойства графа, но не особенности представления графа или динамики поиска.

Программа 18.4 типизирует базовый подход, который мы намереваемся использовать при решении различных задач обработки графов. Мы разработаем класс, ориентированный на решение специальных задач, чтобы клиенты могли создавать объекты, решающие эту задачу. Как правило, все время, затрачиваемое на предварительную обработку, расходуется конструктором, который вычисляет приватные данные, описывающие соответствующие структурные свойства графа. Эти свойства графа помогают обеспечить эффективную реализацию общедоступных функций запросов. В данном случае конструктор выполняет предварительную обработку с помощью поиска в глубину (за линейное время) и поддерживает приватные элементы данных (вектор **id**, проиндексированный именами вершин), который позволяет отвечать на запросы о связности за постоянное время. Что касается других задач обработки графов, то используемые нами конструкторы могут допускать более высокие затраты пространства памяти, времени на предварительную обработку и на ответы на запросы. Как обычно, основное внимание мы уделяем минимизации этих затрат, хотя во многих случаях сделать это весьма непросто. Например, большая часть главы 19 посвящена решению задачи связности орграфов, в рамках которых линейное время на предварительную обработку и постоянное время на обработку запросов о связности, аналогично программе 19.1, представляет собой труднодостижимую цель.

Программа 18.4. Связность графа

Конструктор **CC** вычисляет за линейное время количество связных компонент заданного графа и сохраняет индекс компоненты, ассоциированной с каждой вершиной, в приватном векторе **id**, проиндексированном именами вершин. Клиентские программы могут использовать объект **CC** для определения за постоянное время числа связных компонентов (**count**) или для проверки (**connect**), является ли связанной конкретная пара вершин.

```
template <class Graph> class CC
{ const Graph &G;
  int ccnt;
  vector <int> id;
  void ccR(int w)
    {
      id[w] = ccnt;
      typename Graph::adjIterator A(G, w);
      for (int v = A.beg(); !A.end(); v = A.nxt())
        if (id[v] == -1) ccR(v);
    }
```

```
public:
   CC(const Graph &G) : G(G), ccnt(0), id(G.V(), -1)
     {
        for (int v = 0; v < G.V(); v++)
          if (id[v] == -1) { ccR(v); ccnt++; }
     }
   int count() const { return ccnt; }
   bool connect(int s, int t) const
     { return id[s] == id[t]; }
};
```

Какими преимуществами решение задачи определения связности графа, использующее поиск в глубину и реализованное программой 18.4, обладает перед подходом, использующим алгоритм объединения-поиска, который рассматривался в главе 1 применительно к решению задачи определения связности графа, в случае, когда граф задан списком ребер? Теоретически поиск в глубину обладает большим быстродействием, чем алгоритм объединения-поиска, поскольку он обеспечивает постоянное время выполнения, в то время как алгоритм объединения-поиска не способен на это; на практике, однако, эта разница незначительна. В конечном итоге, алгоритм объединения-поиска выполняется быстрее, поскольку для него не надо строить полное представление графа. Что еще важнее, алгоритм объединения-поиска работает в оперативном режиме (мы можем проверить, имеется ли связь между двумя какими-либо вершины за время, близкое к постоянному), в то время как решение, использующее поиск в глубину, должно выполнить предварительную обработку, чтобы ответить на запрос о связности за постоянное время. В силу этих причин, мы предпочитаем применять алгоритм объединения-поиска, например, когда определение связности графа является нашей единственной целью или когда многочисленные запросы перемешиваются с операциями вставки ребер, в то же время мы можем посчитать решение с использованием поиска в глубину более подходящим применительно к АТД графа, поскольку оно с большей эффективностью использует существующую инфраструктуру. Ни тот, ни другой подход не способен работать эффективно в условиях смеси большого числа вставок ребер, удалений ребер и запросов определения связности; оба подхода требуют отдельных поисков в глубину для вычисления пути. Эти рассуждения показывают, с какими осложнениями приходится сталкиваться при анализе алгоритмов на графах; подробное их исследование проводится в разделе 18.9.

Программа 18.5. Двухпроходный эйлеров цикл

Этот класс DFS печатает каждое ребро дважды, по одному в каждом направлении, в порядке двухпроходного эйлерова цикла. Мы перемещаемся в разных направлениях по обратным ребрам и игнорируем прямые ребра (см. текст). Этот класс является производным от базового класса **SEARCH** из программы 18.2.

```
template <class Graph>
class EULER : public SEARCH<Graph>
{
   void searchC(Edge e)
   { int v = e.v, w = e.w;
      ord[w] = cnt++;
      cout << "-" << w;
      typename Graph::adjIterator A(G, w);
      for (int t = A.beg(); !A.end(); t = A.nxt())
        if (ord[t] == -1) searchC(Edge(w, t));
```

```
        else if (ord[t] < ord[v])
           cout << "-" << t << "-" << w;
      if (v != w) cout << "-" << v; else cout << endl;
}
public:
   EULER(const Graph &G) : SEARCH<Graph>(G)
     { search(); }
};
```

Двухпроходный эйлеров цикл. Программа 18.5 реализует класс, использующий поиск в глубину и осуществляющей поиск пути, который использует все ребра графа в точности два раза – по одному в каждом направлении (см. раздел 17.7). Этот путь соответствует исследованию Тремо, по условиям которого мы разматываем нить, куда бы ни шли. Мы не применяем освещения перекрестках, зато проверяем, есть ли нить в коридоре (следовательно, мы должны проходить по коридорам, ведущим к перекресткам, на которых уже были), и сначала подготавливаем возможность перемещаться взад и вперед по каждой связи назад (первый раз, когда мы сталкиваемся с тем или иным обратным ребром), после чего игнорируем связи вниз (при втором выходе на каждое обратное ребро). Можно также игнорировать связи вниз (при первой встрече) и перемещаться взад и вперед по связям вниз (вторая встреча) (см. упражнение 18.25).

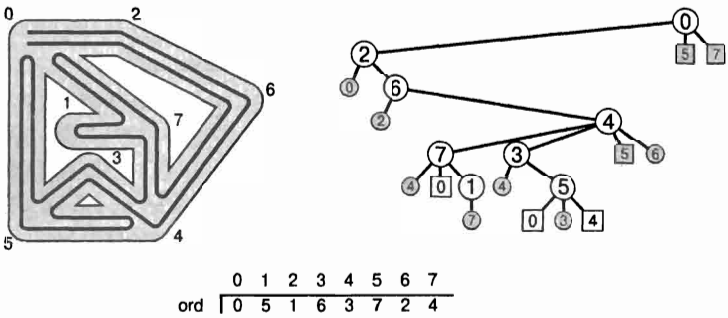


РИСУНОК 18.14. ДВУХПРОХОДНЫЙ ЭЙЛЕРОВ ЦИКЛ

*Поиск в глубину предоставляет возможность исследовать любой лабиринт, проходя коридор в каждом направлении. Мы вносим изменения в исследование Тремо, которое заключается в том, что мы разматываем нить, куда бы ни шли, и проходим взад и вперед по коридорам, в которых нет нити и которые ведут к посещенным перекресткам. На этом рисунке показан порядок обхода, который отличается от изображенного на рисунках 18.2 и 18.3 главным образом тем, что мы можем найти такой путь обхода, который не пересекает сам себя. Такое упорядочение может иметь место, например, когда при построении списков смежных вершин для представления графа ребра подвергались обработке в каком-то другом порядке; либо когда могли быть внесены изменения непосредственно в поиск в глубину с таким расчетом, чтобы было принято во внимание геометрическое расположение узлов (см. упражнение 18.26). Двигаясь по нижнему пути, исходящему из **0** и далее через **2**, **6**, **4** и **7**, мы возвращаемся из **7** в **0**, затем меняем направление обхода на обратное, поскольку **ord[0]** меньше **ord[7]**. Затем мы идем в **1**, назад в **7**, назад в **4**, в **3**, в **5**, из **5** следуем в **0** и назад, идем назад из **5** в **4** и назад, далее назад в **3**, назад в **4**, назад в **6**, назад в **2** и назад в **0**. Такой путь может быть получен с помощью рекурсивного обхода в прямом и обратном порядке дерева DFS (игнорируя при этом заштрихованные вершины, которые означают, что мы встречаемся с данным ребром второй раз), при этом мы распечатываем имя соответствующей вершины, просматриваем в рекурсивном режиме поддеревья, затем снова печатаем имя этой вершины.*

Остовный лес. На заданном связном графе с V вершинами требуется найти множество из $V-1$ ребер, соединяющее эти вершины. Если в графе имеются C связных компонент, необходимо найти остовный лес (с V-C ребрами). Мы уже ознакомились с классом DFS, который решает эту задачу: см. программу 18.3.

Поиск вершин. Сколько вершин находится в той же компоненте, что и заданная вершина? Мы легко можем решить эту проблему, начав поиск в глубину с заданной вершины и подсчитывая количество помеченных вершин. В насыщенном графе мы можем существенно ускорить этот процесс, остановив поиск в глубину после пометки V вершин – в этой точке мы уже знаем, что никакое ребро не приведет нас в вершину, которой мы еще не видели, поэтому мы игнорируем остальные ребра. Подобное усовершенствование, по-видимому, позволит нам посетить все вершины за время, пропорциональное $V \log V$, но не E (см. раздел 18.8).

Раскраска в два цвета, двухдольные графы, нечетные циклы. Существует ли способ покрасить каждую вершину в два цвета таким образом, что ни одно из ребер не соединяет вершины одного и того же цвета? Принадлежит ли заданный граф к категории двудольных (см. раздел 17.1)? Содержит ли заданный граф цикл нечетной длины? Все эти три задачи эквивалентны: первые две суть различные названия одной и той же задачи; любой граф, содержащий нечетный цикл, не допускает описанную выше раскраску в два цвета, в то же время программа 18.6 показывает, что любой граф, в котором нет нечетных циклов, может быть раскрашен в два цвета. Указанная программа представляет собой реализацию функции АТД, использующую поиск в глубину, которая проверяет, является ли заданный граф двудольным, допускает ли он окраску в два цвета, содержит ли он нечетные циклы. Эту рекурсивную функцию можно рассматривать как схему доказательства методом индукции того факта, что программа может раскрасить в два цвета любой граф, свободный от нечетных циклов (или обнаружить в графе нечетный цикл как доказательство того, что граф, не свободный от нечетный циклов, не может быть раскрашен в два цвета). Чтобы раскрасить граф в два цвета, когда процесс раскраски на-

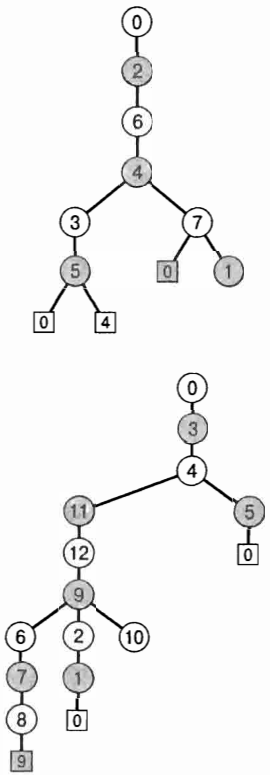


РИСУНОК 18.15. РАСКРАСКА ДЕРЕВА DFS В ДВА ЦВЕТА

*Чтобы раскрасить граф в два цвета, мы меняем цвет в процессе движения вниз по дереву DFS, затем проводим проверку обратных ребер на несовместимость. В дереве, изображенном в верхней части рисунка, дерево DFS графа, взятого в качестве примера и представленного на рис. 18.9, обратные ребра **5-4** и **7-0** показывают, что рассматриваемый граф не допускает окраски в два цвета ввиду наличия циклов нечетной длины **4-3-5-4** и **0-2-6-4-7-0**. В дереве в нижней части рисунка, представляющем собой дерево DFS двудольного графа, изображенного на рис. 17.5, таких несоответствий не существует; в рассматриваемом случае окраска в два цвета изображена с применением штриховки.*

чинается с того, что заданную вершину v раскрашивают в заданный цвет, все вершины, смежные с v, закрашиваются в другой цвет, отличный от цвета вершины v. Этот процесс эквивалентен раскраске соседних уровней дерева DFS по мере того, как мы спускаемся вниз, проверяя обратные ребра на соответствие цветов (см. рис. 18.15). Любое обратное ребро, соединяющее вершины одного цвета, являются доказательством наличия в рассматриваемом графе нечетных циклов.

Программа 18.6. Раскраска графа в два цвета (двудольные графы)

Конструктор рассматриваемого класса DFS устанавливает переменной **OK** значение **true** тогда и только тогда, когда может установить значение 0 или 1 вектору **vc**, индексированному именами вершин таким образом, что каждое из ребер **v-w** графа, **vc[v]** и **vc[w]**, являются попарно различными.

```
template <class Graph> class BI
{ const Graph &G;
  bool OK;
  vector <int> vc;
  bool dfsR(int v, int c)
    {
      vc[v] = (c+1) %2;
      typename Graph::adjIterator A(G, v);
      for (int t = A.beg(); !A.end(); t = A.nxt())
        if (vc[t] == -1)
          { if (!dfsR(t, vc[v])) return false; }
        else if (vc[t] != c) return false;
      return true;
    }
public:
  BI(const Graph &G) : G(G), OK(true), vc(G.V(),-1)
  {
  for (int v = 0; v < G.V(); v++)
    if (vc[v] == -1)
      if (!dfsR(v, 0)) { OK = false; return; }
  }
  bool bipartite() const { return OK; }
  int color(int v) const { return vc[v]; }
};
```

Эти типовые примеры служат иллюстрацией способов, с помощью которых поиск в глубину может дать представление о структуре графа. Они также показывают, что мы можем изучить важнейшие свойства графа посредством его просмотра за линейное время, в процессе которого мы исследуем каждое ребро дважды, по разу в каждом направлении. Далее мы рассмотрим пример, который показывает, как поиск в глубину используется с целью обнаружения более сложных свойств структуры графа за линейное время.

Упражнения

- **18.22.** Постройте реализацию класса, использующего поиск в глубину для проверки существования циклов, который выполняет в конструкторе предварительную обработку графа за время, пропорциональное V, с целью поддержки функций-элементов, определяющих, присутствуют ли в графе какие-либо циклы, и выполняющих распечатку циклов, если таковые имеются.

18.23. Дайте описание семейства графов с V вершинами, для которого на выполнение стандартного DFS на матрице смежности с целью обнаружения циклов затрачивается время, пропорциональное V^2.

▷ **18.24**. Постройте реализацию класса определения связности из программы 18.4 как производного от класса поиска на графе, подобного программе 18.3.

▷ **18.25**. Укажите, какие изменения следует внести в программу 18.5, чтобы она была способна вычислить двусторонний эйлеров цикл, который совершает обход взад и вперед прямых, а не обратных ребер.

• **18.26**. Внесите в программу 18.5 такие изменения, чтобы она всегда могла вычислить двухпроходной эйлеров цикл, который, аналогично показанному на рис. 18.14, может быть начерчен таким образом, что не пересекает сам себя ни в одной вершине. Например, если бы поиск, представленный на рис. 18.14, должен был пройти по ребру **4-3** перед тем, как пройти по ребру **4-7**, то цикл пересек бы сам себя; ваша задача заключается в том, чтобы добиться того, чтобы алгоритм избегал подобного рода ситуаций.

18.27. Разработайте версию программы 18.5, которая сортирует все ребра в порядке двухпроходного эйлерова цикла. Ваша программа должна возвратить вектор ребер, который соответствует двухпроходному эйлерову циклу.

18.28. Покажите, что граф принадлежит к числу раскрашиваемых двумя цветами тогда и только тогда, когда он не содержит нечетных циклов. *Указание*: Докажите методом индукции, что программа 18.6 способна определить, является ли любой заданный граф раскрашиваемым двумя цветами.

○ **18.29**. Объясните, почему подход, использованный в программе 18.6, не допускает обобщения, которое позволило бы получить эффективный метод определения, является ли конкретный граф раскрашиваемым тремя цветами.

18.30. Большая часть графов не относится к категории раскрашиваемых двумя цветами, причем поиск в глубину во многих случаях быстро обнаруживает этот факт. Проведите эмпирические испытания с целью определения числа ребер, исследованных программой 18.6, на графах различных размеров и построенных по различным моделям (см. упражнения 17.64–17.76).

○ **18.31**. Докажите, что в каждом связном графе имеются вершины, удаление которых не нарушает связности графа, и напишите функцию DFS, которая обнаруживает такие вершины. *Указание*: Рассмотрите листья дерева DFS.

18.31. Докажите, что каждый граф, состоящий из более чем одной вершины, содержит, по меньшей мере, две вершины, удаление который не приводит в увеличению числа связных компонент.

18.6. Отделимость и бисвязность

Чтобы продемонстрировать широкие возможности поиска в глубину как основы алгоритмов обработки графов, мы обратимся к задачам, имеющим отношение к обобщенному понятию связности в графах. Мы займемся изучением проблем такого рода: пусть заданы две вершины, существуют ли два различных пути, связывающих эти вершины?

В некоторых ситуациях, когда важно, чтобы граф *был* связным, может оказаться существенным тот факт, что он *остается* связным, если убрать из него какую-либо вершину

или ребро. То есть, мы, возможно, захотим знать более одного пути между каждой парой вершин графа с тем, чтобы застраховаться от возможных отказов и неисправностей. Например, мы можем лететь из Нью-Йорка в Сан-Франциско, даже если аэропорт в Чикаго завален снегом, ибо существует рейс через Денвер. Или можно вообразить себе ситуацию во время военных действий, в условиях которой мы хотим проложить такую железнодорожную сеть, когда противник, дабы нарушить железнодорожное сообщение, должен разбомбить, по меньшей мере, две станции. Аналогично, мы вправе рассчитывать, что соединения в интегральной схеме или в сети связи проложены таким образом, что остальная часть схемы продолжает работать, если оборвался какой-либо провод или какое-то соединение перестало работать.

Упомянутые примеры принципиально отличаются друг от друга: в случае интегральной схемы и сети связи мы заинтересованы в сохранении соединения, когда удаляется *ребро*; в случае авиа- и железнодорожных сообщений мы хотим сохранить соединение, когда удаляется *вершина*. Мы начнем с того, что подробно рассмотрим второй случай.

Определение 18.1. Мостом (bridge) *в графе называется ребро, после удаления которого связный граф распадается на два не связанных между собой подграфа. Граф, у которого нет мостов, называется* **реберно-связным (edge-connected)**.

Когда мы говорим об *удалении* ребра, мы имеем в виду удаление этого ребра из множества ребер, которое определяет граф, даже если после такого удаления одна или обе вершины графа останутся изолированными. Реберно-связный граф остается связным, когда удаляется какое-либо одно ребро. В некоторых контекстах целесообразнее говорить о возможности нарушения связности графа, а не о способности графа оставаться связным. Таким образом, мы будем свободно пользоваться альтернативной терминологией, которая делает акцент на таких моментах: мы называет граф, который относится к категории реберно-связных, *реберно-отделимым* (*edge-separable*) и назовем мосты *ребрами отделимости* (*separation edges*). Если мы удалим все мосты в реберно-отделимом графе, мы разделим его на *реберно-связные компоненты* (*edge-connected components*) или компоненты, связанные мостами (*bridge-connected components*) – максимальные подграфы, не имеющие мостов. На рис. 18.16 показан небольшой пример, который может послужить иллюстрацией этих понятий.

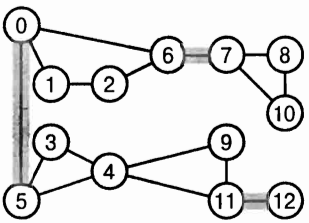
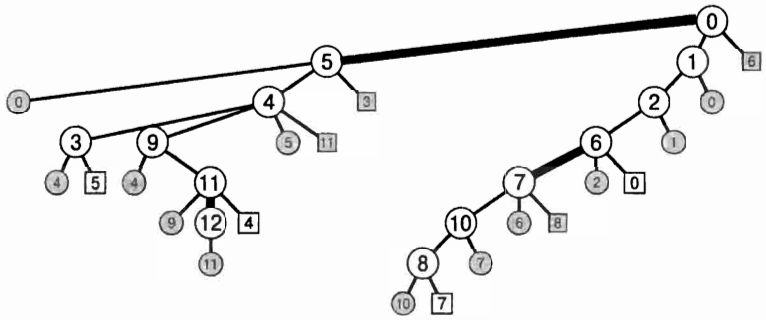


РИСУНОК 18.16. РЕБЕРНО-ОТДЕЛИМЫЙ ГРАФ

*Этот граф не относится к числу реберно-связных. Ребра **0-5**, **6-7** и **11-12** (заштрихованы) представляют собой ребра отделимости (мосты). Граф содержит четыре реберно-связных компоненты: одна включает вершины **0**, **1**, **2** и **6**; другая – вершины **3**, **4**, **9** и **11**; следующая – вершины **7**, **8** и **10**; последняя состоит из вершины **12**.*

На первый взгляд выявление мостов в графе является нетривиальной задачей обработки графов, но на самом деле для ее решения достаточно применить алгоритм DFS, в рамках которого используются базовые свойства деревьев DFS, о которых речь шла выше. В частности, обратные ребра не могут быть мостами, поскольку, как мы знаем, те пары узлов, которые они связывают, соединены некоторым путем в дереве DFS. Более того, мы просто можем добавить в рекурсивную функцию условие для проверки, являются ли ребра дерева мостами. Иллюстрацией основной идеи, строгая формулировка которой дается ниже, может служить рис. 18.17.



	0	1	2	3	4	5	6	7	8	9	10	11	12
ord	0	7	8	3	2	1	9	10	12	4	11	5	6
low	0	0	0	1	1	1	0	10	10	2	10	2	6

РИСУНОК 18.17. ДЕРЕВО DFS, ИСПОЛЬЗУЕМОЕ ДЛЯ ПОИСКА МОСТОВ

*Узлы **5**, **7** и **12** рассматриваемого дерева DFS для графа на рис. 18.16 обладают тем свойством, что никакое обратное ребро не соединяет потомка с предком, и этим свойством не обладают никакие другие узлы. Поэтому, как было указано выше, удаление ребра между одним из этих узлов и их предком отсоединит поддерево с корнем в этом узле от остальной части графа. То есть, ребра **0-5**, **11-12** и **6-7** суть мосты. Мы используем массив **low**, проиндексированный именами вершин, для отслеживания минимального номера при обходе вершин в прямом порядке (значение **ord**), на который ссылается любое обратное ребро в поддереве, корнем которого является эта вершина. Например, значением **low[9]** будет **2**, поскольку одно из обратных ребер в поддереве с корнем в **9** указывает на **4** (вершина, номер которой при обходе в прямом порядке есть **2**), и никакое другое обратное ребро не указывает на вершину более высокого уровня в этом дереве. Узлы **5**, **7** и **12** – это узлы, для которых значения **low** равны значению **ord**.*

Свойство 18.5. *В любом дереве DFS ребро* **v-w** *есть мост тогда и только тогда, когда не существует обратное ребро, которое соединяет один из потомков* **w** *с каким-либо предком* **w**.

Доказательство: Если существует такое ребро, то **v-w** не может быть мостом. С другой стороны, если **v-w** не есть мост, то в графе должен быть другой путь из **w** в **v**, отличный от **w-v**. Каждый такой путь должен содержать одно из таких ребер. ■

Провозглашение этого свойства эквивалентно утверждению, что единственная связь поддерева с корнем в **w**, ведущая в узел, который не входит в это поддерево, есть родительская связь, ведущая из **w** назад в **v**. Это условие соблюдается тогда и только тогда, когда каждый путь, соединяющий любой узел в поддереве узла **w**, с любым узлом, не принадлежащим поддереву узла **w**, включает **v-w**. Другими словами, удаление **v-w** отделяет подграф, соответствующий поддереву узла **w**, от остальной части графа.

Программа 18.7 показывает, как можно усовершенствовать поиск в глубину с таким расчетом, чтобы он мог выявлять в графах мосты, используя для этой цели программу 18.5. Для каждой вершины **v** мы используем рекурсивную функцию, вычисляющую минимальный номер в прямом порядке обхода, на который можно выйти через последовательность из нулевого или большего числа ребер дерева, за которыми следует одно обратное ребро из любого узла поддерева с корнем в вершине **v**. Если вычисленное число равно номеру вершины **v** при прямом порядке обхода, то не существует ребра, связывающего потомок вершины **v** с ее предком, а это означает, что обнаружен мост. Вычисления для

каждой вершины достаточно просты: мы просматриваем списки смежных вершин, следя за тем, на какое минимальное число мы можем выйти, следуя по каждому ребру. Для древесных ребер вычисления выполняются в рекурсивном режиме; для обратных ребер мы используем номер смежной вершины в прямом порядке. Если обращение к рекурсивной функции для ребра **w-t** не приводит к обнаружению пути к узлу с меньшим номером прямого порядка обхода, чем аналогичный номер узла **t**, то ребро **w-t** является мостом.

Свойство 18.6. *Мосты графа обнаруживаются за линейное время*

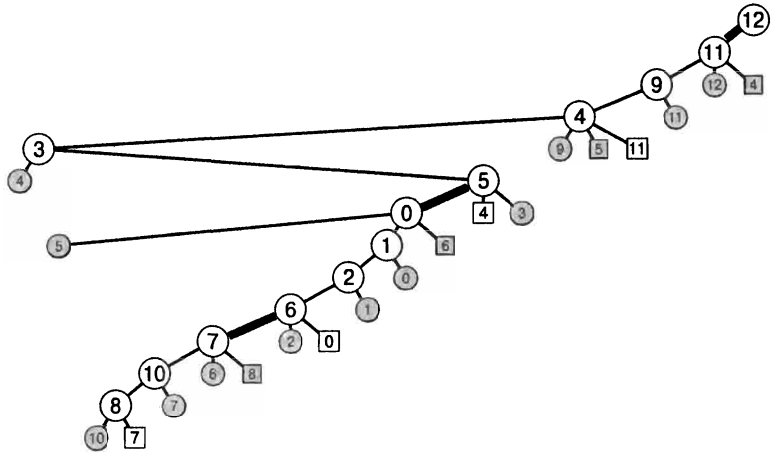
Доказательство: Программа 18.7 представляет собой незначительную модификацию поиска в глубину, при этом добавляются несколько проверок, требующие постоянных затрат времени. В результате из свойств 18. 3 и 18.4 непосредственно следует, что поиск мостов в графе требует время, пропорциональное V^2 для представления графа в виде матрицы смежности и $V + E$ для представления графа в виде списков смежных вершин. ■

Программа 18.7. Реберная связность

Класс DFS производит подсчет мостов заданного графа. Клиентская программа может использовать объект **EC** для выявления количества реберно-связных компонент. Добавление функции-элемента, осуществляющей проверку, содержатся ли какие-либо две вершины в одной и той же реберно-связной компоненте, мы оставляем на самостоятельную проработку (упражнение 18.36). Вектор **low** отслеживает минимальный номер вершины при прямом порядке обхода вершин графа, который может быть достигнут из каждой вершины через некоторую последовательность древесных ребер, за которой следует обратное ребро.

```
template <class Graph>
class EC : public SEARCH<Graph>
{ int bcnt;
   vector <int> low;
   void searchC(Edge e)
   { int w = e.w;
      ord[w] = cnt++; low[w] = ord[w];
      typename Graph::adjIterator A(G, w);
      for (int t = A.beg(); !A.end(); t = A.nxt())
         if (ord[t] == -1)
           {
             searchC(Edge(w, t));
             if (low[w] > low[t]) low[w] = low[t];
             if (low[t] == ord[t])
               bcnt++; // w-t является мостом
           }
         else if (t != e.v)
           if (low[w] > ord[t]) low[w] = ord[t];
   }
public:
   EC(const Graph &G) : SEARCH<Graph>(G),
      bcnt(0), low(G.V(), -1)
      { search(); }
   int count() const { return bcnt+1; }
};
```

В программе 18.7 мы используем поиск в глубину для исследования свойств графа. Разумеется, представление графа оказывает влияние на порядок поиска, но оно никак не влияет на результаты этих исследований, ибо мосты — это характерные особенности гра-



	0	1	2	3	4	5	6	7	8	9	10	11	12
ord	6	7	8	4	3	5	9	10	12	2	11	1	0
low	6	6	6	3	1	3	6	10	10	1	10	1	0

РИСУНОК 18.18. ДРУГОЕ ДЕРЕВО DFS, ИСПОЛЬЗУЕМОЕ ДЛЯ ПОИСКА МОСТОВ

На данной диаграмме показано дерево DFS, отличное от дерева, изображенного на рис. 18.17 и построенного для графа, показанного на рис. 18.16, тогда мы начинали поиск с другого узла. Несмотря на то что мы просматриваем узлы и ребра в совершенно другом порядке, мы, тем не менее, обнаруживаем одни и те же мосты (что вполне естественно). На этом дереве вершины ***0, 7*** *и* ***11*** *— это вершины, для которых значение* ***low*** *равно значению* ***ord****, так что ребра, соединяющие каждую из таких вершин с их родителями (соответственно,* ***12-11, 5-0*** *и* ***6-7****) являются мостами.*

фа, а не выбранный нами способ представления графа или поиска на графе. Как и всегда, любое дерево DFS — это просто еще одно представление графа, так что все такие деревья обладают одними и теми же свойствами связности. Корректность алгоритма зависит от этого фундаментального факта. Например, рис. 18.18 представляет собой иллюстрацию другого поиска на том же графе, он начинается с другой вершины и, естественно, обнаруживает те же мосты. Вопреки свойству 18.6, в тех случаях, когда мы исследуем различные деревья DFS одного и того же графа, мы убеждаемся, что стоимость того или иного поиска может зависеть не только от свойств графа, но также и от свойств дерева DFS. Например, пространство памяти, необходимое для стека, который обеспечивает поддержку рекурсивных вызовов, больше для примера, представленного на рис.18.18, чем для примера на рис. 18.17.

Так же, как мы поступили в отношении регулярной связности в программе 18.4, мы, возможно, воспользуемся программой 18.7 для построения класса, выполняющего проверку, является ли заданный граф реберно-связным, либо подсчитывающего число реберно-связных компонент. При необходимости мы можем поступать так же, как и в случае программы 18.4, которая предоставляет клиентам возможность создавать (за линейное время) объекты, способные за постоянное время отвечать на запросы на предмет того, находятся ли две заданные вершины в одной и той же реберно-связной компоненте (см. упражнение 18.36).

Мы завершим этот раздел анализом других видов обобщений понятия связности, включая задачу определения *вершин*, критичных для сохранения связности графа. Включая эти

сведения в данный раздел, мы держим в одном месте основной материал, необходимый для изучения сложных алгоритмов, которые будут рассматриваться в главе 22. Если вы впервые сталкиваетесь с задачами связности графов, вы можете сейчас пропустить раздел 18.7 и вернуться к нему, когда начнете изучать главу 22.

Когда речь идет об *удалении вершины*, то под этим подразумевается удаление всех инцидентных ей ребер. Как показано на рис. 18.19, удаление любой из вершин моста влечет потерю графом свойства связности (если, конечно, этот мост был единственным ребром, инцидентным одной или обеим вершинам), в то же время существуют другие вершины, не связанные с мостами, для которых характерно это же свойство.

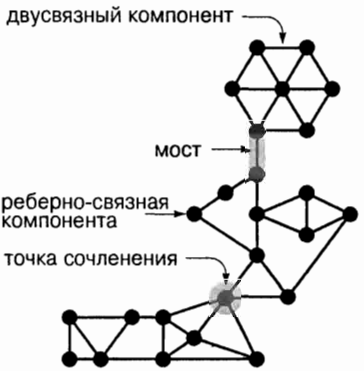


РИСУНОК 18.19. ТЕРМИНОЛОГИЯ ОТДЕЛИМОСТИ ГРАФА

Этот граф состоит из двух реберно-связных компонент и одного моста. К тому же реберно-связная компонента, расположенная над мостом, является двусвязной; компонента, расположенная ниже моста, состоит из двух двусвязных компонент, соединенных друг с другом в точке сочленения.

Определение 18.2. Точка сочленения (articulation point) *графа есть вершина, в случае удаления которой связный граф распадается, по меньшей мере, на два непересекающихся подграфа.*

Мы будем также называть точки сочленения графа *вершинами отделимости* (*separation vertices*) или *разрезающими вершинами* (*cut vertex*). Мы могли бы воспользоваться термином "связанные вершины" для описания графа, в котором нет вершин отделимости, но мы воспользуемся другой терминологией, основанной на родственных понятиях, которые в конечном итоге оказываются эквивалентной терминологией.

Определение 18.3. *Граф называется* **двусвязным (biconnected)**, *если каждая пара вершин соединена двумя непересекающимися путями.*

Требование *непересекающихся* (*disjoint*) путей отличает двухвязность от реберной связности. Альтернативное определение реберной связности отличается тем, что каждая пара вершин связана двумя путями, в которых никакие составляющие его ребра не пересекаются, – эти пути могут иметь общие вершины, но отнюдь не ребра. Двусвязность представляет собой более сильное условие: реберно-связный граф остается связным, если мы удалим из него какую-нибудь вершину (и все ребра, инцидентные этой вершине). Каждый двусвязный граф есть реберно-связный граф, однако реберно-связный граф не обязательно должен быть двусвязным. Мы также пользуемся термином *сепарабельный (separable)* в отношении к графу, который не является двусвязным, поскольку за счет удаления всего лишь одной вершины он может быть разделен на две части. Наличие вершин отделимости являются ключевым свойством двусвязности.

Свойство 18.7. *Граф двусвязен тогда и только тогда, когда он не содержит вершин отделимости (точек сочленения).*

Доказательство: Предположим, что в графе имеется вершина отделимости. Пусть s и t – две вершины, которые окажутся в двух различных частях графа, если будет уда-

лена вершина отделимости. Все пути, связывающие *s* и *t*, должны проходить через вершину отделимости, вследствие чего рассматриваемый граф не может быть двусвязным. Доказательство в обратном направлении намного труднее и может быть предложено в качестве упражнения математически подготовленным читателям (см. упражнение 18.40). ■

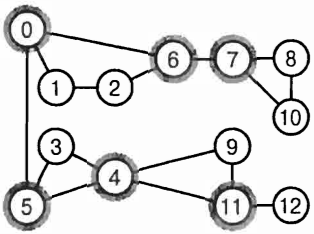


РИСУНОК 18.20. ТОЧКИ СОЧЛЕНЕНИЯ (ВЕРШИНЫ ОТДЕЛИМОСТИ)

Данный граф не принадлежит к числу двусвязных графов. Вершины ***0, 4, 5, 6, 7*** *и* ***11*** *(заштрихованы) представляют собой точки сочленения. Рассматриваемый граф содержит пять двусвязных компонент: одна из них состоит из ребер* ***4-9, 9-11*** *и* ***4-11****, другая – из ребер* ***7-8, 8-10*** *и* ***7-10****; еще одна – из ребер* ***0-1, 1-2, 2-6*** *и* ***6-0****; следующая – из ребер* ***3-5, 4-5*** *и* ***3-4****; и наконец, одиночная вершина* ***12****. Если добавить в граф ребро, соединяющее вершину* ***12*** *с вершинами* ***7, 8*** *или* ***10****, получается двусвязный граф.*

Мы убедились в том, что можем разбить множество ребер несвязного графа на совокупность связных подграфов и что можем разбить ребра графа, не относящегося к категории реберно-связных, на множество мостов и реберно-связных подграфов (которые соединены между собой мостами). Аналогично, мы можем разбить любой граф, не относящийся к числу двусвязных, на множество мостов и *двусвязных компонент* (*biconnected components*), каждая из которых представляет собой двусвязный подграф. Двусвязные компоненты и мосты нельзя рассматривать как подходящее разбиение графа, поскольку точки сочленения могут входить во многие связные компоненты (см., например, рис. 18.20). Двусвязные компоненты присоединены к графу в точках расчленения, возможно, через мосты.

Связная компонента графа обладает тем свойством, что существует путь между любыми двумя вершинами графа. Аналогично, для двусвязной компоненты характерно то, что между любой парой вершин существуют два непересекающихся пути.

Можно использовать тот же подход, основанный на алгоритме DFS, который применялся в программе 18.7 для определения, является ли тот или иной граф двусвязным или нет, и для выявления точек сочленения. Мы не будем рассматривать соответствующий программный код, поскольку он во многом идентичен программе 18.7, за исключением разве что проверки, не является ли корень дерева DFS точкой сочленения (см. упражнение 18.44). Разработка программного кода, выполняющего распечатку двусвязных компонент графа, также может послужить хорошим упражнением, которое лишь немного сложнее, чем соответствующая программа для определения реберной связности графа (см. упражнение 18.44).

Свойство 18.8. *Точки сочленения и двусвязные компоненты графа можно обнаружить за линейное время.*

Доказательство: Как и в случае свойства 18.7, этот факт непосредственно следует из того, что решение упражнений 18.43 и 18.44 предусматривает внесение некоторых небольших изменений в поиск в глубину, из которых вытекает необходимость нескольких проверок каждого ребра, выполняемых за постоянное время. ■

Определение 18.4. *Граф называется* **k-связным (k-connected)**, *если существуют, по меньшей мере, k непересекающихся по вершинам путей, соединяющих каждую пару вершин графа.* **Вершинная связность (vertex connectivity)** *графа есть минимальное число вершин, которые нужно удалить, чтобы разделить этот граф на две части.*

В этой терминологии "1-связный" есть ни что иное как "связный", а "2-связный" – это то же, что и "двусвязный". Граф с точкой сочленения обладает вершинной связностью, равной 1 (или 0), так что, как утверждает свойство 18.7, граф 2-связен тогда и только тогда, когда значение его вершинной связности не меньше 2. Особый случай классического результата теории графов, известный как *теорема Уитни* (*Whitney's theorem*) утверждает, что граф k-связен в том и только том случае, когда его вершинная связность не меньше k. Теорема Уитни непосредственно следует из теоремы Менгера (*Menger's theorem*) (см. раздел 22.7), согласно которой минимальное число вершин, удаление которых приводит к утрате связи между некоторыми двумя вершинами, равно максимальному числу путей, не имеющих общих вершин (чтобы доказать теорему Уитни, нужно применить теорему Менгера к каждой паре вершин).

Определение 18.5. *Граф называется* **k–реберно-связным (k–edge-connected)**, *если существуют, по меньшей мере, k путей, которые не имеют общих ребер, соединяющих каждую пару вершин графа.* **Реберная связность (edge connectivity)** *графа есть минимальное число ребер, которые нужно удалить, чтобы разделить этот граф на две части.*

В этой терминологии "1–реберно-связный" есть ни что иное как "реберно-связный" (т.е. для реберно-связного графа значение реберной связности не меньше 1). Другой вариант теоремы Менгера утверждает, что минимальное число вершин в графе, удаление которых приводит к разрыву связи между некоторыми двумя вершинами графа, равно максимальному числу путей, не имеющих общих вершин и связывающих эти две вершины графа. Отсюда следует, что рассматриваемый граф k–реберно-связен тогда и только тогда, когда его реберная связность равна k.

Имея в своем распоряжении приведенные определения, можно приступить к обобщению задач определения связности, которые рассматривались в начале данного раздела.

st-связность. Каким является минимальное число ребер, удаление которых приведет к разделению двух конкретных вершин s и t заданного графа? Чему равно минимальное число вершин, удаление которых приведет к разделению двух заданных вершин s и t заданного графа?

Общая связность. Является ли заданный граф k-связным? Является ли заданный граф k-реберно-связным? Какой является реберная связность и вершинная связность заданного графа?

И хотя решения всех этих задач намного сложнее, чем решения простых задач связности, рассмотренных в данном разделе, они, тем не менее, входят в обширный класс задач обработки графов, которые мы можем решать с помощью универсальных алгоритмических средств, изучением которых займемся в главе 22 (при этом важную роль играет поиск в глубину); конкретные решения будут изучаться в разделе 22.7.

Упражнения

▷ **18.33**. Если граф является лесом, все его ребра суть разделяющие ребра. В таком случае, какие вершины представляют собой разделяющие вершины?

▷ **18.34**. Рассмотрим граф

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

Вычертите дерево стандартного DFS на списках смежных вершин. Воспользуйтесь им для поиска мостов и реберно-связных компонент.

18.35. Докажите, что каждая вершина любого графа принадлежит в точности одной реберно-связной компоненте.

○ **18.36**. Добавьте общедоступную функцию-элемент в программу 18.7, которая позволила бы клиентским программам проверять, находится ли та или иная пара вершин в одной и той же реберно-связной компоненте.

▷ **18.37**. Рассмотрим граф

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

Вычертите дерево стандартного DFS на списках смежных вершин. Воспользуйтесь им для поиска точек сочленения и двусвязных компонент.

▷ **18.38**. Выполните предыдущее упражнение с применением дерева стандартного DFS на списках смежных вершин.

18.39. Докажите, что каждое ребро графа есть либо мост, либо оно принадлежит в точности одной двусвязной компоненте.

● **18.40**. Докажите, что любой граф без точек сочленения является двусвязным. *Указание*: Если задана пара вершин s и t и путь, который их соединяет, воспользуйтесь тем фактом, что ни одна из вершин этого пути не есть точка сочленения при построении двух непересекающихся путей, соединяющих вершины s и t.

18.41. Пользуясь программой 18.2, постройте класс для определения, является ли граф двусвязным, воспользовавшись алгоритмом решения "в лоб", который выполняется за время, пропорциональное $V(V+E)$. *Указание*: Если перед началом поиска отметить ту или иную вершину как уже просмотренную, ее можно эффективно удалить из графа.

○ **18.42**. Распространите свое решение на упражнение 18.41 с тем, чтобы построить класс, который определяет, является ли заданный граф 3-связным. Выведите формулу, позволяющую получить приближенное значение числа раз, когда ваша программа исследует ребро графа, как функцию от V и E.

18.43. Покажите, что корень дерева DFS есть точка сочленения тогда и только тогда, когда у него имеется два или большее число (внутренних) потомков.

● **18.44**. Пользуясь программой 18.2, постройте класс, который выполняет распечатку двусвязных компонент графа.

18.45. Чему равно минимальное число ребер, которое должно содержаться в любом двусвязном графе с V вершинами?

18.46. Внесите изменения в программу 18.7 с тем, чтобы упростить задачу определения, обладает ли заданный граф свойством реберной связности (возврат, как только она обнаружит мост, если граф не реберно-связный), и снабдите ее инструментальными средствами, позволяющими отслеживать число исследованных ребер. Проведите эмпирическую проверку с целью определения связанных с этим затрат на примерах графов различных размеров, в основу которых положены различные модели графов (упражнения 17.64–17.76).

○ **18.47**. На основе программы 18.2 постройте производный класс, позволяющий клиентским программам создавать объекты, которым известны числа точек сочленения, мостов и двусвязных компонентов графа.

● **18.48**. Выполните эксперимент с целью определения эмпирическим путем средних значений количественных оценок, описанных в упражнении 18.47, для графов различных размеров, в основу которых положены различные модели графов (упражнения 17.64–17.76).

18.49. Определите реберную совместимость и вершинную совместимость графа

0-1 0-2 0-8 2-1 2-8 8-1 3-8 3-7 3-6 3-5 3-4 4-6 4-5 5-6 6-7 7-8.

18.7. Поиск в ширину

Предположим, что мы хотим найти *кратчайший путь* (*shortest path*) между двумя конкретными вершинами некоторого графа – путь, соединяющий вершины, обладающие тем свойством, что никакой другой путь, соединяющий эти вершины, не содержит меньшее число ребер. Классический метод решения этой задачи, получивший название *поиска в ширину* (*BFS – breath-first search*), служит основой многочисленных алгоритмов обработки графов; именно он и будет изучаться в данном разделе. Поиск в глубину мало пригоден для решения этой задачи, поскольку предлагаемый им порядок прохождения графа не имеет отношения к поиску кратчайших путей. В отличие от поиска в глубину, поиск в ширину предназначен как раз для достижения этой цели. Поиск кратчайшего пути от вершины **v** к вершине **w** мы начнем с того, что среди всех вершин, в которые можно перейти по одному ребру из вершины **v**, мы попытаемся обнаружить вершину **w**, затем мы проверяем все вершины, в которые мы можем перейти по двум ребрам, и т.д.

Когда во время просмотра графа мы попадаем в такую точку, из которой исходят более одного ребра, мы выбираем одно из них и запоминаем остальные для дальнейшего просмотра. В поиске в глубину для этой цели мы используем стек магазинного типа (которым управляет система, благодаря чему обеспечивается поддержка рекурсивной функции поиска). Применение правила LIFO (Last In First Out – последним пришел, первым обслужен), которое характеризует работу стека магазинного типа, соответствует исследованию соседних коридоров в лабиринте: из всех еще не исследованных коридоров выбирается последний из тех, с которым мы столкнулись. В поиске в ширину мы хотим проводить исследование вершин в зависимости от их удаления от исходной точки. В случае реального лабиринта для проведения исследований в таком порядке может потребоваться специальная команда исследователей; однако в компьютерной программе эта цель достигается намного проще: мы просто вместо стека используем *очередь FIFO* (*FIFO queue* – первым пришел, первым обслужен).

Программа 18.8 представляет собой реализацию поиска в ширину. В ее основе лежит поддержка очереди всех ребер, которые соединяют посещенные вершины с непосещенными. Для исходной вершины мы помещаем в очередь фиктивную петлю, после чего выполняем следующие действия до тех пор, пока очередь не опустеет:

- Выбираем ребра из очереди до тех пор, пока не найдем такое ребро, которое ведет на непосещенную вершину.
- Просматриваем эту вершину; ставим в очередь все ребра, исходящие из этой вершины в вершины, которые мы еще не посещали.

Рисунок 18.21 иллюстрирует последовательный процесс поиска в ширину (BFS) на конкретном примере.

Ребро **7-4** показано серым цветом, поскольку мы могли бы и не устанавливать его в очередь, так как имеется еще одно ребро, которое ведет в вершину **4**, уже помещенную в очередь. В завершение поиска мы удаляем оставшиеся ребра из очереди, полностью игнорируя при этом серые ребра, когда они вверху очереди (справа). Ребра поступают в очередь и покидают ее в порядке их удаленности от вершины **0**.

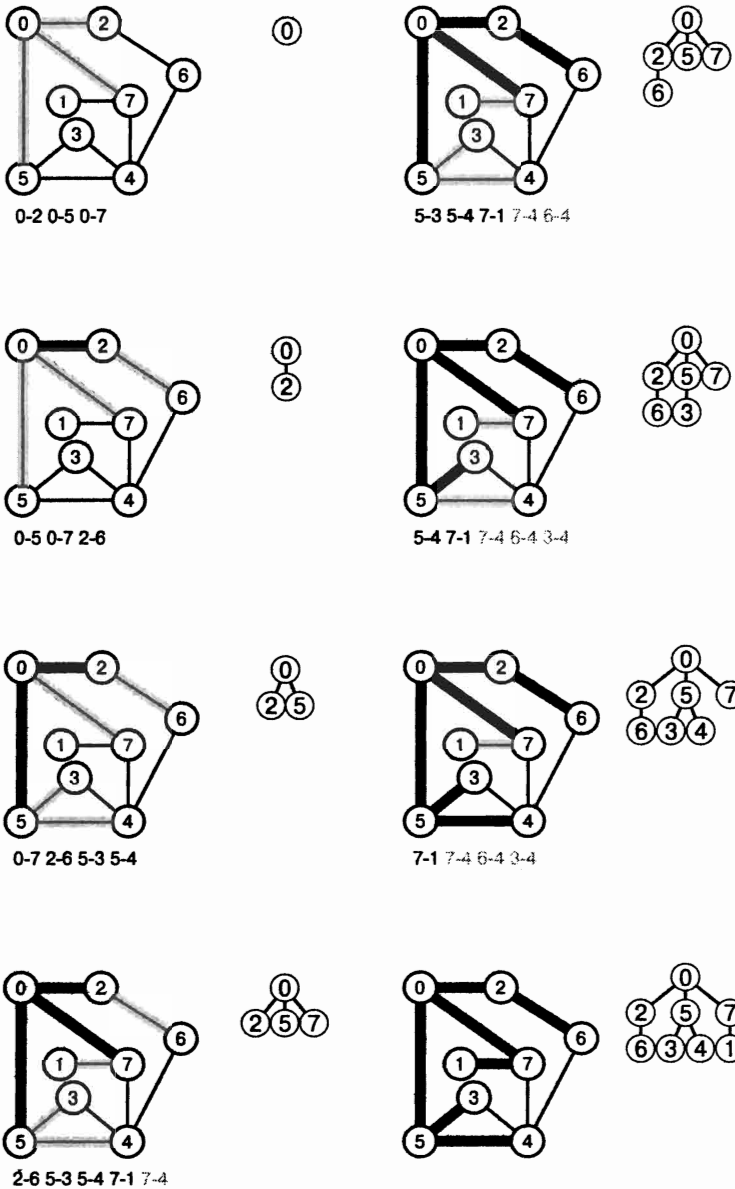


РИСУНОК 18.21. ПОИСК В ШИРИНУ

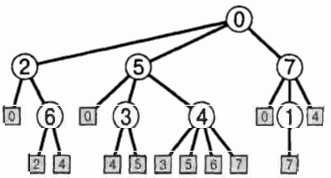
Этот рисунок прослеживает процесс поиска в ширину на заданном примере. Мы начинаем его на всех ребрах, смежных с исходной вершиной в очереди (диаграмма вверху слева). Далее, мы переносим ребро 0-2 из очереди в дерево и приступаем к обработке инцидентных ему ребер 2-0 и 2-6 (вторая диаграмма сверху слева). Мы не помещаем ребро 2-0 в очередь, поскольку вершина 0 уже содержится в дереве. Затем мы переносим ребро 0-5 из очереди в дерево; и снова ребра, инцидентные вершине 5, не приводят нас к новым вершинам, однако мы добавляем в очередь ребра 5-3 и 5-4 (третья диаграмма сверху слева). После этого мы добавляем ребро 0-7 в дерево и устанавливаем ребро 7-1 в очередь (диаграмма внизу слева).

Как мы могли убедиться в разделе 18.4, поиск в глубину подобен исследованию лабиринта, проводимого одним человеком. Поиск в ширину можно сравнить с исследованием, проводимым группой людей, рассыпавшихся веером по всем направлениям. Несмотря на то что поиски в глубину и в ширину отличаются друг от друга во многих аспектах, здесь уместно подчеркнуть то, что между этими двумя методами существует прочная глубинная связь – такая, какую мы обнаружили, когда проводили краткий анализ методов в главе 5. В разделе 18.8 мы рассмотрим обобщенный метод поиска на графе, который мы можем переделать таким образом, чтобы он включал в себя оба эти алгоритма, равно как и множество других. Каждый алгоритм обладает собственными динамическими характеристиками, которые мы используем для решения соответствующих задач обработки графов. Что касается поиска в ширину, то наибольший интерес для нас представляет расстояние каждой вершины от исходной вершины (длина кратчайшего пути, соединяющего две эти вершины).

Свойство 18.9. *В процессе поиска в ширину вершины поступают в очередь FIFO и покидают ее в порядке, определяемом их расстоянием от исходной вершины.*

Доказательство: Справедливо более сильное условие: очередь всегда содержит ноль или большее число вершин, удаленных на k из исходной точки, за которой следует ноль или большее число вершин, удаленных на $k + 1$ от исходной точки, где k – некоторое целое значение. Это более сильное условие легко доказывается методом индукции. ■

Что касается поиска в глубину, то понимание динамических характеристик этого алгоритма достигается с помощью леса поиска DFS, который описывает структуру рекурсивных вызовов алгоритма. Основное свойство упомянутого леса состоит в том, что он представляет пути из каждой вершины в точку, откуда начался поиск связной компоненты, в которой она находится. Как показывает реализация и рис.18.22, такое остовное дерево помогает достичь понимания поиска в ширину. Как и в случае поиска в глубину, мы имеем лес, который характеризует динамику поиска, по одному дереву на каждую связную компоненту, один узел дерева приходится на каждую верши-



	0	1	2	3	4	5	6	7
ord	0	7	1	5	6	2	4	3
st	0	7	0	5	5	0	2	0

РИСУНОК 18.22. ДЕРЕВО BFS

*Это дерево представляет собой компактное описание динамических характеристик поиска в глубину в стиле, свойственном дереву, которое показано на рис. 18.9. Обход дерева в порядке уровней показывает нам, как разворачивается поиск: сначала мы наносим визит вершине **0**, потом мы посещаем вершины **2**, **5** и **7**, затем, находясь в **2**, мы выясняем, что в вершине **0** мы уже были, и направляемся в **6** и т.д. У каждого узла дерева имеются потомки, и каждый из них представляет узлы, смежные с этим узлом, в том порядке, в каком они рассматриваются алгоритмом BFS. Как и на рис. 18.9, связи дерева BFS соответствуют ребрам графа: если мы заменим ребра, ведущие во внешние узлы, на линии, ведущие в заданный узел, то мы получим чертеж графа. Связи, ведущие во внешние узлы, представляют собой ребра, которые не были помещены в очередь, так как они направлены в помеченные узлы: это либо родительские связи, либо перекрестные связи, которые указывают на некоторый узел, находящийся на том же уровне или на уровне, более близком к корню дерева.*

*Вектор **st** есть представление дерева в виде родительских связей, которое мы можем использовать при поиске кратчайшего пути из любого узла в корень. Например, **3-5-0** представляет собой путь в графе из **3** в **0**, поскольку **st[3]** есть **5**, а **st[5]** – **0**. Никакой другой путь из **3** в **0** не может быть короче.*

ну графа и одно ребро дерева — на каждое ребро графа. Поиск в ширину соответствует обходу каждого дерева этого леса в порядке уровней. Как и в случае поиска в глубину, мы используем вектор, индексированный именами вершин для точного представления леса с родительскими связями. Этот лес содержит важную для поиска в ширину информацию о структуре графа.

Свойство 18.10. *Для любого узла w в дереве BFS с корнем в вершине v путь из v в w соответствует кратчайшему пути из v в w в соответствующем графе.*

Доказательство: Длины путей в дереве из узлов, выходящих из очереди, в корень дерева представляют собой неубывающую последовательность, при этом в очереди находятся все узлы, расположенные ближе к корню, чем *w*; следовательно, не удалось найти более короткого пути до *w до* того, как она выйдет из очереди, и никакие пути в *w после* того, как она выйдет из очереди, не могут быть короче, чем длина пути в вершину *w* в дереве. ■

Программа 18.8. Поиск в ширину

Данный класс поиска на графе посещает вершину на основании просмотра инцидентных ей ребер, помещая все ребра, ведущие на непосещенную вершину, в очередь вершин, планируемых для просмотра. Вершины маркируются в порядке их посещения вектором **ord**. Функция **search**, вызываемая конструктором, выполняет построение явного представления дерева BFS с родительскими связями (ребра, которые приводят нас на каждый узел в первый раз) в другом векторе **st**, который затем может быть использован для решения базовой задачи поиска кратчайшего пути (см. текст).

```
#include "QUEUE.cc"
template <class Graph>
class BFS : public SEARCH<Graph>
{ vector<int> st;
  void searchC(Edge e)
  { QUEUE<Edge> Q;
    Q.put(e);
    while (!Q.empty())
      if (ord[(e = Q.get()).w] == -1)
      { int v = e.v, w = e.w;
        ord[w] = cnt++; st[w] = v;
        typename Graph::adjIterator A(G, w);
        for (int t = A.beg(); !A.end(); t = A.nxt())
          if (ord[t] == -1) Q.put(Edge(w, t));
      }
  }
public:
  BFS(Graph &G) : SEARCH<Graph>(G), st(G.V(), -1)
    { search(); }
  int ST(int v) const { return st[v]; }
};
```

Программа 18.9. Усовершенствованный алгоритм BFS

Чтобы в очереди, которая используется во время выполнения BFS, было не более *V* мест, мы маркируем вершины в момент постановки их в очередь.

```
void searchC(Edge e)
{ QUEUE<Edge> Q;
  Q.put(e); ord[e.w] = cnt++;
```

```
    while (!Q.empty())
      {
          e = Q.get(); st[e.w] = e.v;
          typename Graph::adjIterator A(G, e.w);
          for (int t = A.beg(); !A.end(); t = A.nxt())
            if (ord[t] == -1)
               { Q.put(Edge(e.w, t)); ord[t] = cnt++; }
      }
}
```

Как показано на рис. 18.21 и отмечено в главе 5, нет необходимости помещать в очередь ребро с такой же вершиной назначения, что и у хотя бы одного ребра, которое уже находится в очереди, поскольку правило FIFO гарантирует, что мы выполним обработку ребра, которое уже находится в очереди (и нанесем визит в соответствующую вершину), раньше, чем приступим к обработке нового в очереди ребра. Один из способов реализации такой стратегии заключается в том, чтобы использовать реализацию АТД очереди, по условиям которой такое дублирование запрещается стратегией игнорирования новых элементов (см. раздел 4.7). Другой способ предусматривает применение для этой цели глобального вектора маркирования вершин: вместо того, чтобы маркировать вершину в момент ее *исключения* из очереди как посещенную, мы делаем это в момент *включения* ее в очередь. Обнаружение у той или иной вершины такой маркировки (т.е., изменилось ли значение соответствующего ее элемента относительно начального сигнального значения) означает запрет на включение в очередь каких-либо других ребер, которые указывают на эту вершину. Это изменение, показанное в программе 18.9, позволяет получить программную реализацию поиска в ширину, по условиям которой в очереди никогда не бывает более V ребер (в каждую вершину ведет максимум одно ребро).

Свойство 18.11. *Поиск в ширину посещает все вершины и ребра графа за время, пропорциональное V^2, в случае представления графа в виде матрицы смежности, и за время, пропорциональное $V + E$, в случае представления графа в виде списков смежных вершин.*

Доказательство: Подобно тому, как мы поступали при доказательстве аналогичных свойств поиска в глубину, путем анализа программного кода мы обнаруживаем, что проверяем каждый элемент строки матрицы смежности или списка смежных вершин в точности один раз для каждой вершины, которую мы посещаем, следовательно, достаточно показать, что мы посетили каждую вершину. Теперь же для каждой связной компоненты рассматриваемый алгоритм сохраняет следующие инварианты: все вершины, на которые можно выйти из исходной вершины, (*i*) включены в дерево BFS, (*ii*) поставлены в очередь или (*iii*) достижимы из одной из вершин, установленных в очередь. Каждая вершина перемещается из (*iii*) в (*ii*) и в (*i*), а число вершин в (*i*) увеличивается при каждой итерации цикла; таким образом, в конечном итоге дерево BFS будет содержать все вершины, на которые можно выйти из исходной вершины. Это дает нам, как и в случае поиска в глубину, основание утверждать, что поиск в ширину представляет собой алгоритм, линейный по времени выполнения. ■

С помощью поиска в ширину мы можем решить задачи обнаружения остовного дерева, связных компонент, поиска вершин и ряд других базовых задач связности, которые были сформулированы и описаны в разделе 18.4, поскольку рассмотренные решения зависят только от способности алгоритма поиска анализировать каждый узел и каждое ребро, связанное с исходной точкой. Как мы скоро убедимся, поиски в ширину и в глубину

являются шаблонами многочисленных алгоритмов, обладающих этим свойством. Как уже отмечалось в начале данного раздела, наш повышенный интерес к поиску в ширину объясняется тем, что он является естественным алгоритмом поиска на графе для приложений, в условиях которых требуется знать кратчайший путь между двумя заданными вершинами. Далее, мы рассмотрим конкретное решение этой задачи и его расширение, способное решать две другие родственные задачи.

Кратчайший путь. Найти кратчайший путь на графе, ведущий из *v* в *w*. Мы можем решить эту задачу, инициируя процесс поиска в ширину, который поддерживает в вершине *v* представление **st** дерева поиска в виде родительских связей, прекращая этот процесс по достижении вершины *w*. Путь из *w* в *v* вверх по дереву и является самым коротким. Например, после построения объекта **bfs** класса **BFS<Graph>** может использовать следующий программный код для распечатки пути, ведущего из *w* в *v*:

```
for (t = w; t !=w; t = bfs.ST(t)) cout << t << "-";
cout << v << endl;
```

Чтобы получить путь *v* в *w*, замените в этом узле операцию **cout** на операцию заталкивания индексов в стек, затем войдите в цикл, который распечатывает индексы этих вершин после выталкивания их из стека. Либо начните поиск в *w* и остановите его тогда, когда *v* окажется на первом месте.

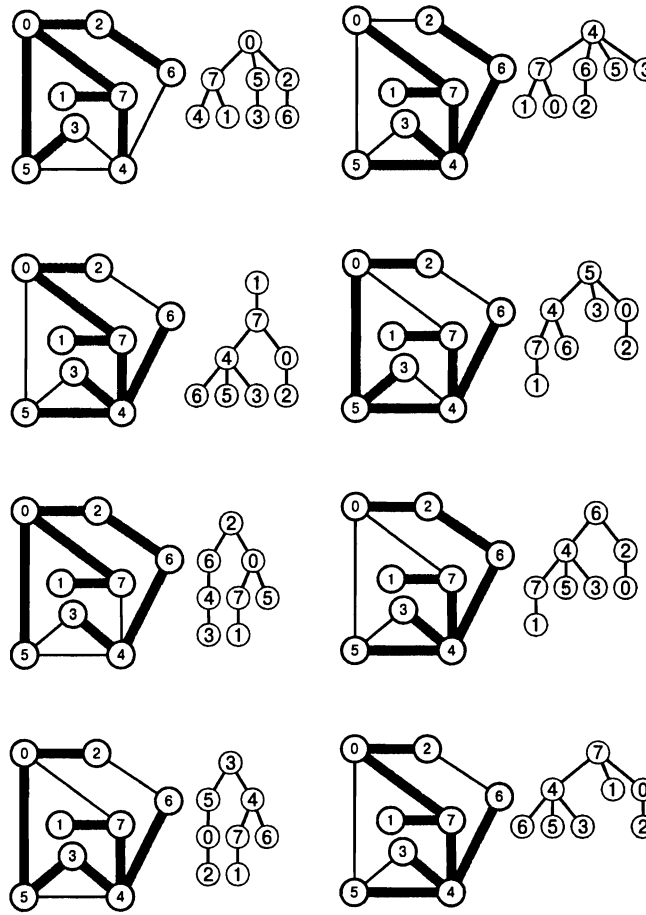
Кратчайший путь из единственного источника. Найти кратчайшие пути, соединяющие заданную вершину *v* со всеми другими вершинами графа. Полное дерево с корнем в вершине *v* позволяет решить эту задачу: путь из каждой вершины, ведущий в корень есть кратчайший путь в корень. Поэтому, чтобы решить эту задачу, мы выполняем поиск в ширину до завершения, начиная с *v*. Вектор **st**, который появляется в результате таких вычислений, есть представление дерева BFS в виде совокупности родительских связей, а программный код, показанный чуть выше, позволяет получить кратчайшие пути для любой другой вершины *w*.

Кратчайшие пути для всех пар вершин. Найти кратчайший путь, соединяющий каждую пару вершин графа. Способ решения этой задачи предусматривает использование класса BFS, который решает задачу единственного источника для каждой вершины графа и поддерживает функции-элементы, которые способны эффективно выполнить обработку громадного числа запросов на вычисление кратчайших путей за счет сохранения представлений длин путей и дерева родительских связей для каждой вершины (см. рис. 18.23). Такая предварительная обработка требует времени, пропорционального VE, и пространства памяти, пропорционального V^2, что делает невозможной обработку крупных разреженных графов. Тем не менее, она позволяет строить АТД с оптимальными рабочими характеристиками: после определенных затрат ресурсов на предварительную обработку (и пространства памяти для сохранения результатов такой обработки), можно вычислить длины кратчайших путей за постоянное время, а сами пути — за время, пропорциональное их длине (см. упражнение 18.55).

Такие решения, полученные благодаря применению поиска в ширину, достаточно эффективны, однако здесь мы не будем рассматривать дополнительные подробности, поскольку они представляют собой специальные случаи алгоритмов, которые будут исследоваться в главе 21. Понятие *кратчайшего пути (shortest path)* в графе в общем случае используется для описания соответствующих задач в орграфах и сетях.

РИСУНОК 18.23. ПРИМЕР ПОИСКА КРАТЧАЙШИХ ПУТЕЙ ДЛЯ ВСЕХ ПАР ВЕРШИН

Изображенные на данном рисунке диаграммы описывают результат выполнения поиска в ширину из каждой вершины, благодаря чему производится вычисление кратчайших путей, соединяющих все пары вершин. Каждый поиск приводит к построению дерева BFS, которое определяет кратчайшие пути, соединяющие все вершины графа с вершиной в корне дерева. Результаты всех выполненных поисков сводятся в две матрицы, показанные в нижней части рисунка. В левой матрице элемент на пересечении строки **v** *и столбца* **v** *представляет длину кратчайшего пути из* **v** *в* **w** *(глубина* **v** *в дереве* **w**). *Каждая строка правой матрицы содержит массив* **st** *для соответствующего поиска. В условиях рассматриваемого примера кратчайший путь из* **3** *в* **2** *состоит из трех ребер, как показывает элемент левой матрицы, расположенный на пересечении строки 3 и столбца 2. Третье сверху слева дерево BFS говорит нам, что таким путем является* **3-4-6-2**, *и эта информация закодирована в строке 2 матрицы справа. Матрица не обязательно должна быть симметричной, когда существуют несколько кратчайших путей, поскольку обнаруженные пути зависят от порядка выполнения поиска в ширину. Например, дерево BFS, показанное внизу слева, и строка 3 правой матрицы говорят нам, что кратчайшим путем из* **2** *в* **3** *является* **2-0-5-3**.



	0	1	2	3	4	5	6	7
0	0	2	1	2	2	1	2	1
1	2	0	3	3	2	3	3	1
2	1	3	0	3	2	2	1	2
3	2	3	3	0	1	1	2	2
4	2	2	2	1	0	1	1	1
5	1	3	2	1	1	0	2	2
6	2	3	1	2	1	2	0	2
7	1	1	2	2	1	2	2	0

	0	1	2	3	4	5	6	7
0	0	7	0	5	7	0	2	0
1	7	1	0	4	7	4	4	1
2	2	7	2	4	6	0	2	0
3	5	7	0	3	3	3	4	4
4	7	7	6	4	4	4	4	4
5	5	7	0	5	5	5	4	4
6	2	7	6	4	6	4	6	4
7	7	7	0	4	7	4	4	7

Этой теме посвящена глава 21. Решения, которые там изучаются, являются строгими обобщениями описанных здесь решений с применением поиска в ширину.

Базовые характеристики динамики поиска резко контрастируют с аналогичными характеристиками поиска в глубину, что и демонстрирует большой граф, изображенный на рис. 18.24, если его сравнить с рис. 18.13. Дерево имеет небольшую глубину, зато обладает протяженной шириной. Оно может служить иллюстрацией множества особенностей, которыми поиск в ширину на графе отличается от поиска в глубину. Например:

- Существует относительно короткий путь, соединяющий каждую пару вершин графа.
- Во время поиска большая часть вершин будет смежными со множеством непосещенных вершин.

Опять-таки, этот пример типичен для поведения, которое мы ожидаем от поиска в ширину, однако проверка факта, что такими особенностями обладают модели графов, представляющие для нас интерес, и графы, с которыми приходится сталкиваться на практике, требует подробного анализа.

Поиск в глубину прокладывает свой путь в графе, запоминая в стеке точки, в которых произрастают другие пути; поиск в ширину проходит по графу, используя очередь для запоминания границ, которых он достиг. Поиск в глубину исследует граф, осуществляя поиск вершин, расположенных на значительном удалении от исходной точки, принимая к рассмотрению более близкие вершины только в случаях выхода на безусловные тупики.

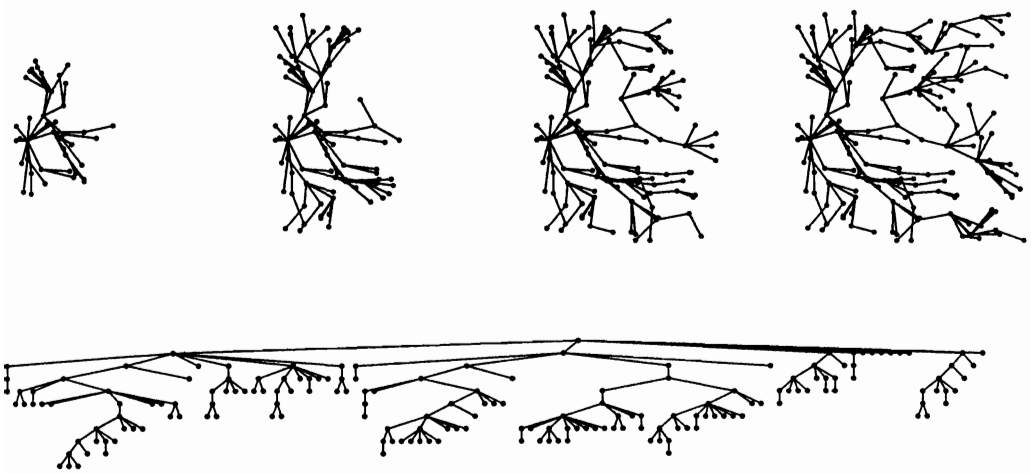


РИСУНОК 18.24. ПОИСК В ШИРИНУ

Данный рисунок служит иллюстрацией того, что поиск в ширину протекает в случайном эвклидовом графе с близкими связями (слева) в том же стиле, что и показанный на рис. 18.13. Как легко видеть из этого примера, дереву BFS свойственна малая глубина и большая ширина на рассматриваемом типе графа (а также для многих других типов графов, которые часто встречаются на практике). Иначе говоря, в рассматриваемом случае вершины соединены между собой, в основном, короткими путями. Различие между формами деревьев DFS и BFS свидетельствуют о существенном различии динамических характеристик этих алгоритмов.

Поиск в ширину полностью покрывает область, прилегающую к исходной точке, удаляясь от нее только после того, когда все ближние подступы уже исследованы. Порядок, в котором происходит посещение вершин, зависит от структуры графа и его представления, однако эти глобальные свойства деревьев поиска получают больше информации от алгоритмов, чем от самих графов или их представлений.

Главное, что требуется для правильного понимания алгоритмов обработки графов, заключается в том, чтобы четко представлять себе, что не только различные стратегии поиска служат эффективным средством изучения тех или иных свойств графа, но и то, что многие из них можно реализовать стандартным путем. Например, поиск в ширину, показанный на рис. 18.13, свидетельствует о том, что в рассматриваемом графе присутствует длинный путь, а поиск в ширину, изображенный на рис. 18.24, говорит о том, что в рассматриваемом графе имеется множество коротких путей. Несмотря на отмеченные различия в динамике, поиски в глубину и ширину имеют много общего; основное отличие между ними заключается лишь в структуре данных, которая используется для сохранения еще не исследованных ребер (и в чисто случайном совпадении, которое заключается в том, что мы можем использовать рекурсивную реализацию поиска в ширину, при этом неявную поддержку стека обеспечивает сама система). Теперь мы обратимся к изучению обобщенных алгоритмов поиска на графах, которые охватывают поиск в глубину, поиск в ширину и множество других полезных стратегий, и могут послужить основой для решения разнообразных классических задач обработки графов.

Упражнения.

18.50. Сделайте чертеж леса BFS, который может быть построен на основании стандартного BFS на списках смежных вершин графа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

18.51. Сделайте чертеж леса BFS, который может быть построен на основании стандартного BFS на матрице смежности графа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

▷ **18.52**. Внесите изменения в программы 18.8 и 19.9, которые позволили бы им использовать функцию **queue** из библиотеки STL вместо АТД из раздела 4.8.

○ **18.53**. Постройте реализацию поиска в ширину (версию программы 18.9), которая использует очереди вершин (см. программу 5.4). В программный код поиска в ширину включите специальную проверку, не допускающую проникновения в эту очередь дубликатов вершин.

18.54. Постройте матрицы всех кратчайших путей (в стиле рис. 18.22) для графа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

в предположении, что используется представление этого графа в виде матрицы смежности.

18.55. Разработайте класс поиска кратчайших путей, который реализует обслуживание запросов о кратчайших путях после предварительной обработки. В частности, определите двухмерную матрицу как приватный элемент данных, и напишите программу конструктора, который присваивает значения всем ее элементам, как показано на рис. 18.23. Далее, постройте реализации двух функций запросов, функцию **length(v, w)**

(которая возвращает длину кратчайшего пути между вершинами **v** и **w**) и функцию **path(v, w)** (которая возвращает имя вершины, смежной с **v** на кратчайшем пути между **v** и **w**).

▷ **18.56**. Какие сведения предоставляет нам дерево BFS об удаленности вершина **v** от **w** в случае, когда ни та, ни другая не являются корнем этого дерева?

18.57. Разработайте класс, объектам которого известна длина пути, достаточного, чтобы связать между собой любую пару вершин конкретного графа. (Это числовая величина называется *диаметром* (*diameter*)). *Примечание*: Необходимо сформулировать соглашение на случай, если граф окажется несвязным.

18.58. Разработайте простой и оптимальный рекурсивный алгоритм вычисления диаметра *дерева* (см. упражнение 18.57).

○ **18.59**. Снабдите класс BFS из программы 18.9 инструментальными средствами, добавив функции-элементы (и соответствующие приватные элементы данных), возвращающие высоту дерева BFS и процентное содержание ребер, которые должны просматриваться при посещении обрабатываемых вершин.

● **18.60**. Проведите соответствующие эксперименты с целью выявления средних значений числовых величин, описанных в упражнении 18.59, на графах различных размеров, основой которых служат различные модели (упражнения 17.64–17.76).

18.8. Обобщенный поиск на графах

Поиски в ширину и в глубину представляют собой фундаментальные и весьма важные методы, которые лежат в основе многочисленных алгоритмов обработки графов. Зная их основные свойства, мы сможем перейти на более высокий уровень абстракции, на котором мы обнаруживаем, что оба метода суть частные случаи обобщенной стратегии перемещения по графу, именно той, которая была заложена нами в реализацию поиска в ширину (программа 18.9).

Основной принцип достаточно прост: мы снова обращаемся к описанию поиска в ширину из раздела 18.6, но теперь вместо понятия *очередь* (*queue*) мы употребляем обобщенный термин *бахрома* (*fringe*) для описания множества ребер, которые являются кандидатами для следующего включения в дерево поиска. Мы немедленно приходим к общей стратегии поиска связной компоненты графа. Начав с петли исходной вершины в бахроме и пустого дерева, до тех пор, пока бахрома не станет пустой, выполняйте следующую операцию:

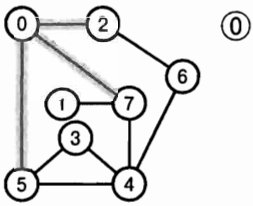
Возьмите очередное ребро из бахромы и перенесите его в дерево. Если вершина, в которую оно ведет, еще не посещалась, пройдите на эту вершину и поместите в бахрому все ребра, которые ведут из этой вершины в вершины, на которых вы еще не были.

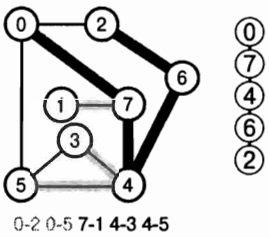
Эта стратегия описывает семейство алгоритмов поиска, которые обеспечивают посещение всех вершин и ребер связного графа *независимо* от того, какой тип обобщенной очереди используется для хранения ребер в бахроме.

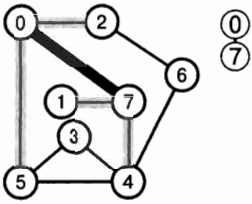
Когда для реализации бахромы используется очередь, мы получаем поиск в ширину, который рассматривается в разделе 18.6. Когда для реализации бахромы используется стек, получается поиск в глубину. На рис. 18.25, который мы предлагаем сравнить с рис. 18.6 и 18.21, этот эффект представлен во всех деталях.

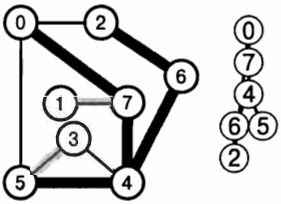
РИСУНОК 18.25 АЛГОРИТМ ПОИСКА В ГЛУБИНУ, ПОСТРОЕННЫЙ НА БАЗЕ СТЕКА

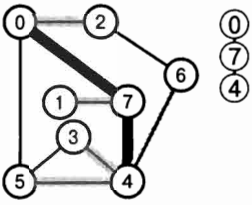
Вместе с рис. 18.21 данный рисунок служит доказательством того, что поиски в ширину и в глубину отличаются друг от друга только структурой данных, положенной в их основу. Для поиска в ширину мы использовали очередь, тогда как для поиска в глубину — стек. Выполнения этого вида поиска начинается с просмотра всех вершин, смежных с отправной вершиной (вверху слева). Затем мы перемещаем ребро ***0-7*** *из стека в дерево и заталкиваем в стек инцидентные ему ребра* ***7-1****,* ***7-4*** *и* ***7-6****, ведущие в вершины, которых еще нет в дереве (вторая диаграмма сверху слева). Дисциплина обслуживания LIFO предполагает, что когда мы помещаем то или иное ребро в стек, любые ребра, ведущие в ту же вершину, устаревают и игнорируются, когда появляются в вершине стека. Эти ребра показаны на рисунке в серых тонах. В третьих, мы переносим ребро* ***7-6*** *из стека в дерево и заталкиваем инцидентные ему ребра в стек (третья диаграмма сверху слева). Затем мы извлекаем ребро 4-6 из стека и помещаем в него инцидентные ему ребра, два из которых приводят нас к новым вершинам (диаграмма внизу слева). В завершение поиска мы извлекаем из стека оставшиеся ребра, при этом полностью игнорируем "серые" ребра, когда они всплывают в верхушку стека (справа).*

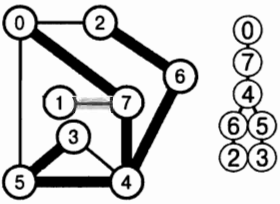


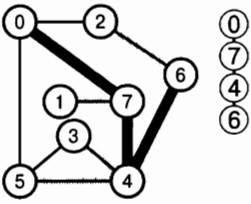


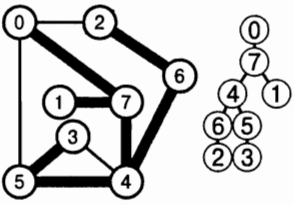












Доказательство эквивалентности рекурсивного поиска в глубину и поиска в глубину, в основу которого положен стек, представляет собой интересное упражнение, предусматривающее удаление рекурсии, в процессе которого мы фактически преобразуем стек, лежащий в основе рекурсивной программы, в стек, реализующий бахрому (см. упражнение 18.63). Порядок просмотра, осуществляемый поиском в глубину и представленный на рис. 18.25, отличается от порядка просмотра, показанного на рис. 18.6, только тем, что дисциплина обслуживания, реализуемая стеком, требует, чтобы мы проверяли ребра, инцидентные каждой вершине, в порядке, обратном тому, в котором они размещены в матрице смежности (или в списках смежных вершин). Определяющее обстоятельство заключается в том, что если мы заменим очередь, которая является структурой данных в программе 18.8, на стек (что легко сделать, поскольку интерфейсы АТД этих двух структур данных отличаются только именами функций), то мы поменяем ориентацию этой программой с поиска в ширину на поиск в глубину.

Этот обобщенный метод, как следует из раздела 18.7, может оказаться не таким эффективным, как хотелось бы, поскольку бахрома оказывается загроможденной ребрами, указывающими на вершины, которые уже были перенесены в дерево, когда данное ребро находилось в бахроме. В случае очередей FIFO нам удается избежать подобной ситуации благодаря маркировке вершин назначения в момент установки их в очередь. Мы игнорируем ребра, ведущие в вершины, находящиеся в накопителе, поскольку знаем, что они никогда не будут использованы: старое ребро уходит из очереди (и посещенная вершина) раньше, чем новое (см. программу 18.9). Что касается реализации со стеком, то в этом случае мы заинтересованы в обратном: когда в бахрому нужно поместить ребро с той же вершиной назначения, что и ребро, которое уже в ней находится, мы знаем, что *старое* ребро никогда не будет использовано, поскольку новое ребро покинет стек (равно как и посещенная вершина) раньше старого. Чтобы охватить эти два крайних случая и обеспечить возможность реализации бахромы, которая может воспользоваться какой-нибудь другой стратегией, блокирующей попадание в бахрому ребер, указывающих на ту же вершину, что и ребра в бахроме, мы скорректируем нашу обобщенную схему следующим образом:

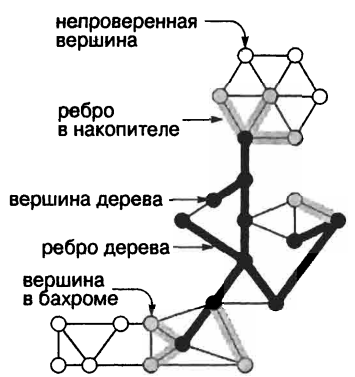


РИСУНОК 18.26. ТЕРМИНОЛОГИЯ ПОИСКА НА ГРАФЕ

Во время поиска на графе мы поддерживаем дерево поиска (черный цвет) и бахрому (серый цвет) из ребер, которые являются кандидатами на следующее включение в дерево. Любая вершина либо входит в состав дерева (черный цвет), либо в бахрому (серый цвет), либо остается непроверенной (белый цвет). Вершины дерева соединены древесными ребрами, а каждая вершина накопителя соединена ребром накопителя с некоторой вершиной дерева.

> Возьмите очередное ребро из бахромы и перенесите его в дерево. Посетите вершину, в которую оно ведет, и поместите все ребра, которые выходят из этой вершины, в бахрому, руководствуясь стратегией замены в бахроме, которая требует, чтобы никакие два ребра в бахроме не указывали на одну и ту же вершину.

Стратегия отсутствия дубликатов вершин назначения в бахроме позволяет отказаться от проверки, имело ли место посещение вершины назначения ребра, покидающего очередь. В случае поиска в ширину мы

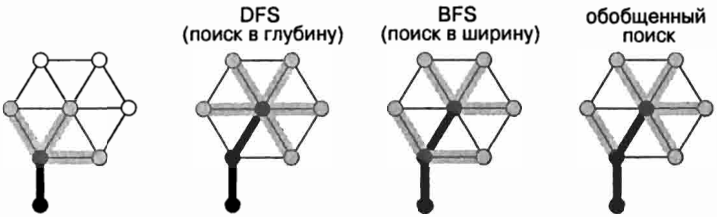


РИСУНОК 18.27. СТРАТЕГИИ ПОИСКА НА ГРАФЕ

Этот рисунок иллюстрирует различные возможности при выборе следующего действия поиска, показанного на рис. 18.26. Мы переносим вершину из бахромы в дерево (из центра колеса, изображенного вверху справа) и проверяем все ее ребра, помещая ребра, ведущие в непроверенные вершин, в бахрому и применяя при этом правило замещения, чтобы решить, следует ли ребра, указывающие на вершины в бахроме, пропустить или заменить ими ребро, которое присутствует в бахроме и указывает на ту же самую вершину. В описке в глубину мы всегда заменяем старые ребра, а в поиске в ширину — всегда игнорируем новые ребра; по условиям других стратегий мы заменяем одни ребра и пропускаем другие.

используем реализацию очереди в сочетании со стратегией игнорирования новых элементов; что касается поиска в глубину, то нам нужен стек в сочетании со стратегией игнорирования старых элементов, в то же время любая обобщенная очередь и любая стратегия замен все еще дает эффективный метод просмотра всех вершин и ребер графа за линейное время, при этом необходимо дополнительное пространство памяти, пропорциональное *V*. Рисунок 18.27 представляет собой схематическую иллюстрацию этих различий. В нашем распоряжении имеется семейство стратегий поиска на графе, в число которых входит как BFS (поиск в ширину), так и DFS (поиск в глубину), члены которого отличаются друг от друга только реализацией обобщенной очереди. Как будет вскоре показано, это семейство охватывает другие многочисленные классические алгоритмы обработки графов.

Программа 18.10 представляет реализацию, основанную на этих идеях, для графов, представленных списками смежных вершин. Она помещает ребра бахромы в обобщенную очередь и использует обычные векторы, индексированные именами вершин, для идентификации вершин в бахроме, так что она может воспользоваться явной операцией АТД *update* (обновить), всякий раз, когда сталкивается с другим ребром, ведущим в вершину в бахроме. Реализация АТД способна выбирать, игнорировать ли ей новое ребро или заменить им старое ребро.

Программа 18.10 Обобщенный поиск на графе

Данный класс просмотра обобщает алгоритмы DFS и BFS и поддерживает другие алгоритмы обработки графов (см. раздел 21.2, в котором обсуждаются эти алгоритмы и альтернативные реализации). Он поддерживает обобщенную очередь ребер, получившую название *бахрома* (*fringe*). Мы инициализируем бахрому петлей исходной вёршины; затем, пока бахрома не пуста, мы переносим ребро **e** из бахромы в дерево (присоединение к **e.v**) и проводим просмотр списка смежных вершин вершины **e.w**, во время которого помещаем непросмотренные вершины в бахрому и вызываем функцию **update** при появлении новых ребер, указывающих на вершины, зафиксированные в бахроме.

Этот программный код позволяет использовать векторы **ord** и **st** для того, чтобы никакие два ребра в бахроме не указывали на одну и те же вершину. Вершина **v** является

вершиной назначения ребра, помещенного в бахрому, тогда и только тогда, когда она помечена (значение **ord[v]** не равно -1).

```
#include "GQ.cc"
template <class Graph>
class PFS : public SEARCH<Graph>
{ vector<int> st;
  void searchC(Edge e)
  { GQ<Edge> Q(G.V());
    Q.put(e); ord[e.w] = cnt++;
    while (!Q.empty())
     {
       e = Q.get(); st[e.w] = e.v;
       typename Graph::adjIterator A(G, e.w);
       for (int t = A.beg(); !A.end(); t = A.nxt())
         if (ord[t] == -1)
           { Q.put(Edge(e.w, t)); ord[t] = cnt++; }
         else
           if (st[t] == -1) Q.update(Edge(e.w, t));
     }
  }
public:
  PFS(Graph &G) : SEARCH<Graph>(G), st(G.V(), -1)
    { search(); }
  int ST(int v) const { return st[v]; }
};
```

Свойство 18.12. *Обобщенный просмотр графа посещает все вершины и ребра графа за время, пропорциональное V^2 в случае представления графа в виде матрицы смежности, и пропорциональное $V + E$ в случае представления графа в виде списков смежных вершин плюс, в худшем случае, время, затрачиваемое на V операций вставки, V операций удаления и E операций обновления в очереди размера V.*

Доказательство: Доказательство свойства 18.12 не зависит от реализации очереди и благодаря этому обстоятельству следует из полученных ранее результатов. Указанные в формулировке свойства дополнительные затраты времени на операции, выполняемые обобщенной очередью, следуют непосредственно из программной реализации. ■

Существует множество других, заслуживающих внимания, эффективных моделей АТД бахромы. Например, как и в случае первой предложенной нами реализации поиска в ширину, мы можем воспользоваться нашей первой обобщенной схемой и просто поместить все ребра в бахрому и игнорировать те их них, которые ведут в вершины, уже включенные в дерево в тот момент, когда мы извлекаем их из бахромы. Недостаток такого подхода, как и в случае поиска в ширину, заключается в том, что максимальный размер очереди должен быть равным E, а не V. Либо мы должны выполнять обновления неявно в рамках реализации АТД, следя за тем, чтобы никакие два ребра с одной и той же вершиной назначения не могли одновременно находиться в очереди. Однако простейший способ реализации АТД, решающего эту задачу, по существу, эквивалентен использованию вектора, индексированного именами вершин (см. упражнения 4.51 и 4.54), ибо проверка этого вектора более удобна для клиентских программ, применяющих поиск на графе.

Сочетание программы 18.10 с абстракцией обобщенной очереди дает нам в руки универсальный и гибкий механизм поиска на графе. Чтобы проиллюстрировать это утверждение, кратко рассмотрим две интересных и полезных альтернативы поискам в глубину и ширину.

Первая альтернативная стратегия основана на использовании *рандомизированной очереди* (*randomized queue*) (см. раздел 4.1). Из рандомизированных очередей элементы *удаляются* случайным образом: любой элемент такой структуры данных может с равной вероятностью быть удален. Программа 18.11 представляет собой реализацию, которая обеспечивает упомянутую функциональность. Если мы используем этот программный код для реализации АТД обобщенной очереди, то получаем алгоритм случайного поиска на графе, по условиям которого любая вершина, находящаяся в накопителе, с равной вероятностью может стать следующей вершиной, включенной в дерево. Выбор ребра (ведущего в эту вершину), добавляемого в дерево, зависит от реализации операции *обновления* (*update*). Реализация, представленная программой 18.11, не производит обновления данных, таким образом, каждая вершина из бахромы включается в дерево вместе с ребром, которое послужило причиной ее переноса в бахрому. С другой стороны, мы могли бы избрать стратегию "обновлять всегда" (которая предполагает добавление в дерево последнего ребра, если оно направлено в какую-либо из вершин, помещенных в бахрому), либо стратегию случайного выбора.

Другая стратегия, играющая исключительно важную роль в изучении алгоритмов обработки графов, поскольку служит основой целого ряда классических алгоритмов, которые будут изучаться в главах 20–22, – это использование для бахромы АТД *очереди с приоритетами* (*priority queue*) (см. главу 9). Мы присваиваем определенное значение приоритета каждому ребру, обновляем его соответствующим образом и выбираем ребро с наивысшим приоритетом для очередного включения в дерево. Подробный анализ этой формулировки проводится в главе 20. Операции по поддержке очереди по приоритетам требуют больших затрат, чем аналогичные операции в отношении стеков и очередей, поскольку для этого необходимо выполнять операции сравнения элементов очереди, но в то же время они могут поддерживать значительно более широкий класс алгоритмов поиска на графах. Как мы увидим далее, некоторые из наиболее важных задач обработки графов можно решать путем выбора подходящей процедуры назначения приоритетов в рамках обобщенного поиска на графе, построенного на базе очереди по приоритетам.

Программа 18.11. Реализация рандомизированной очереди

Когда мы удаляем элемент из этой структуры данных, в равной степени вероятно, что это один из элементов, сохраняемых на текущий момент в этой структуре данных. Можно использовать этот программный код, реализующий АТД обобщенной очереди для поиска на графе, для поиска на графе в "стохастическом" режиме (*см. рассуждения по тексту раздела*).

```
template <class Item>
class GQ
  {
    private:
      vector<Item> s; int N;
    public:
      GQ(int maxN) : s(maxN+1), N(0) { }
      int empty() const
        { return N == 0; }
```

```
    void put(Item item)
      { s[N++] = item; }
    void update(Item x) { }
    Item get()
      { int i = int(N*rand()/(1.0+RAND_MAX));
        Item t = s[i];
        s[i] = s[N-1];
        s[N-1] = t;
        return s[--N]; }
};
```

Все обобщенные алгоритмы поиска на графе исследуют каждое ребро всего лишь один раз и в худшем случае требуют дополнительного пространства памяти, пропорционального V; в то же время они различаются некоторыми показателями производительности. Например, на рис. 18.28 показано, как меняется наполнение бахромы в процессе выполнения поиска в глубину, в ширину и рандомизированного поиска; на рис. 18.29 показано дерево, вычисленное с применением рандомизированного поиска для примера, представленного на рис. 18.13 и рис. 18.24. Для рандомизированного поиска не характерны ни длинные пути, свойственные поиску в глубину (DFS), ни узлы с высокими степенями, свойственные поиску в ширину (BFS). Форма этих деревьев и диаграмм бахромы в значительной мере зависит от конкретной структуры графа, на котором производится поиск, тем не менее, они в определенной степени могут служить характеристиками различных алгоритмов.

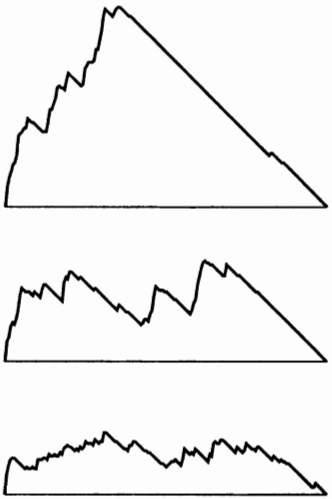


РИСУНОК 18.28. РАЗМЕРЫ БАХРОМЫ В УСЛОВИЯХ ПРИМЕНЕНИЯ ПОИСКА ГЛУБИНУ, РАНДОМИЗИРОВАННОГО ПОИСКА И ПОИСКА В ШИРИНУ

Изображенные на рисунке диаграммы отображают наполнение бахромы во время поисков, представленных на рис. 18.13, 18.24 и 18.29, показывают, какое огромное влияние оказывает выбор структуры данных бахромы на поиск на графе. Когда используется стек в условиях поиска в глубину (верхняя диаграмма), происходит наполнение бахромы с самого начала поиска, поскольку мы находим новые узлы на каждом шаге, затем, когда поиск завершается, удаляем из бахромы все, что там было. Когда используется рандомизированная очередь (в центре), максимальный размер очереди намного меньше. В случае применения очереди FIFO в условиях поиска в ширину (внизу) максимальный размер очереди еще ниже, а в процессе поиска обнаруживаются новые узлы

Степень обобщения поиска можно повысить еще больше, если во время поиска работать с лесом (не обязательно с деревом). Мы уже вплотную пошли к изучению этого уровня абстракции, однако сейчас мы не будем затрагивать эту тему, а рассмотрим несколько алгоритмов такого типа в главе 20.

Упражнения

- **18.61**. Проанализируйте преимущества и недостатки обобщенной реализации поиска на графе, в основу которой положена следующая стратегия. "Перенести ребро из бахромы в дерево. Если вершина, к которой оно ведет, не посещалась, выйти на эту вершину и поместить в бахрому все инцидентные ей ребра".

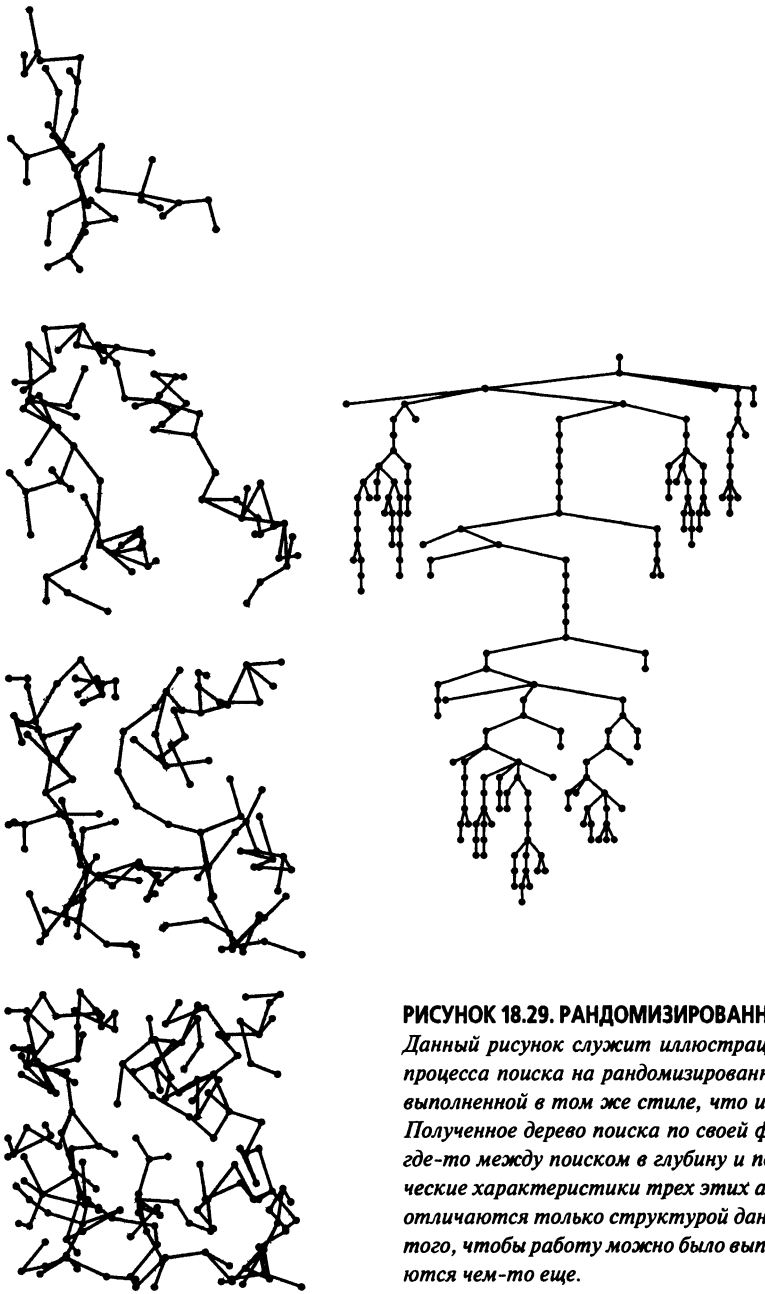


РИСУНОК 18.29. РАНДОМИЗИРОВАННЫЙ ГРАФ ПОИСКА

Данный рисунок служит иллюстрацией динамики развития процесса поиска на рандомизированном графе (слева), выполненной в том же стиле, что и рис. 18.13 и 18.24. Полученное дерево поиска по своей форме занимает место где-то между поиском в глубину и поиском в ширину. Динамические характеристики трех этих алгоритмов, которые отличаются только структурой данных, необходимой для того, чтобы работу можно было выполнить, вряд ли отличаются чем-то еще.

18.62. Разработайте реализацию АТД графа, представленного списками смежных вершин, которая содержит ребра (а не только их вершины назначения) в списках, а затем реализуйте на графе поиск, основанный на стратегии, описанной в упражнении 18.61, который посещает каждое ребро и в то же время разрушает граф, воспользовавшись тем обстоятельством, что вы можете перемещать ребра всех вершин в бахрому за счет изменения одной связи.

● **18.63**. Докажите, что рекурсивный поиск в глубину (программа 18.3) эквивалентен обобщенному поиску на графе с использованием стека (программа 18.10) в том смысле, что обе программы посещают все вершины всех графов в одном и том же порядке тогда и только тогда, когда эти программы просматривают списки смежных вершин в обратных порядках.

18.64. Предложите три различных порядка обхода при рандомизированном поиске на графе

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

18.65. Может ли рандомизированный поиск наносить визиты вершинам графа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

в порядке следования их индексов? Докажите правильность вашего ответа.

18.66. Воспользуйтесь библиотекой STL для построения обобщенной очереди ребер графа, которая не принимает ребер с дублированными вершинами, используя для этой цели стратегию игнорирования новых элементов.

○ **18.67**. Разработайте алгоритм рандомизированного поиска на графе, который выбирает из накопителя то или иное ребро с равной вероятностью. *Указание*: См. программу 18.8.

○ **18.68**. Дайте описание стратегии обхода лабиринта, которая соответствует использованию стека магазинного типа в обобщенном поиске на графе (см. раздел 18.1).

○ **18.69**. Снабдите обобщенный поиск на графе (см. программу 18.10) инструментальными средствами, позволяющими вывести на печать значения высоты дерева и процентное отношение просмотренных ребер каждой просматриваемой вершины.

● **18.70**. Проведите эксперименты с целью определить эмпирическим путем среднее значения количественных величин, описанных в упражнении 18.69, применительно к обобщенному поиску на графе с использованием случайной очереди в графах различных размеров (см. упражнения 17.64–17.76).

● **18.71**. Реализуйте производный класс, строящий динамические графические анимации обобщенного поиска на графе, в которой с каждой вершиной ассоциируется координаты (x, y) (см. упражнения 17.55–17.59). Протестируйте полученную программу на случайных эвклидовых графах с близкими связями, используя столько точек, сколько сможете обработать за приемлемый промежуток времени. Ваша программа должна строить изображения, подобные диаграммам, представленным на рис. 18.13, 18.24 и 18.29; свобода выбора различных цветов вместо степеней серого для обозначения дерева, бахромы и непросмотренных вершин и ребер остается за вами.

18.9. Анализ алгоритмов на графах

Нам предстоит рассмотреть широкий набор задач обработки графов и методов их решения, в связи с чем мы не всегда сравниваем различные многочисленные алгоритмы решения одной и той же задачи. Тем не менее, всегда полезно набраться опыта работы с алгоритмами, проверяя их на реальных данных или на специально подготовленных и понятных нам данных, обладающих свойствами, которые мы можем обнаружить в фактических применениях.

Как уже отмечалось в главе 2, мы стремимся – в идеальном случае – получить естественные модели ввода данных, обладающие тремя важными свойствами:

- Они с достаточной степенью точности отражают реальное положение вещей, что делает возможным их применение для прогнозирования производительности.
- Они достаточно просты, что делает возможным применение для их исследования математического анализа.
- Можно построить специальные генераторы экземпляров задач, которые можно использовать для тестирования наших алгоритмов.

Имея в своем распоряжении эти три компонента, мы можем приступать к реализации сценария со стадиями разработки, анализа, реализации и тестирования, который позволяет создавать эффективные алгоритмы решения практических задач.

В таких областях, как сортировка и поиск, по указанным выше направлениям в частях 3 и 4 мы достигли определенных успехов. Мы можем анализировать алгоритмы, генерировать случайные экземпляры задач и совершенствовать программные реализации с тем, чтобы получать высокоэффективные программы для использования в самых разнообразных практических ситуациях. В ряде других областей, которые мы также изучаем, могут возникнуть различные трудности. Например, математический анализ, который мы хотели бы применить на достаточно высоком уровне, остается для нас недоступным при решении многих геометрических задач, а разработка точной модели ввода все еще остается трудной проблемой для многих алгоритмов обработки строк (в самом деле, на эту часть приходится существенная часть вычислений). Аналогично, алгоритмы на графах приводят нас к ситуации, когда во многих приложениях мы вынуждены ходить по тонкому льду в попытках найти нужные решения с учетом всех трех свойств, которые только что были отмечены:

- Возможности математического анализа ограничены, вследствие чего многие важные вопросы на стадии анализа остаются без ответа.
- Существует большое разнообразие различных типов графов, и мы не в состоянии адекватно проверить предложенные алгоритмы на всех типах графов.
- Выявление отличительных признаков типов графов, которые появляются в практических задачах, в большинстве случаев представляют собой нечетко сформулированную задачу.

Графы являются достаточно сложными структурами, и часто нам не удается в полной мере выявить основные свойства графов, с которыми нам приходится сталкиваться на практике, *или* искусственных графов, которые мы, возможно, можем строить и анализировать.

Ситуация, по-видимому, не такая безнадежная, как она описана выше, по одной простой причине: многие из рассматриваемых алгоритмов на графах *оптимальны* в худшем случае, так что прогнозирование производительности алгоритмов не представляет трудностей. Например, программа 18.7 обнаруживает мосты по результатам лишь одного исследования каждого ребра и каждой вершины. Это требует таких же затрат, как и построение структуры данных графа, и мы можем с достаточной точностью предсказать, например, что удвоение числа ребер приводит к увеличению времени выполнения в два раза независимо от того, обработку каких типов графов мы выполняем.

Когда время выполнения алгоритма зависит от структуры входного графа, такого рода прогнозы делать намного труднее. Тем не менее, если возникает необходимость обработки очень большого числа крупных графов, нам нужны эффективные алгоритмы по той же причине, по какой они требуются и в любой другой проблемной области, поэтому мы продолжим изучение основных свойств алгоритмов и приложений и будем стараться выявить те методы, которые обеспечивают наилучшую обработку графов, которые могут возникать на практике.

Чтобы проиллюстрировать некоторые из этих проблем, мы вернемся к изучению свойства связности графа, к задаче, которую анализировалась еще в главе 1 (!). Связность случайных графов привлекала к себе внимание математиков в течение многих лет, этой теме посвящено множество пространных публикаций. Эта литература выходит за рамки настоящей книги, в то же время она представляет собой фон, который оправдывает использование этой задачи как основы для некоторых экспериментальных исследований, углубляющих наше понимание базовых алгоритмов, которые мы используем, и типов графов, которые мы изучаем.

Например, наращивание графов за счет добавления случайных ребер во множество первоначально изолированных вершин (по существу, этот процесс реализуется программой 19.12) представляет собой подробно исследованный процесс, который послужил основой классической теории случайных графов. Хорошо известно, что по мере возрастания числа ребер, такой граф сливается, образуя одну гигантскую компоненту. Литература по случайным графам дает обширную информацию о природе этого процесса. Например,

Свойство 18.13. *Если* $E > \frac{1}{2} V \ln V + \mu V$ *(при положительном значении* μ*), то случайный граф с* V *вершинами и* E *ребрами состоит из одной связной компоненты и изолированных вершин, среднее число которых не превышает* $e^{-2\mu}$*, с вероятностью, приближающейся к* 1 *при бесконечном возрастании* V*.*

Доказательство: Этот факт был установлен в пионерской работе Эрдеша (Erdos) и Реньи (Renyi) в 1960 г. Само доказательство выходит за рамки данной книги (см. раздел ссылок). ■

Таблица 18.1. Связность на примере двух моделей случайного графа

В этой таблице показано число связных компонент и размер максимальной связной компоненты графов, содержащих 100000 вершин и полученных на базе двух различных распределений. Что касается модели случайного графа, то такие эксперименты подтверждают хорошо известный факт, что граф с высокой вероятностью состоит в основном из одной большой компоненты, если среднее значение степени вершины превышает некоторое небольшое постоянное значение. В двух правых столбцах приводятся экспериментальные данные для случая, когда мы накладываем ограничение

на выбор ребер, которые выбираются из числа соединяющих каждую вершину с только одной из 10 указанных соседних вершин.

	Случайно выбранные ребра		10 соседних случайно выбранных ребер	
E	C	L	C	L
1000	99000	5	99003	3
2000	98000	4	98010	4
5000	95000	6	95075	5
10000	90000	8	90300	7
20000	80002	16	81381	9
50000	50003	1701	57986	27
100000	16236	79633	28721	151
200000	1887	98049	3818	6797
500000	4	99997	19	99979
1000000	1	100000	1	100000

Обозначения:

C – число связных компонентов

L – размер наибольшей связной компонентой

Это свойство говорит о том, что мы вполне можем ожидать, что крупные неразреженные случайные графы являются связными. Например, если $V > 1000$ и $E > 10V$, то $\mu > 10 - \frac{1}{2}\ln 1000 > 6.5$ и среднее число вершин, не содержащихся в гигантской компоненте, (почти наверняка) меньше, чем $e^{-13} < 0.000003$. Если вы генерируете миллион случайных графов, в которые входят 1000 вершин с плотностью, превосходящей 10, можно получить несколько графов с одной изолированной вершиной, зато все остальные графы будут связными.

На рис. 18.30 случайные графы сравниваются со случайными близкими графами, в которые мы включаем только ребра, соединяющие вершины с другими вершинами, индексы которых различаются на некоторое небольшое постоянное значение. Модель близкого графа порождает графы, которые по своим характеристикам существенно отличаются от случайных графов. В конечном итоге мы получим крупную компоненту, но она появится внезапно, когда произойдет слияние двух крупных компонент.

Из таблицы 18.2 следует, что эти структурные различия между случайными графами и близкими графами сохраняются и тогда, когда V и E находятся в пределах, представляющих практический интерес. Такие структурные различия, несомненно, могут отразиться на производительности разрабатываемых нами алгоритмов.

Таблица 18.2. Эмпирическое исследование алгоритмов поиска на графе

В этой таблице показаны значения относительного времени решения задачи вычисления количества связных компонент (и размера наибольшего из этих компонент) для графов с различным числом вершин и ребер. В соответствии с предположением, алгоритмы, которые используют представление графа в виде матрицы смежности, выполняются довольно медленно на разреженных графах, в то же время они вполне конкурентоспособны применительно к насыщенным графам. Что касается данной

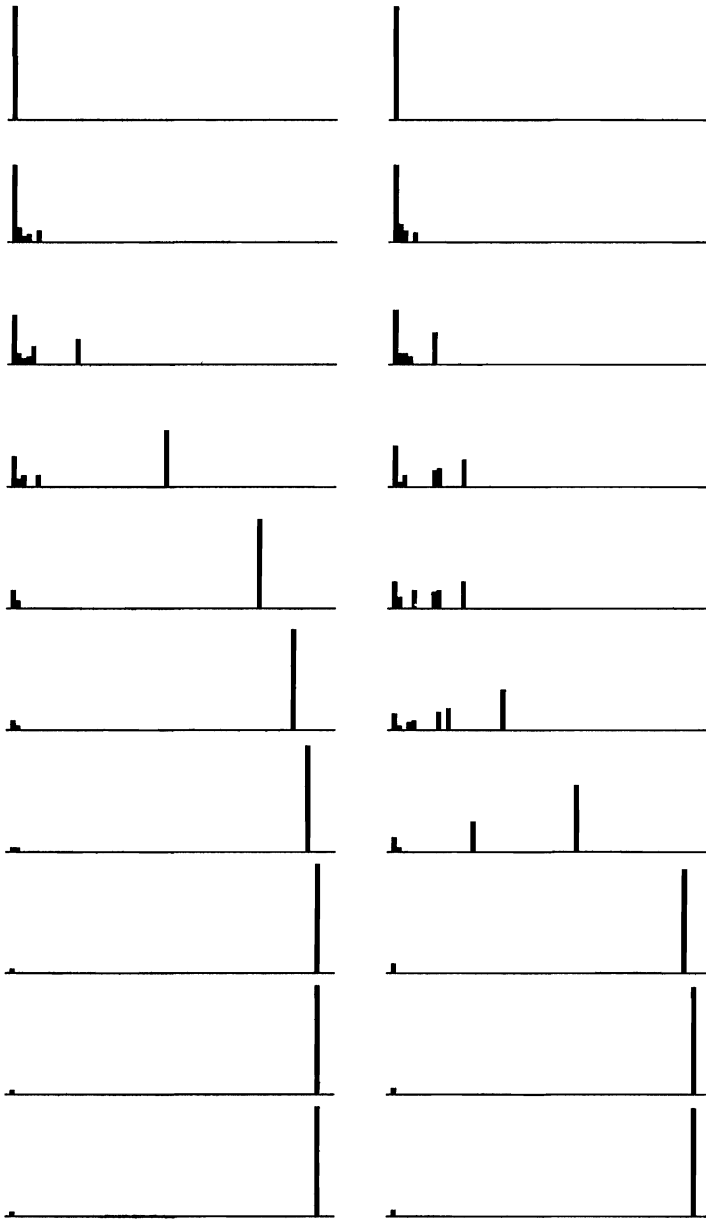


РИСУНОК 18.30. СВЯЗНОСТЬ В СЛУЧАЙНЫХ ГРАФАХ

На этом рисунке показана эволюция двух типов случайных графов на 10 равных шагах в условиях, когда в общем случае в первоначально пустые графы добавляются 2E ребер. Каждая диаграмма представляет собой гистограмму числа вершин в компонентах размером от 1 до V (слева направо). Мы отталкиваемся от ситуации, когда все вершины содержатся в компонентах размера 1 и заканчиваем, когда практически все вершины входят в одну компоненту гигантских размеров. Диаграмма слева отображает стандартный случайный граф: гигантская компонента формируется быстро, а все другие компоненты малы. Диаграмма справа отображает случайный граф с близкими связями: компоненты различных размеров сохраняются в течение более длительного времени.

конкретной задачи, то алгоритмы объединения-поиска, которые мы рассматривали в главе 1, самые быстрые, поскольку они выполняют построение структуры данных, предназначенной специально для решения данной задачи, а поэтому не нуждаются в другом представлении графа. Но если структура данных, представляющая граф, уже построена, то алгоритмы DFS и BFS демонстрируют более высокое быстродействие и большую гибкость. Добавление проверки с целью прекращения выполнения алгоритма, когда известно, что граф представляет собой единую связную компоненту, существенно повышает быстродействие поиска в глубину и алгоритма объединения-поиска (но не поиска в ширину) при обработке насыщенных графов.

			Матрица смежности			Список смежных вершин				
E	**U**	**U***	**I**	**D**	**D***	**I**	**D**	**D***	**B**	**B***
5000 вершин										
500	1	0	255	312	356	1	0	0	0	1
1000	0	1	255	311	354	1	0	0	0	1
5000	1	2	258	312	353	2	2	1	2	1
10000	3	3	258	314	358	5	2	1	2	1
50000	12	6	270	315	202	25	6	4	5	6
100000	23	7	286	314	181	52	9	2	10	11
500000	117	5	478	248	111	267	54	16	56	47
100000 вершин										
5000	5	3				3	8	7	24	24
10000	4	5				6	7	7	24	24
50000	18	18				26	12	12	28	28
100000	34	35				51	28	24	34	34
500000	133	137				259			88	89

Обозначения

U – взвешенное быстрое объединение посредством сжатия пути делением пополам (программа 1.4)

I – начальная конструкция представления графа

D – рекурсивный поиск в глубину

B – поиск в ширину (программа 18.9)

***** – выход, когда установлен факт полной связности графа

В таблицу 18.2 сведены эмпирические результаты затрат на определение числа связных компонент случайного графа с применением различных алгоритмов. Возможно, прямое сравнение этих алгоритмов не вполне правомерно, поскольку они разрабатывались для решения различных задач, тем не менее, эти эксперименты подтверждают некоторые выводы, к которым мы пришли.

Во-первых, из таблицы ясно, что не следует использовать представление графа в виде матрицы смежности в случае разреженных графов больших размеров (и нельзя его использовать применительно к очень большим графам), причем не только в силу того, что затраты на инициализацию матрицу неприемлемы, но и по причине того, что алгоритм

просматривает каждый элемент матрицы, вследствие чего время его выполнения оказывается пропорциональным размеру (V^2) матрицы, а не числу единиц в ней (E). Из таблицы следует, например, что на обработку графа, содержащего 1000 ребер, затрачивается примерно такое же время, которое требуется на обработку графа, содержащего 100000 ребер в случае использования матрицы смежности.

Во-вторых, из таблицы 18.2 легко видеть, что затраты на распределение памяти под узлы списков достигают больших значений, когда списки смежных вершин строятся для крупных разреженных графов. Затраты на построение таких списков превосходят стоимость их обхода более чем в пять раз. В типичной ситуации, когда есть намерения выполнить многочисленные поиски различных типов после того, как граф будет построен, цена становится приемлемой. В противном случае мы должны рассмотреть альтернативные реализации, позволяющие снизить эти затраты.

В третьих, отсутствие числовых значений в столбцах DFS для крупных разреженных графов весьма существенно. Такие графы порождают избыточную глубину рекурсии, которая (в конечном итоге) приводит к аварийному завершению программы. Если для таких графов мы хотим использовать поиск в глубину, необходимо применять нерекурсивные версии программ, о чем шла речь в разделе 18.7.

В-четвертых, эта таблица показывает, что метод, в основу которого положен алгоритм объединения-поиска, описанный в главе 1, обладает большим быстродействием, чем поиски в глубину и в ширину, главным образом за счет того, что для него не требуется представление графа целиком. Однако, не имея такого представления, мы не можем дать ответ на такие простые запросы, как: "существует ли ребро, соединяющее вершины **v** и **w**?". Таким образом, методы, основанные на алгоритме объединения-поиска, не подходят, если мы хотим получить от них нечто большее, на что они способны (в частности, ответы на запросы типа "существует ли ребро, соединяющее вершины **v** и **w**?" вперемешку с добавлением ребер). Если внутреннее представление графа построено, нецелесообразно строить алгоритм объединения-поиска только для того, чтобы узнать, связный граф или нет, поскольку и поиск в глубину, и поиск в ширину могут дать ответ так же быстро.

Когда мы проводим эмпирические испытания, которые приводят к появлению подобного рода таблиц, различные аномальные явления могут потребовать дальнейших пояснений. Например, на многих компьютерах архитектура кэш-памяти и другие свойства запоминающего устройства могут решающим образом повлиять на производительность алгоритма на крупных графах. Необходимость повышения производительности критических приложений может потребовать, помимо всех факторов, которые мы сейчас рассматриваем, и самого подробного изучения архитектуры машины.

Подробное исследование этих таблиц позволяет обнаружить больше свойств этих алгоритмов, чем мы способны изучить. Однако наша цель состоит не в том, чтобы провести тщательное сравнение, но и показать, что несмотря на множество проблем, с которыми приходится сталкиваться при сравнении различных алгоритмов на графах, мы можем и должны проводить эмпирические исследования и извлекать пользу из любых аналитических результатов как с целью получить представление об основных особенностях алгоритмов, так и с целью прогнозирования их производительности.

Упражнения

○ **18.72**. Выполните эмпирические исследования, основной целью которых должно быть получение таблицы, подобной таблице 18.2, для задачи, определяющей, является ли заданный граф двухдольным (допускает ли раскрашивание в два цвета).

18.73. Выполните эмпирические исследования, основная цель которых состоит в том, чтобы получить таблицу, подобную таблице 18.2, для задачи, определяющей, является ли заданный граф двусвязным.

○ **18.74**. Выполните эмпирические исследования с целью определить размер второй по величине связной компоненты разреженных графов различных размеров, построенных на базе различных моделей (упражнения 17.64–17.76).

18.75. Напишите программу, которая способна строить диаграммы, подобные показанной на рис. 18.30, и протестируйте ее на графах различных размеров, в основе которых лежат различные модели (упражнения 17.64–17.76).

○ **18.76**. Внесите изменения в программу из упражнения 18.75 с целью построения аналогичных гистограмм размеров реберно-связных компонент.

●● **18.77**.Числа в таблицах, приведенных в данном разделе, получены при исследовании только одного образца. Возможно, мы захотим подготовить аналогичную таблицу, для получения одного элемента которой мы готовы провести 1000 экспериментов и получить выборочное среднее и среднее отклонение, но, по-видимому, мы не сможем включить примерно такое число элементов. Гарантирует ли данный подход более эффективное использование времени компьютера? Дайте обоснование своего ответа.

Глава 19

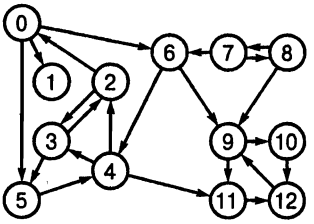
Орграфы и ориентированные ациклические графы

Если мы придаем значение, в каком порядке описываются две вершины каждого ребра графа, мы получаем совершенно другой комбинаторный объект, известный как *ориентированный граф* (*directed graph*), или *орграф* (*digraph*). На рис. 19.1 показан пример орграфа. В орграфах обозначение *s-t* описывает ребро, которое ведет из вершины *s* в вершину *t*, но при этом оно не дает никакой информации о том, существует ли ребро, ведущее из *t* в *s*. Имеется четыре вида отношений, в которых могут находиться две любые вершины орграфа: связывающее их ребро отсутствует, ребро *s-t*, ведущее из *s* в *t*, ребро *t-s*, ведущее из *t* в *s*, и два ребра *s-t* и *t-s*, которые указывают на наличие связи в обоих направлениях. Требование однонаправленности естественно для многих приложений, оно легко реализуется в разрабатываемых нами программах и представляется нам безобидным, однако оно влечет за собой появление новой комбинаторной структуры, которая оказывает глубокое воздействие на наши алгоритмы; в силу присущих ей свойств работа с орграфами существенно отличается от работы с неориентированными графами. Обработку орграфов можно сравнить с ездой по городу, в котором *все* улицы имеют односторонне движение, и при этом направление движения не указано с помощью какого-нибудь унифицированного способа. Можно себе представить, какие приключения нас ожидают, если в подобной ситуации мы вознамеримся проехать из одной точки города в другую.

Направления ребер в графе мы интерпретируем различными способами. Например, при построении графа телефонных вызовов можно принимать за направление ребра направление от вызывающего абонента к абоненту, принима-

ющему вызов. В графе финансовых операций можно применять тот же принцип, при этом ребро представляет собой финансовые средства, товары или информацию, передаваемые от одного агента другому. Мы можем найти подходящий пример, соответствующий этой классической модели в Internet; в этом случае вершины представляют Web-страницы, а ребра — ссылки между страницами. В разделе 19.4 мы рассмотрим другие примеры; большая их часть моделирует ситуации с большей степенью абстракции.

Довольно часто приходится сталкиваться с ситуациями, когда направление ребра отражает отношение предшествования. Например, орграф может моделировать производственную линию: вершины обозначают работу, которую необходимо выполнить, при этом существует ребро, ведущее из вершины *s* в вершину *t*, если работа, соответствующая вершине *s*, должна быть выполнена, прежде чем может быть выполнена работа, соответствующая вершине *t*. Другой способ моделирования той же ситуации заключается в использовании *диаграммы PERT* (PERT, Project Evaluation and Review Technique — сетевое планирование и управление): ребра представляют работы, а вершины неявно описывают отношение предшествования (все работы, сходящиеся в конкретной вершине, должны быть завершены, прежде чем начнутся работы, исходящие из этой вершины). Какие следует принимать решения относительно сроков выполнения каждой из этих работ, чтобы ни одно из отношений предшествования не было нарушено? В этом заключается суть проблемы, известной как задача *составления расписаний (scheduling problem)*. Она не имеет смысла, если в орграфе имеется цикл, в подобного рода ситуациях мы работаем с *графами DAG* (directed acyclic graph — ориентированный ациклический граф). В разделах 19.5–19.7 мы рассмотрим базовые свойства графа DAG и алгоритмы решения простой задачи, получившей название *топологической сортировки* (*topological sorting*). На практике при решении задач составления расписаний в общем случае учитывается вес вершин или ребер, которые моделируют время или стоимость каждой работы. Такие задачи мы будем изучать в главах 21 и 22.



4-2	11-12	4-11	5-4
2-3	12-9	4-3	0-5
3-2	9-10	3-5	6-4
0-6	9-11	7-8	6-9
0-1	8-9	8-7	7-6
2-0	10-12		

РИСУНОК 19.1. ОРИЕНТИРОВАННЫЙ ГРАФ (ОРГРАФ)

Орграф определяется списком вершин и ребер (внизу), при этом порядок, в котором мы перечисляем вершины при описании ребра, показывает, что ребро направлено из первой вершины во вторую. На чертежах орграфа для изображения ориентированных ребер (сверху) используются стрелки.

Число возможных орграфов поистине огромно. Каждое из возможных V^2 ориентированных ребер (в том числе петли) может входить в график или не входить в него, таким образом, общее число разных орграфов равно 2^{V^2}. Как показано на рис. 19.2, с увеличением числа вершин это число стремительно возрастает, даже по сравнению с числом различных неориентированных графов, уже при небольших значениях *V*. Как и в случае неориентированных графов, число изоморфных друг другу классов (вершины одного из изоморфных графов можно переименовать таким образом, чтобы получить график, идентичный другому) намного меньше, однако мы не можем извлечь для себя конкретную пользу из подобного сокращения числа графов, поскольку нам неизвестен достаточно эффективный алгоритм выявления изоморфных графов.

Разумеется, любая программа способна обработать лишь крохотную часть возможных орграфов; в самом деле, эти числа настолько велики, что у нас есть все основания утверждать, что фактически ни один орграф не окажется в числе тех, которые будут обработаны какой-либо заданной программой. В общем случае трудно описать свойства орграфов, с которыми приходится сталкиваться на практике, и это обстоятельство заставляет нас так разрабатывать алгоритмы, чтобы они могли принимать на входе любой возможный орграф. С другой стороны, с такого рода ситуациями нам уже приходилось сталкиваться (например, ни одна из 1000! перестановок 1000 элементов никогда не была обработана ни одной из программ сортировки). С другой стороны, вас, возможно, неприятно поразит тот факт, что, например, даже если все электроны вселенной способны привести в действие суперкомпьютеры, способные выполнять обработку 10^{10} графов за секунду в течение всей жизни вселенной, то эти суперкомпьютеры не просмотрят и 10^{-100} процента орграфов, состоящих из 10 вершин (см. упражнение 19.9).

V	неориентированные графы	орграфы
2	8	16
3	64	512
4	1024	65536
5	32768	33554432
6	2097152	68719476736
7	268435456	562949953421312

РИСУНОК 19.2. ПОДСЧЕТ ЧИСЛА ГРАФОВ

В то время как число различных неориентированных графов с V вершинами велико, даже когда V мало, число различных орграфов с V вершинами намного больше. Что касается неориентированных графов, то их число определяется формулой $2^{V(V+1)/2}$, в то время как число орграфов определяется формулой 2^{V^2}.

Это небольшое отступление, касающееся подсчета графов, возможно, выделяет несколько моментов, которым мы уделяем особое внимание, когда проводим анализ алгоритмов, и показывает их непосредственное отношение к изучению орграфов. Должны ли мы разрабатывать алгоритмы так, чтобы они хорошо работали в худшем случае, когда вероятность обнаружить орграф для худшего случая настолько мала? Есть ли смысл выбирать алгоритмы на основании анализа обычных случаев или все это лежит в области математической фантазии? Если нашей целью является получение программной реализации, которая эффективно работает на орграфах, с которыми мы имеем дело на практике, мы непосредственно сталкиваемся с проблемой описания таких орграфов. Разработать математические модели, достаточно точно описывающие орграфы, с которыми нам, возможно, придется иметь дело в приложениях, еще труднее, чем модели неориентированных графов.

В этой главе мы вновь обратимся, но уже в контексте орграфов, к поднабору фундаментальных задач обработки графов, которые уже рассматривались в главе 17. При этом мы исследуем несколько задач, специфичных только для орграфов. В частности, мы рассмотрим алгоритм поиска в глубину и несколько его приложений, включая задачу *обнаружения циклов* (*cycle detection*) (с целью определить, не является ли рассматриваемый орграф графом DAG); задачу *топологической сортировки* (*topological sort*) (для решения, например, задачи составления расписаний для только что описанных графов DAG), а также вычисление *транзитивных замыканий* (*transitive closure*) и *сильных компонент* (*strong components*) (которые решают фундаментальную задачу поиска ориентированной пути между двумя заданными вершинами). Как и в любых других областях обработки графов, эти алгоритмы занимают диапазон от тривиальных до исключительно хитроумных; они характеризуются сложными комбинаторными структурами орграфа и одновременно предоставляют нам возможность изучать эти структуры.

Упражнения

- **19.1.** Назовите пример крупного орграфа, возможно, графа, описывающего управление какими-либо объектами в оперативном режиме, — что-то наподобие графа бизнес-операций одной из систем, работающей в оперативном режиме, или орграфа, определяемого связями Web-страниц.

- **19.2.** Назовите пример крупного графа DAG, описывающего какую-нибудь деятельность, выполняемую в интерактивном режиме, возможно, графа, определяемого зависимостями, связывающими определения функций в крупной системе программного обеспечения, или связями каталогов в крупной файловой системе.

19.3. Постройте таблицу, аналогичную представленной на рис. 19.2, не учитывая при этом все орграфы и петли.

19.4. Каково число орграфов, содержащих V вершин и E ребер?

○ **19.5.** Сколько орграфов соответствует каждому неориентированному графу, содержащему V вершин и E ребер?

▷ **19.6.** Сколько нужно числовых разрядов, чтобы выразить число орграфов, содержащих V вершин и E ребер, в виде 10-значного числа?

○ **19.7.** Начертите неизоморфные орграфы, содержащие три вершины.

●●● **19.8.** Укажите число различных орграфов, содержащих V вершин и E ребер, если мы считаем орграфы различными только тогда, когда они не изоморфны.

○ **19.9.** Вычислите верхнюю границу процентного отношения орграфов, содержащих 10 вершин, которые когда-либо могут быть просмотрены каким-либо компьютером в условиях предположений, сделанных в тексте, и с учетом того обстоятельства, что во вселенной имеется 10^{80} электронов и что срок существования вселенной не превысит 10^{20} лет.

19.1. Глоссарий и правила игры

Сформулированные нами определения орграфов фактически идентичны сделанным в главе 17 определениям неориентированных графов (как и в отношении используемых алгоритмов и программ), однако в эти определения имеет смысл внести некоторые коррективы. Небольшие различия в формулировках, касающихся направлений ребер, влекут появление структурных свойств, которые будут в центре внимания данной главы.

Определение 19.1. Ориентированный граф (*или* **орграф**) *представляет собой некоторый набор* **вершин** *плюс некоторый набор* **ориентированных ребер**, *которые соединяют упорядоченные пары вершин (при этом дублированные ребра отсутствуют). Мы говорим, что ребро направлено* **из** *первой вершины* **во** *вторую вершину.*

Как и в случае неориентированных графов, в этом определении мы исключаем наличие дублированных ребер, тем не менее, зарезервируем эту возможность для различных приложений и реализаций, в которых она способна обеспечить положительный практический эффект. Мы недвусмысленно заявляем о допустимости петель в орграфах (и обычно допускаем, что у каждой вершины такая петля имеется), поскольку они играют важную роль в фундаментальных алгоритмах.

Определение 19.2. Ориентированный путь (directed path) *в орграфе есть список вершин, в котором имеется (ориентированное) ребро орграфа, соединяющее каждую вершину списка со следующим элементом этого списка. Мы говорим, что вершина t* **достижима (reachable)** *из вершины s, если существует ориентированный путь из s в t.*

Мы принимаем соглашение, согласно которому каждая вершина достижима сама из себя, и обычно реализуем это свойство за счет того, что в представление нашего орграфа включаем петли.

Для понимания многих алгоритмов, описываемых в данной главе, требуется понимание свойства связности орграфа и того, как это свойство отражается на базовом процессе перемещения от одной вершины к другой вдоль ребер графа. Достижение такого понимания для орграфов намного сложнее, чем для неориентированных графов. Например, иногда достаточно одного взгляда, чтобы сказать, является ли небольшой неориентированный граф связным или содержит ли он циклы; эти свойства довольно непросто обнаружить в орграфах, что демонстрирует типичный пример на рис. 19.3.

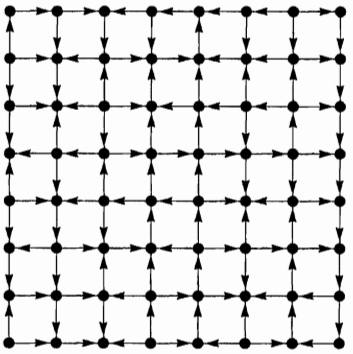


РИСУНОК 19.3. СЕТЧАТЫЙ ОРГРАФ

Этот небольшой по размерам граф имеет много общего с крупной сетью, которую мы впервые рассматривали в главе 1. Отличие заключается в том, что на каждой линии сетки имеется ориентированное ребро, направление которого выбрано случайно. Даже несмотря на то, что в рассматриваемом графе содержатся всего лишь несколько вершин, его свойства связности не так-то просто установить. Существует ли ориентированная цепь из вершины в верхнем левом углу в вершину в нижнем правом углу?

В то время как подобные примеры подчеркивают различия, важно иметь в виду, что то, что человек считает трудной задачей, может совпадать, а может и не совпадать с тем, что трудной задачей считает программа — например, написание класса DFS, предназначенного для обнаружения циклов в орграфе не превосходит по трудности задачу поиска циклов в неориентированном графе. Что более важно, для орграфов и графов характерны существенные структурные различия. Например, тот факт, что вершина *t* достижима из вершины *s* в орграфе, ничего не говорит о том, достижима ли *s* из *t*. Это различие очевидно, но в то же время существенно, в чем мы убедимся несколько позже.

Как мы уже отмечали в разделе 17.3, представления, использованные нами для орграфов, по существу те же, что и представления, использованные для неориентированных графов. В самом деле, они более наглядны, поскольку ребро в них представлено только один раз, как это видно из рис. 19.4. В представлении графа в виде списков смежных вершин ребро **s-t** представлено как узел списка, содержащего **t** в связном списке, который соответствует вершине **s**. В представлении графа в виде матрицы смежности мы вынуждены поддерживать полную матрицу размера V x V и представлять в ней ребро единицей на пересечении строки *s* и столбца *t*. Мы не ставим 1 в строку **t** и столбец **s**, если в графе нет ребра **t-s**. В общем случае матрица смежности для орграфа не является симметричной относительно главной диагонали.

В этих представлениях не существует различий между неориентированными графами и ориентированными графами с петлей в каждой вершине и двумя ориентированными ребрами для каждого ребра, соединяющего разные вершины неориентированного графа (по одному в каждом направлении). Следовательно, можно использовать алгоритмы, которые мы разрабатываем в этой главе для орграфов, для обработки неориентированных графов, естественно, при соответствующей интерпретации результатов. Наряду с этим, мы используем программы, которые рассматривали в главе 17, в качестве основы для программ обработки орграфов: программы 17.7–17.9, реализующие классы **DenseGRAPH** и **SparseMultiGRAPH**, строят орграфы, когда конструктор во втором аргументе получает значение **true**.

Полустепень захода (*indegree*) вершины орграфа есть число ориентированных ребер, которые ведут в эту вершину. *Полустепень исхода* (*outdegree*) вершины орграфа есть число ориентированных ребер, которые исходят из этой вершины. Ни одна из вершин орграфа не достижима из вершины с полустепенью исхода 0, эта вершина получила название *сток* (*sink*); вершина с полустепенью захода 0 получила название *исток* (*source*), она не достижима ни из одной другой вершины орграфа. Орграфы, в которых допускается существование петель и в которых каждая вершина обладает полустепенью исхода, равной 1, называется *картой* (*map*) (отображение самого на себя множества целых чисел в диапазоне от 0 до $V-1$). Мы легко можем подсчитать полустепени захода и исхода для каждой вершины и найти стоки и исходы за линейное время и с использованием пространства памяти, пропорционального V; при это применяются векторы, индексированные именами вершин.

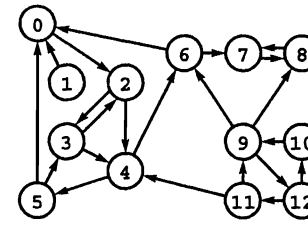
Обращение (reverse) орграфа – это орграф, который мы получаем, поменяв ориентацию всех ребер орграфа на обратную. На рис. 19.5 представлено обращение орграфа, изображенного на рис. 19.1, и его представления. Мы используем обращение орграфа в алгоритмах, когда хотим знать, откуда исходят ребра, поскольку стандартные представления, которыми мы пользуемся, показывают нам только куда ребра идут. Например, в результате обращения орграфа меняются местами полустепени исхода и захода.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	1	0	0	0	1	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0
2	1	0	1	1	0	0	0	0	0	0	0	0	0
3	0	0	1	1	0	1	0	0	0	0	0	0	0
4	0	0	1	1	1	0	0	0	0	0	0	1	0
5	0	0	0	0	1	1	0	0	0	0	0	0	0
6	0	0	0	0	1	0	1	0	0	1	0	0	0
7	0	0	0	0	0	0	1	1	1	0	0	0	0
8	0	0	0	0	0	0	0	1	1	1	0	0	0
9	0	0	0	0	0	0	0	0	0	1	1	1	0
10	0	0	0	0	0	0	0	0	0	0	1	0	1
11	0	0	0	0	0	0	0	0	0	0	0	1	1
12	0	0	0	0	0	0	0	0	0	1	0	0	1

РИСУНОК 19.4. ПРЕДСТАВЛЕНИЯ ОРГРАФА

В представлениях орграфа в виде матрицы смежности и в виде списков смежных вершин каждое ребро фигурирует только один раз, как видно из представления орграфа, показанного на рис. 19.1, в виде матрицы смежности (сверху) и в виде списков смежных вершин (внизу). Оба эти представления включают петли в каждой вершине, которые обычно используются при обработке орграфов.

В случае представления графа в виде матрицы смежности мы могли бы вычислить его обращение, скопировав матрицу и выполнив ее транспонирование (меняя местами строки и столбцы). Если мы знаем, что граф в дальнейшем не будет изменяться, мы фактически можем использовать его обращение без каких-либо дополнительных вычислений, путем простой перестановки вершин в *ссылках (references)* на соответствующие ребра, когда мы ссылаемся на обращение. Например, ребру **s-t** орграфа *G* соответствует значение 1 элемента **adj[s][t]**. Таким образом, если бы мы вычислили обращение *R* орграфа *G*, он содержал бы 1 в элементе **adj[t][s]**. Однако нет необходимости делать это, если мы построим свою реализацию на основе функции проверки ребра **edge(s,t)**, поскольку для переключения на работу с обращением достаточно поменять такую ссылку на **edge(t,s)**. Эта возможность может показаться очевидной, но ею часто пренебрегают. Что касается представления орграфа в виде списков смежных вершин, то это совершенно другая структура данных, и на ее построение, как показывает программа 19.1, тратится время, пропорциональное числу ребер орграфа.



	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	0	1	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	0	0	0	0	0	0	0	0
3	0	0	1	1	1	0	0	0	0	0	0	0	0
4	0	0	0	0	1	1	1	0	0	0	0	0	0
5	1	0	0	1	0	1	0	0	0	0	0	0	0
6	1	0	0	0	0	0	1	1	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	0	0	0	0
8	0	0	0	0	0	0	0	0	1	1	0	0	0
9	0	0	0	0	0	0	1	0	1	1	0	0	1
10	0	0	0	0	0	0	0	0	0	1	1	0	0
11	0	0	0	0	1	0	0	0	0	1	0	1	0
12	0	0	0	0	0	0	0	0	0	0	1	1	1

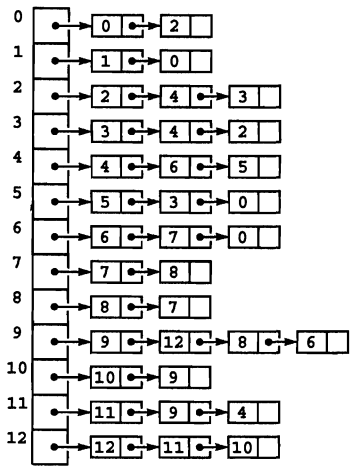


РИСУНОК 19.5. ОБРАЩЕНИЕ ОРГРАФА

Изменение направлений ребер орграфа на обратные соответствует транспонированию матрицы смежности, однако оно требует переделки списков смежных вершин (см. рис. 19.1 и 19.4).

Программа 19.1. Обращение орграфа

Данная функция добавляет ребра орграфа, переданного в первом аргументе, в орграф, указанный во втором аргументе. Она использует два спецификатора шаблона, так что граф может иметь различные представления.

```
template <class inGraph, class outGraph>
void reverse(const inGraph &G,
             outGraph &R)
  {
   for (int v = 0; v < G.V(); v++)
     {typename inGraph::adjIterator A(G, v);
      for (int w = A.beg(); !A.end();
                w = A.nxt())
        R.insert(Edge(w, v));
     }
  }
```

Еще одна точка зрения, к которой мы обратимся в главе 22, предусматривает наличие двух представлений каждого ребра, как это имеет место в случае неориентированных графов (см. раздел 17.3), но с дополнительным разрядом, указывающим направление ребра. Например, чтобы воспользоваться этим методом в представлении орграфа в виде списка смежных вершин,

мы будем представлять ребро **s-t** узлом **t** в списке смежных вершин вершины **s** (установлено такое значение разряда направления, которое указывает, что движение вдоль ребра в направлении от **s** в **t** является обратным обходом ребра) и узлом **s** в списке смежных вершин вершины **t** (установлено такое значение разряда направления, которое указывает, что движение вдоль ребра в направлении от **t** в **s** представляет собой прямой обход ребра). Подобное представление поддерживает алгоритмы, которые выполняют обходы ребра орграфа в обоих направлениях. Вообще говоря, в подобных случаях удобно включать указатели, соединяющие оба представления каждого узла. Мы отложим подробное исследование такого представления до главы 22, в которой оно играет существенную роль.

В орграфах, по аналогии с неориентированными графами, мы говорим об ориентированных циклах. Такими циклами являются ориентированные пути, ведущие от некоторой вершины к ней самой, и о простых ориентированных путях и циклах, в которых все вершины и ребра различны. Обратите внимание на тот факт, что путь **s-t-s** есть цикл длиной 2 в орграфе, в то время как цикл в неориентированном графе должен проходить через три различных вершины.

Во многих приложениях орграфов мы не рассчитываем, что столкнемся с какими бы то ни было циклами, в таких случаях нам приходится иметь дело с другим типом комбинаторного объекта.

Определение 19.3. Ориентированный ациклический граф (directed acyclic graph), *или* **граф DAG**, — *это орграф, не содержащий направленных циклов.*

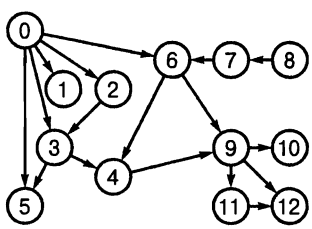
Графы DAG могут встретиться, например, в приложениях, в которых мы используем орграфы для моделирования отношения предшествования. Графы DAG не только вполне естественно используются в этих, но и во многих других приложениях, в частности, как мы убедимся далее, при изучении структур орграфов общего вида. Пример графа DAG показан на рис. 19.6.

В силу упомянутых выше свойств, направленные циклы являются ключевым фактором для понимания связности орграфов, не являющихся графами DAG. Неориентированный граф связен, если существует путь из каждой его вершины в любую другую вершину; что касается орграфов, то мы модифицируем данное выше определение следующим образом:

Определение 19.4. *Орграф называется* **сильно связным (strongly connected)**, *если каждая его вершина достижима из любой другой вершины.*

Граф, изображенный на рис. 19.1, не принадлежит к числу сильно связанных, поскольку, например, в нем нет ориентированных путей из вершины **9** через вершину **12** к любым другим вершинам графа.

Прилагательное *сильно* указывает на то, что каждая пара вершин связана более сильным отношением, чем достижимость. Мы говорим, что пара вершин **s** и **t**



2-3 11-12 3-5 6-4
0-6 9-12 8-7 6-9
0-1 9-10 5-4 7-6
2-0 9-11 0-5

РИСУНОК 19.6. ОРИЕНТИРОВАННЫЙ АЦИКЛИЧЕСКИЙ ГРАФ, ИЛИ ГРАФ DAG

В этом графе нет циклов, это его свойство непосредственно не просматривается из списка ребер и его совсем непросто обнаружить даже путем исследования чертежа графа.

любого графа *сильно связана* (*strongly connected*) или что эти вершины являются *взаимно достижимыми* (*mutually reachable*), если существует ориентированный путь из **s** в **t** и ориентированный путь из **t** в **s**. (Из того факта, что в соответствии с нашим соглашением каждая вершина достижима из самой себя, следует, что каждая вершина сильно связана сама с собой.) Орграф сильно связан тогда и только тогда, когда все пары вершин сильно связаны. Определяющей характеристикой сильно связных орграфов является свойство, которое в случае связных неориентированных графов мы воспринимаем как само собой разумеющееся: если существует путь из **s** в **t**, то существует путь из **t** в **s**. В случае неориентированных графов нам этот факт известен, поскольку *один и тот же* путь, пройденный в обратном направлении, отвечает всем требованиям определения; в случае орграфа это уже будет другой путь.

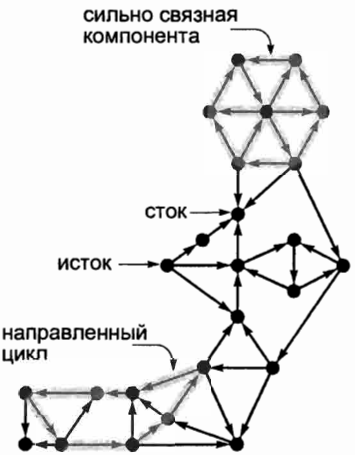


РИСУНОК 19.7. ТЕРМИНОЛОГИЯ ОРГРАФОВ

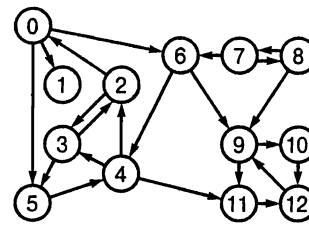
Истоки (вершины, в которые не ведет ни одно ребро) и стоки (вершины, из которых не выходит ни одно ребро) легко обнаружить на чертежах орграфа, подобных данному, но в то же время направленные циклы и сильно связные компоненты обнаружить намного труднее. Какой направленный цикл данного орграфа является самым длинным? Сколько сильно связных компонент с числом вершин, большим одного, содержится в этом орграфе?

По-другому о том факте, что пара вершин сильно связана, можно сказать следующим образом: они лежат на некотором ориентированном циклическом пути. Напомним, что мы используем термин *циклический путь (cyclic path)* вместо термина *цикл (cycle)*, дабы показать, что путь не обязательно должен быть простым. Например, на рис. 19.1 вершины **5** и **6** сильно связаны, поскольку вершина **6** достижима из вершины **5** через прямой путь **5-4-2-0-6**, а **5** достижима из **6** по ориентированному пути **6-4-3-5**. Из существования этих путей следует, что вершины **5** и **6** лежат на ориентированном циклическом пути **5-4-2-0-6-4-3-5**, но они не лежат в каком-то (простом) ориентированном цикле. Обратите внимание на тот факт, что ни один граф DAG, содержащий более одной вершины, не может быть сильно связным.

Как и простая связность в неориентированных графах, это отношение транзитивно: если **s** сильно связана с **t**, а **t** сильно связана с **u**, то **s** сильно связана с **u**. Сильная связность есть отношение эквивалентности, которое разделяет вершины графа на классы эквивалентности, содержащие вершины, сильно связанные друг с другом (см. раздел 19.4, в котором отношения эквивалентности рассматриваются более подробно). Опять-таки, сильная связность наделяет орграфы свойством, которое мы считаем само собой разумеющимся в отношении связности неориентированных графов.

Свойство 19.1. *Орграф, не принадлежащий к классу сильно связных графов, содержит некоторый набор* **сильно связных компонент (strongly connected components)** (*или, для краткости,* **strong components – сильных компонент**), *которые представляют собой максимальные сильно связные подграфы, и некоторый набор ориентированных ребер, идущих от одной компоненты к другой.*

Доказательство: Подобно компонентам неориентированных графов, сильные компоненты в орграфах суть индуцированные подграфы, содержащие некоторые подмножества вершин графа: каждая вершина есть в точности одна сильная компонента. Чтобы доказать этот факт, сначала заметим, что каждая вершина принадлежит минимум одной сильной компоненте, которая содержит, по меньшей мере, саму вершину. Далее отметим, что каждая вершина принадлежит максимум одной сильной компоненте: если бы та или иная вершина принадлежала сразу двум различным компонентам, то существовал бы путь, проходящий через эту вершину и соединяющий любые вершины этих компонент друг с другом, в обоих направлениях, что противоречит предположению о максимальности обеих компонент. ■



	0	1	2	3	4	5	6	7	8	9	10	11	12
id	2	1	2	2	2	2	2	3	3	0	0	0	0

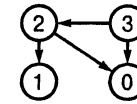


РИСУНОК 19.8. СИЛЬНЫЕ КОМПОНЕНТЫ И ЯДРО DAG

Рассматриваемый орграф (вверху) состоит из четырех сильных компонент, как это следует из массива ***id*** *(в центре), индексированного именами вершин (в качестве индексов взяты произвольные целые числа). Компонента* ***0*** *содержит вершины* ***9, 10, 11*** *и* ***12****; компонента* ***1*** *состоит из единственной вершины* ***1****; компонента* ***2*** *содержит вершины* ***0, 2, 3, 4, 5*** *и* ***6****, а компонента* ***3*** *состоит из вершин* ***7*** *и* ***8****. Если мы начертим граф, определяемый ребрами, соединяющими различные компоненты, то получим DAG (внизу).*

Например, орграф, который содержит единственный направленный цикл, состоит из одной сильной компоненты. С другой стороны, каждая вершина в графе DAG — это сильная компонента, поэтому каждое ребро в графе DAG ведет из одной компоненты в другую. В общем случае, не все ребра орграфа содержатся в сильных компонентах. Такое положение отличается от аналогичной ситуации для связных компонент в неориентированных графах, в которых каждая вершина, *а также* каждое ребро принадлежит некоторой связной компоненте, но оно подобно аналогичной ситуации для реберно-связных компонент в неориентированных графах. Сильные компоненты в орграфах соединены между собой ребрами, которые ведут из вершины одной компоненты в вершину другой, но не возвращаются назад.

Свойство 19.2. *Пусть задан орграф D, определим еще один орграф K(D), в котором одна вершина соответствует каждой сильной компоненте орграфа D, и одно ребро K(D) соответствует каждому ребру орграфа D, которое соединяет вершины различных сильных компонент (соединяет вершины в K, соответствующие сильным компонентам, которые он соединяет в D). Тогда K(D) будет граф DAG (который мы будем называть* **базовым графом DAG (kernel DAG)**, или **ядром DAG**, *графа D).*

Доказательство: Если бы орграф *K(D)* содержал направленный цикл, то вершины двух различных сильных компонент орграфа *D* оказались бы в направленном цикле, что привело бы к противоречию. ■

На рис.19.8 показаны сильные компоненты и базовый граф DAG на демонстрационном примере орграфа. Мы будем рассматривать алгоритмы поиска сильных компонент и построения ядра графа DAG в разделе 19.6.

Из этих определений, свойств и примеров становится ясно, что нужно быть предельно точным при использовании путей в орграфе. Мы должны исследовать, по меньшей мере, три следующих ситуации:

Связность. Мы зарезервируем термин *связный (connected)* для неориентированных графов. В случае орграфов мы можем утверждать, что две вершины связаны, если они связаны в неориентированном графе, получаемом, когда направления ребер игнорируются, тем не менее, мы будем избегать подобной трактовки.

Достижимость. Мы говорим, что вершина **t** орграфа достижима из вершины **s**, если существует ориентированный путь из **s** в **t**. Как правило, мы избегаем употреблять термин *достижимый (reachable)*, когда имеем дело с неориентированными графами, хотя можем считать его эквивалентным понятию *connected (связный)*, поскольку идея достижимости одной вершины из другой имеет определенный практический смысл в некоторых неориентированных графах (в частности, в графах, служащих представлением лабиринтов).

Сильная связность. Две вершины орграфа сильно связаны, если они взаимно достижимы; в неориентированных графах из связности двух вершин следует существование путей из одной вершины в другую. Сильная связность в орграфах подобна в некотором смысле реберной связности в неориентированных графах.

Мы хотим обеспечить поддержку операции АТД орграфа, которые принимают в качестве аргументов две вершины **s** и **t** и позволяют нам проверить:

- достижима ли **t** из **s**;
- существует ли сильная связность вершин **s** и **t** (взаимная достижимость).

Какие ресурсы мы можем выделить для осуществления этих операций? Как можно было убедиться в разделе 17.5, поиск в глубину обеспечивает простое решение задачи связности в неориентированных графах. При этом требовались затраты времени, пропорциональные V, но если есть возможность изыскать временной ресурс на предварительную обработку, пропорциональный $V+E$, и выделить пространство памяти, пропорциональное V, то можно отвечать на запросы о связности графа за постоянное время. Далее в этой главе мы исследуем алгоритмы выявления сильной связности, которые имеют аналогичные рабочие характеристики.

Однако главная трудность в достижении поставленной перед нами цели заключается в том, что ответы на запросы относительно достижимости в орграфах получить намного труднее, чем на запросы о связности или сильной связности орграфа. В этой главе мы будем изучать классические алгоритмы, для выполнения которых требуется время, пропорциональное VE, и пространство памяти, пропорциональное V^2. Кроме того, мы разработаем реализации, которые будут способны выдавать ответы относительно достижимости на орграфах определенного типа за постоянное время при линейном расходе памяти и времени на предварительную обработку, и исследуем трудности достижения этой оптимальной производительности для всех орграфов.

Упражнения

▷ **19.10.** Покажите структуру списков смежных вершин, построенных программой 17.9 для орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

▷ **19.11.** Напишите программу, генерирующую случайные разреженные орграфы для такого специально выбранного набора значений V и E, что их можно использовать для подходящих эмпирических тестов на орграфах, построенных на базе модели со случайными ребрами.

▷ **19.12.** Напишите программу, генерирующую случайные разреженные графы для такого специально выбранного набора значений V и E, что их можно использовать для подходящих эмпирических тестов на графах, построенных на базе модели со случайными ребрами.

○ **19.13.** Напишите программу, которая генерирует орграфы путем соединения вершин, упорядоченных в решетку размера $\sqrt{V} \times \sqrt{V}$, с соседними вершинами, при этом направления ребер выбираются случайным образом (см. рис. 19.11).

○ **19.14.** Расширьте возможности программы из упражнения 19.13, добавив R дополнительных случайных ребер (все возможные ребра выбираются с равной вероятностью). Для больших значений R уменьшите решетку таким образом, чтобы общее число ребер оставалось примерно равным V. Проверьте полученную программу способом, описанным в упражнении 19.11.

○ **19.15.** Внесите такие изменения в вашу программу из упражнения 19.14, чтобы дополнительное ребро выходило из вершины **s** в вершину **t** с вероятностью, обратно пропорциональной эвклидову расстоянию между **s** и **t**.

○ **19.16.** Напишите программу, которая генерирует в единичном интервале V случайных интервалов длиной d каждый, после чего строит орграф с ребром из интервала **s** в интервал **t** тогда и только тогда, когда, по меньшей мере, одна из граничных точек **s** попадает в **t** (см. упражнение 17.75). Определите, как выбрать d таким, чтобы ожидаемое число ребер было равным E. Протестируйте свою программу методом, описанным в упражнении 19.11 (для разреженных орграфов), и методом, описанным в упражнении 19.12 (для насыщенных орграфов).

● **19.17.** Напишите программу, которая выбирает V вершин и E ребер из реального орграфа, который вам удалось найти при выполнении упражнения 19.1. Протестируйте свою программу, как описано в упражнениях 19.11 (для разреженных орграфов) и 19.12 (для насыщенных орграфов).

● **19.17.** Напишите программу, которая с одинаковой вероятностью строит любой из возможных орграфов с V вершинами и с E ребрами (см. упражнение 17.70). Протестируйте свою программу, как описано в упражнениях 19.11 (для разреженных орграфов) и 19.12 (для насыщенных орграфов).

▷ **19.19.** Реализуйте класс, который предоставляет клиентским программам возможность определить полустепени захода и исхода любой заданной вершины орграфа за постоянное время после предварительной подготовки в конструкторе, на что должно тратиться линейное время. Затем добавьте функции-элементы, которые возвращают число истоков и стоков за постоянное время.

○ **19.20.** Воспользуйтесь программой из упражнения 19.19 с тем, чтобы найти среднее число истоков и стоков в орграфах различных типов (см. упражнение 19.11–19.18).

▷ **19.21.** Покажите структуру списков смежных вершин, которая получается, когда вы используете программу 19.1 для построения обращения орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

○ **19.22**. Опишите обращение карты.

19.23. Разработайте класс орграфа, который предоставляет клиентам возможность непосредственно ссылаться как на орграфы, так и на их обращения, и получите реализацию для любого представления, которое поддерживает запросы к функции **edge**.

19.24. Постройте альтернативную реализацию класса, построенного в упражнении 19.23, которая поддерживает обе ориентации ребер в списках смежных вершин.

▷ **19.25**. Опишите семейство сильно связанных орграфов с V вершинами, не содержащих (простых) направленных циклов длиной больше 2.

○ **19.26**. Получите сильные компоненты и ядро DAG орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

○ **19.27**. Получите ядро DAG решеточного орграфа, показанного на рис. 19.3.

19.28. Какое число графов имеют V вершин, каждая из которых имеет степень исхода, равную k?

● **19.29**. Какое значение принимает ожидаемое число различных представлений случайного орграфа в виде списков смежных вершин? *Указание*: Разделите общее число возможных представлений на общее число орграфов.

19.2. Анатомия поиска DFS в орграфах

Мы можем воспользоваться программой поиска в глубину на неориентированных графах, которая рассматривалась в главе 18, для обхода каждого ребра и каждой вершины орграфа. Основной принцип этого рекурсивного алгоритма можно сформулировать следующим образом: чтобы посетить каждую вершину, достижимую из данной вершины, мы помечаем эту вершину как посещенную, затем (рекурсивно) посещаем все вершины, в которые можно пройти из каждой вершины, внесенные в список смежных с ней вершин.

В неориентированных графах возможны два представления одного ребра, однако второе представление, которое употребляется при поиске в глубину, всегда приводит в помеченную вершину, в силу чего оно игнорируется (см. раздел 18.2). В орграфе употребляется только одно представление каждого ребра, поэтому мы можем рассчитывать на то, что алгоритмы поиска в глубину в орграфах упростятся. Однако орграфы по своей природе являются более сложными комбинаторными объектами, так что эти ожидания не имеют под собой почвы. Например, деревья поиска, которые используются с целью лучшего понимания функционирования алгоритма, обладают более сложной структурой для орграфов, нежели для неориентированных графов. Подобное осложнение делает алгоритмы обработки орграфов более трудными для разработки. Например, как мы вскоре убедимся, гораздо труднее проводить исследования ориентированных путей в орграфах, чем исследовать пути в неориентированных графах.

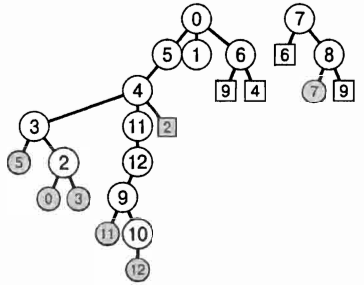
Подобно тому, как мы поступали в главе 18, воспользуемся двумя терминами. Термин *стандартный поиск в глубину на списках смежных вершин (standard adjacency-lists DFS)* будет обозначать процесс вставки последовательности ребер в АТД орграфа, представленного в виде списков смежных вершин (конструктор из программы 17.9, при вызове которого во втором аргументе передается значение **true**), с последующим выполнением поиска в глубину, например, с помощью программы 18.3. Второй, и аналогичный, термин *стандартный поиск в глубину на матрице смежности (standard adjacency-matrix DFS)* будет обозна-

чать процесс вставки некоторой последовательности ребер в АТД орграфа, представленного в виде матрицы смежности (конструктор из программы 17.7, при вызове которого во втором аргументе передается значение **true**), с последующим выполнением поиска в глубину, например, с помощью программы 18.3.

Например, на рис. 19.9 приводится дерево рекурсивного вызова, которое описывает, как действует алгоритм стандартного поиска в глубину на орграфе, представленном в виде списков смежных вершин на примере орграфа из рис. 19.1. Так же, как и в случае неориентированных графов, подобные деревья содержат внутренние узлы, соответствующие вызовам рекурсивной функции поиска в глубину для каждой вершины, при этом связи, ведущие во внешние узлы, соответствуют ребрам, которые приводят в уже просмотренные вершины. Классификация узлов и связей дает нам информацию о поиске (и об орграфе), однако такая классификация применительно к орграфам существенно отличается от классификации применительно к неориентированным графам.

В неориентированных графах каждая связь в дереве поиска в глубину относится к одному из четырех классов, в зависимости от того, соответствует ли она ребру графа, которое ведет к рекурсивному вызову, и в зависимости от принадлежности ребра, просматриваемого алгоритмом поиска в глубину, к первому или второму виду представления. В орграфах имеет место соответствие один к одному между связями дерева и ребрами графа, они попадают в один из четырех различных классов:

- Класс ребер, представляющих рекурсивный вызов (*древесные (tree)* ребра).
- Класс ребер, ведущих из той или иной вершины к ее предшественнику в дереве поиска в глубину (*обратные (back)* ребра).
- Класс ребер, ведущих из вершины к потомку в его дереве поиска в глубину (*прямые (down)* ребра).



	0	1	2	3	4	5	6	7	8	9	10	11	12
pre	0	9	4	3	2	1	10	11	12	7	8	5	6
post	10	8	0	1	6	7	9	12	11	3	2	5	4

РИСУНОК 19.9. ЛЕС DFS ДЛЯ ОРГРАФОВ

Данный лес описывает стандартный поиск в глубину на орграфе, представленном в виде списков смежных вершин на примере графа, показанного на рис. 19.1. Внешние узлы представляют ранее просмотренные внутренние узлы с теми же метками; во всем остальном этот лес является представлением орграфа, все ребра которого направлены вниз. В рассматриваемом лесе используются четыре типа ребер: древесные ребра, ведущие во внутренние узлы; обратные ребра, ведущие во внешние узлы, представляющие предшественников (заштрихованные кружки); нисходящие ребра, ведущие во внешние узлы, представляющие потомков (заштрихованные квадратики); и поперечные ребра, ведущие во внешние узлы, не являющиеся ни предшественниками, ни потомками (незаштрихованные квадратики). Мы можем определить тип ребер, ведущих в посещенные узлы, сравнивая их номера узлов-источников и узлов назначения при обходе в прямом и обратном порядке (внизу):

Прямой порядок	Обратный порядок	Пример	Тип ребра
<	>	4-2	Нисходящее
>	<	2-0	Обратное
<	>	7-6	Поперечное

Например, 7-6 представляет собой поперечное ребро, поскольку номер узла 7 в прямом и обратном порядке обхода больше, чем номер узла 6.

- Класс ребер, ведущих из одной вершины в другую, не являющуюся ни предшественником, ни потомком в дереве поиска в глубину (*поперечные (cross)* ребра).

Древесное ребро есть ребро, ведущее в непосещенную вершину, соответствующую рекурсивному вызову при поиске в глубину. Обратные, поперечные и прямые ребра ведут в посещенные вершины. Чтобы определить тип заданного ребра, мы используем нумерацию в прямом и обратном порядке (очередность, в которой посещение узлов совершается, соответственно, при прямом и обратном обходе леса).

Свойство 19.3. *В лесе поиска в глубину (DFS), соответствующем орграфу, ребро, которое ведет в посещенную вершину, есть обратное ребро, если оно ведет в узел с более высоким номером при обходе в обратном порядке; в противном случае это поперечное ребро, если оно ведет в узел с более низким номером при обходе в прямом порядке, и прямое ребро, если оно ведет в узел с более высоким номером при обходе в прямом порядке.*

Доказательство: Все эти факты непосредственно следуют из определений. Предшественник узла в дереве поиска в глубину имеет более низкие номера при обходе в прямом порядке и более высокие номера в обратном порядке; их потомки имеют более высокие номера в прямом порядке и более низкие номера в обратном порядке. Верно также и то, что оба эти номера меньше в ранее посещенных узлах других деревьев DFS, и оба эти номера выше в тех узлах, которые предстоит посетить в других деревьях, однако нам не потребуется писать программный код для проверки подобных случаев. ■

Программа 19.2 реализует класс DFS, который определяет тип каждого ребра графа. Рисунок 19.10 служит иллюстрацией ее работы на примере орграфа, представленного на рис. 19.1. Проверка, проводимая с тем, чтобы убедиться, ведет ли то или иное ребро в узел с более высоким номером при обратном порядке обхода, эквивалентна проверке, был ли вообще присвоен узлу номер при обратном обходе. Любой узел, которому был присвоен номер в прямом порядке, но которому еще не был присвоен номер в обратном порядке, в дереве DFS является предшественником и получит более высокий номер в обратном порядке, чем номер в обратном порядке текущего узла.

Как мы могли убедиться в главе 17, в которой рассматривались неориентированные графы, типы ребер — это и свойства динамики поиска, а не только свойства графа. В самом деле, как следует из рис. 19.11, различные леса DFS одного и того же графа могут существенно различаться по характеру. Например, четность номеров деревьев леса DFS зависит от начальной вершины.

Тем не менее, несмотря на все эти различия, несколько классических алгоритмов обработки орграфов способны определить свойства орграфов благодаря тому, что

```
0-0 древесное
  0-5 древесное
    5-4 древесное
      4-3 древесное
        3-5 обратное
        3-2 древесное
          2-0 обратное
          2-3 обратное
      4-11 древесное
        11-12 древесное
          12-9 древесное
            9-11 обратное
            9-10 древесное
              10-12 обратное
      4-2 прямое
  0-1 древесное
  0-6 древесное
    6-9 поперечное
    6-4 поперечное
7-7 древесное
  7-6 поперечное
  7-8 древесное
    8-7 обратное
    8-9 поперечное
```

РИСУНОК 19.10. ТРАССИРОВКА ОРГРАФА ПОИСКА В ГЛУБИНУ

Трассировка поиска в глубину представляет собой выходные результаты программы 19.2 для примера орграфа, показанного на рис. 19.1. Она в точности соответствует обходу в прямом порядке дерева DFS на рис. 19.9.

они предпринимают соответствующие действия, когда сталкиваются различными типами ребер во время поиска в глубину. Например, рассмотрим следующую фундаментальную задачу.

Обнаружение направленного цикла. Существуют ли в заданном орграфе направленные циклы? (Является ли рассматриваемый орграф графом DAG?) В неориентированных графах любое ребро, ведущее в посещенную вершину, означает наличие цикла в таком графе; в орграфе мы должны в этом плане сосредоточить внимание на обратных ребрах.

Программа 19.2. DFS на орграфе

Данный класс DFS использует нумерацию в прямом и обратном порядке для того, чтобы показать, какую роль играет каждое ребро графа в поиске в глубину (см. рис. 19.10).

```
template <class Graph> class DFS
{ const Graph &G;
  int depth, cnt, cntP;
  vector<int> pre, post;
  void show(char *s, Edge e)
    { for (int i = 0; i < depth; i++) cout << " ";
      cout << e.v << "-" << e.w << s << endl; }
  void dfsR(Edge e)
    { int w = e.w; show(" tree", e);
      pre[w] = cnt++; depth++;
      typename Graph::adjIterator A(G, w);
      for (int t = A.beg(); !A.end(); t = A.nxt())
      { Edge x(w, t);
        if (pre[t] == -1) dfsR(x);
        else if (post[t] == -1) show(" back", x);
        else if (pre[t] > pre[w]) show(" down", x);
        else show(" cross", x);
      }
      post[w] = cntP++; depth--;
    }
public:
  DFS(const Graph &G) : G(G), cnt(0), cntP(0),
    pre(G.V(), -1), post(G.V(), -1)
    { for (int v = 0; v < G.V(); v++)
        if (pre[v] == -1) dfsR(Edge(v, v)); }
};
```

Свойство 19.4. *Орграф является графом DAG тогда и только тогда, когда, воспользовавшись алгоритмом поиска в глубину для проверки каждого ребра, мы не сталкиваемся с обратными ребрами.*

Доказательство: Любое обратное ребро принадлежит некоторому направленному циклу, который состоит из этого ребра плюс путь в дереве, соединяющий два соответствующих узла, так что мы не найдем ни одного обратного ребра при использовании поиска в глубину на DAG. Чтобы доказать обратное утверждение, мы покажем, что если в орграфе имеется цикл, то поиск в глубину выходит на обратное ребро. Предположим, что v есть первая из вершин цикла, на которую выходит поиск в глубину. Эта вершина имеет наименьший номер в прямом порядке обхода всех вершин цикла. В силу этого обстоятельства, ребро, которое ведет в нее, будет обратным ребром: на него мы выйдем во время рекурсивного вызова для вершины v (доказательство того, что мы таки на него обязательно выйдем, приводится в свойстве 19.5), при этом оно указывает из одного из узлов цикла на v, т.е. на узел с меньшим номером в обратном порядке (см. свойство 19.3). ■

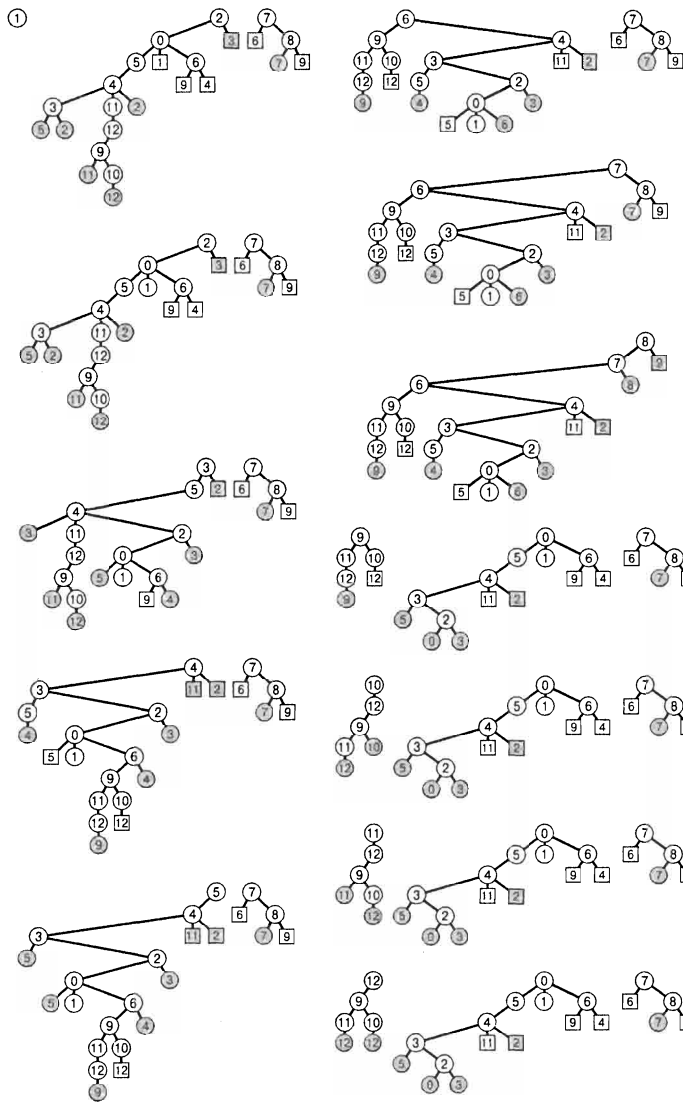


РИСУНОК 19.11. ЛЕС ПОИСКА В ГЛУБИНУ НА ОРГРАФЕ

Изображенные на рисунке леса описывают поиск DFS (depth first search – поиск в глубину) на том же графе, что и представленный на рис. 19.9, в рамках которого функция поиска на графе проверяет вершины (и вызывает эту рекурсивную функцию для непосещенных вершин) в порядке ***s, s+1, ..., 0, 1, ... , s-1*** *для каждого* ***s****. Структура леса определяется как динамикой поиска, так и структурой самого графа. Каждый узел порождает одних и тех же потомков (в порядке следования узлов в его списке смежных вершин) в каждом лесу. Крайнее дерево слева в каждом лесу содержит все вершины, достижимые из его корня, однако проблема достижимости из других вершин усложняется фактом наличия обратных, поперечных и прямых ребер. Даже число деревьев в лесе зависит от выбора начальной вершины, так что совсем не обязательно существование прямого соответствия между деревьями в лесу и сильными компонентами, какое возможно в случае неориентированных графов. Например, мы убеждаемся в том, что все вершины достижимы из вершины* ***8*** *только в том случае, когда мы начинаем поиск в глубину из вершины* ***8****.*

Мы можем преобразовать любой орграф в DAG, выполнив поиск в глубину и удалив любые ребра графа, которые соответствуют обратным ребрам при поиске в глубину. Например, из рис. 19.9 мы видим, что удалив ребра **2-0**, **3-5**, **2-3**, **9-11**, **10-12**, **4-2** и **7-8**, мы получим из орграфа, показанного на рис 19.1, граф DAG. Конкретный граф DAG, который мы получаем подобным способом, зависит от представления преобразуемого графа и от вызванных этим преобразованием последствий для динамических свойств поиска в глубину (см. упражнение 19.37). Этот метод представляет собой способ генерации произвольных графов DAG (см. упражнение 19.76), используемых при тестировании алгоритмов обработки графов DAG.

Обнаружение направленного цикла является простой задачей, однако отличие только что описанного решения от решения, которое рассматривалось в главе 18 для случае неориентированных графов, убеждает нас в необходимости трактовать два этих типа графов как различные комбинаторные объекты, даже если их представления подобны, и одни и те же программы работают на обоих типах в условиях некоторых видов приложений. Согласно нашему определению, возникает впечатление, будто мы используем для решения этой проблемы тот же метод, что и для обнаружения циклов в неориентированных графах (поиск обратных ребер), однако реализация, которой мы воспользовались для неориентированных графов, на орграфе работать не будет. Например, в разделе 18.5 мы очень осторожно подходили к различиям между родительскими связями и обратными связями, поскольку существование родительской связи отнюдь не указывает на наличие цикла (циклы в неориентированных графах должны содержать, по меньшей мере, три вершины). В то же время игнорирование обратных связей, ведущих в орграфах в родительские узлы, некорректно; мы рассматриваем дважды связные пары вершин в орграфах как циклы. Теоретически мы могли бы определить обратные ребра в неориентированных графах так же, как это только что было проделано, однако в таком случае мы должны были бы сделать однозначное исключение для случая с двумя вершинами. И что более важно, мы можем обнаруживать циклы в неориентированных графах за время, пропорциональное V (см. раздел 18.5), однако нам может понадобиться время, пропорциональное E, чтобы обнаружить цикл в орграфе (см. упражнение 19.32).

Одна из главных целей поиска в глубину заключается в том, чтобы обеспечить систематический способ посещения всех вершин и всех ребер графа. В силу этого обстоятельства он дает нам фундаментальный подход к решению проблемы достижимости в орграфах, хотя опять-таки ситуация намного сложнее, нежели в случае неориентированных графов.

Достижимость из единственного источника. Какие вершины заданного орграфа могут быть достигнуты из заданной начальной вершины s? Сколько таких вершин имеется в заданном графе?

Свойство 19.5. *Посредством рекурсивного поиска в глубину, начинающегося в вершине s, мы можем решить задачу достижимости из единого истока для вершины s за время, пропорциональное числу ребер в подграфе, индуцированном достижимыми вершинами.*

Доказательство: Доказательство во многом аналогично доказательству свойства 18.1, однако имеет смысл еще раз подчеркнуть различие между достижимостью в орграфах и связностью в неориентированных графах. Это свойство, несомненно, справедливо для орграфа, который состоит из одной вершины и не содержит ребер. Для любого

орграфа, содержащего более одной вершины, предположим, что это свойство справедливо для всех орграфов, состоящего из меньшего числа вершин. Теперь первое ребро, которое мы выберем из вершины *s*, делит рассматриваемый орграф на два подграфа, индуцированных двумя поднаборами вершин (см. рис 19.12): (*i*) вершинами, достижимыми через ориентированные пути, которые начинаются с этого ребра и не содержат *s* в своем дальнейшем протяжении; и (*ii*) вершины, которые мы не можем достичь через какой-либо ориентированный путь, начинающийся с этого ребра, без возврата в *s*. К этим подграфам мы применяем индуктивное предположение, заключающееся в том, что ориентированные ребра, ведущие в *s*, будут проигнорированы, поскольку эта вершина имеет меньший номер в прямом порядке, чем любая вершина второго подграфа, так что все ориентированные ребра, исходящие из вершин второго подграфа и ведущие в первый подграф, будут проигнорированы. При этом отмечается факт, что не существует ориентированных ребер, ведущих из одной из вершин первого подграфа в любую отличную от *s* вершину второго графа (существование такого ребра приводит к противоречию, ибо вершина назначения должна находиться в первом подграфе). ■

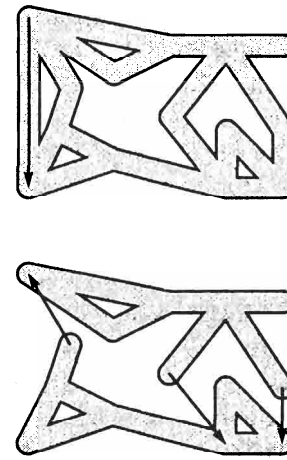


РИСУНОК 19.12. ДЕКОМПОЗИЦИЯ ОРГРАФА

Доказательство методом индукции того факта, что поиск в глубину приводит нас в любое место, достижимое из заданного узла орграфа, по существу ничем не отличается от исследования Тремо. Основное действие изображено здесь в виде лабиринта (вверху), чтобы можно было провести сравнение с рис. 18.4. Мы разбиваем граф на две части меньших размеров (внизу), прожденные двумя множествами вершин: вершинами, которые могут быть достигнуты, если следовать вдоль первого ребра, исходящего из начальной вершины, без ее повторного посещения (нижняя часть), и теми вершинами, которые остаются недостижимыми, если следовать вдоль первого ребра, не возвращаясь в исходную вершину (верхняя часть). Любое ребро, исходящее из вершины, принадлежащей второму множеству, и ведущее в какую-либо вершину первого множества, пропускается, ибо все вершины первого множества помечены еще до того, как начнется поиск во втором подграфе.

В отличие от неориентированных графов, поиск в глубину на орграфе не дает полной информации о достижимости из любого другого узла, отличного от исходного, поскольку ребра дерева являются ориентированными, а поисковые структуры содержат поперечные ребра. Когда мы покидаем какую-либо вершину и отправляемся в обход вниз по дереву, мы не можем быть уверены в том, что существует обратный путь в эту вершину через ребра орграфа; на самом деле такого пути в общем случае не существует. Например, нет такого пути, чтобы вернуться в вершину **4** после выбора ребра дерева **4-11** на рис. 19.9. Более того, если мы игнорируем поперечные и обратные ребра (поскольку они ведут в вершины, которые принимали посещения и больше на могут быть активными), мы игнорируем всю информацию, которую они несут (набор вершин, которые достижимы из вершины назначения отвергнутого ребра). Например, проход на рис. 19.9 вдоль ребра **6-9** представляет собой единственную возможность убедиться в том, что вершины **10**, **11** и **12** достижимы из вершины **6**.

Чтобы определить, какая из вершин достижима из некоторой заданной вершины, мы, по-видимому, должны начинать обход с нового поиска в глубину из этой вершины (см. рис. 19.11). Можем ли мы воспользоваться информацией, накопленной во время предыдущих поисков с тем, чтобы сделать этот процесс для последующих вершин более эффективным? Мы рассмотрим такие аспекты достижимости в разделе 19.7.

При определении, принадлежит ли заданный неориентированный граф к числу связных, мы полагаемся на знание того факта, что вершины соединены со своими предками в дереве DFS посредством (по меньшей мере) одного пути в этом дереве. В противоположность этому, рассматриваемый путь в дереве ведет в неправильном направлении в орграфе: ориентированный путь из конкретной вершины орграфа к ее предку существует только в том случае, если существует обратное ребро из какого-либо ее потомка в эту же вершину или в более дальний ее предок. Более того, связность неориентированных графов для каждой вершины ограничивается деревом DFS с корнем в этой вершине; в отличие от этого, в орграфах поперечные ребра могут перенести нас в любую ранее посещенную часть рассматриваемой структуры поиска, даже если эта часть находится в другом дереве леса DFS. В неориентированных графах мы могли воспользоваться преимуществом этих особенностей свойства связности для того, чтобы идентифицировать каждую вершину со связной компонентой в одном поиске в глубину, а затем использовать эту информацию в качестве основы для операций АТД, выполняемых за постоянное время, с целью определить, являются ли любые две вершины связными. В случае орграфа, как мы уже смогли убедиться в этой главе, подобные цели труднодостижимы.

В этой, равно как и в предыдущих главах, мы не раз подчеркивали, что различные способы выбора непосещенных вершин приводят к различным динамикам поиска в глубину. Что касается орграфов, то структурная сложность деревьев DFS приводит к различиям в динамике поиска, которые еще ярче выражены, чем те, что наблюдались в неориентированных графах. Например, рис. 19.11 может служить иллюстрацией того, что мы получаем заметные различия для орграфов даже в тех случаях, когда мы просто меняем порядок, в котором вершины проверяются высокоуровневыми функциями. Только крохотная часть этих возможностей отображена на этом рисунке — в принципе, каждый из $V!$ различных порядков проверки вершин может приводить к различным результатам. В разделе 19.7 мы будем рассматривать один важный алгоритм, который построен на использовании упомянутой гибкости и выполняет обработку непосещенных вершин на верхнем уровне (корни деревьев DFS) в особом порядке, которые позволяет немедленно выявить сильные компоненты.

Упражнения

19.30. Начертите лес DFS, который получается при применении поиска в глубину к орграфу

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4,

представленному в виде списков смежных вершин.

19.31. Начертите лес DFS, который получается при применении поиска к орграфу

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4,

представленному в виде матрицы смежности.

○ **19.32**. Опишите семейство орграфов с V вершинами и E ребрами, для которых стандартный поиск в глубину, ориентированный на представление графа в виде смежных вершин, требует для обнаружения циклов время, пропорциональное E.

▷ **19.33**. Покажите, что во время выполнения поиска в глубину на орграфе никакое ребро не соединяет какой-либо узел с другим узлом, номера которых как в прямом, так и в обратном порядке меньше соответствующих номеров первого узла.

▷ **19.34**. Покажите все возможные леса орграфа

0-1 0-2 0-3 1-3 2-3,

и сведите в таблицу число древесных, обратных, поперечных и прямых ребер для каждого леса.

19.35. Если обозначить числа древесных, обратных, поперечных и прямых ребер, соответственно, через t, b, c и d, то получим $t + b + c + t = E$ и $t < V$ для любого поиска в глубину на любом орграфе с V вершинами и E ребрами. Какие другие отношения между этими переменными можно обнаружить? Какие из этих значений зависят только от свойств графов и какие из них зависят от динамических свойств поиска в глубину?

▷ **19.36**. Докажите, что каждый исток в конкретном орграфе должен быть корнем некоторого дерева в лесе, соответствующем любому поиску в глубину на этом орграфе.

○ **19.37**. Постройте связный DAG, который есть подграф графа, изображенный на рис. 19.1, за счет удаления пяти ребер (см. рис. 19.11).

19.38. Реализуйте класс орграфа, который предоставляет клиенту возможность проверки того, что заданный орграф на самом деле есть DAG, и постройте реализацию, основанную на поиске в глубину.

○ **19.39**. Воспользуйтесь найденным вами решением упражнения 19.38 для (эмпирической) оценки вероятности того, что случайный орграф с V вершинами и E ребрами представляет собой DAG, для различных типов орграфов (см. упражнения 19.11–19.18).

19.40. Проведите эмпирические исследования с целью определить относительное процентное содержания деревьев, обратных, поперечных и прямых ребер, когда мы проводим поиск в глубину на различных типах орграфов (см. упражнения 19.11–19.18).

19.41. Опишите, как построить последовательность ориентированных ребер V вершин, для которых не существует ни поперечных, ни прямых ребер и для которых число обратных ребер пропорционально V^2 при стандартном поиске в глубину, ориентированном на представление графа в виде списков смежных вершин.

○ **19.42**. Опишите, как построить последовательность ориентированных ребер V вершин, для которых не существует ни обратных, ни прямых ребер и для которых число поперечных ребер пропорционально V^2 при стандартном поиске в глубину, ориентированном на представление графа в виде списков смежных вершин.

19.43. Опишите, как построить последовательность ориентированных ребер V вершин, для которых не существует ни обратных, ни поперечных ребер и для которых число прямых ребер пропорционально V^2 при стандартном поиске в глубину, ориентированном на представление графа в виде списков смежных вершин.

○ **19.44**. Сформулируйте правила, соответствующие обходу Тремо лабиринта, в котором все коридоры являются односторонними.

• **19.12.** Распространите полученные решения на упражнения 17.56 –17.60 с таким расчетом, чтобы учитывались стрелки на ребрах (в качестве примеров используйте иллюстрации, приводимые в данной главе).

19.3. Достижимость и транзитивное замыкание

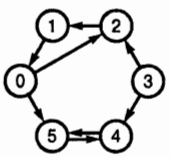
Чтобы получить эффективное решение задачи достижимости в орграфах, начнем с формулировки фундаментального определения.

Определение 19.5. Транзитивное замыкание (transitive closure) *орграфа есть орграф с теми же вершинами, но ребро из s в t в этом транзитивном замыкании возможно в том и только том случае, когда существует ориентированный путь из s в t в заданном орграфе.*

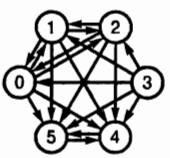
Другими словами, в транзитивном замыкании имеется ребро, исходящее из любой вершины в каждую вершину, достижимую из этой вершине в исходном орграфе. Вполне понятно, что транзитивное замыкание содержит в себе всю информацию, необходимую для решения задачи достижимости. На рис. 19. 13 предлагается к рассмотрению небольшой пример.

Один из привлекательных способов понять транзитивное замыкание основан на представлениях орграфа в виде матрицы смежности и на следующей фундаментальной вычислительной задаче.

Перемножение булевых матриц. *Булевой матрицей (Boolean matrix)* называется матрица, элементы которой принимают двоичные значения, т.е. 0 или 1. Пусть заданы булевы матрицы *A* и *B*. Вычислите произведение *C* двух заданных матриц, используя логические операции *and (и)* и *or (или)*, соответственно, вместо арифметических операций сложения и умножения.



	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1



	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	1	1
2	1	1	1	0	1	1
3	1	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

РИСУНОК 19.13. ТРАНЗИТИВНОЕ ЗАМЫКАНИЕ

Рассматриваемый орграф (вверху) содержит восемь ориентированных ребер, но его транзитивное замыкание (внизу) показывает, что существуют ориентированные пути, соединяющие 19 из 30 пар вершин. Структурные свойства орграфа отражаются в его транзитивном замыкании. Например, строки 0, 1 и 2 матрицы смежности в транзитивном замыкания идентичны (равно как и столбцы 0, 1 и 2), поскольку эти вершины содержатся в ориентированном цикле рассматриваемого орграфа.

Алгоритмы вычисления произведения двух матриц размерностей V x V, включенные в учебники и справочники, вычисляют для каждого s и t скалярное произведение строки s первой матрицы и строки t второй матрицы следующим образом:

```
for (s = 0; s < V; s++)
   for (t = 0; t < V; t++)
      for (i = 0, C[s][t] = 0; i < V; i++)
         C[s][t] = A[s][i]* B[i][t];
```

В матричной системе обозначений мы просто записываем эту операцию как $C = A * B$. Эта операция определена для матриц, состоящих из любых типов элементов, для которых определены операции **0**, **+** и *****. В частности, если **a + b** интерпретируется как логическая операция *or*, а операция **a*b** – как логическая операция *and*, то получается умножение булевых матриц. В языке C++ мы можем воспользоваться следующим вариантом:

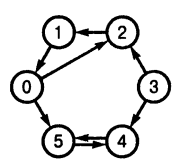
```
for (s = 0; s < V; s++)
   for (t = 0; t < V; t++)
      for (i = 0, C[s][t] = 0; i < V; i++)
         if (A[s][i] && B[i][t]) C[s][t] = 1;
```

Чтобы вычислить элемент **C[s][t]** произведения матриц, мы инициализируем его значением **0**, затем присваиваем ему значение **1**, если находим некоторое значение **i**, для которого как **A[s][i]**, так и **B[i][t]** равны **1**. Выполнение этих вычислений эквивалентно присвоению **C[s][t]** значения **1** в том и только том случае, когда результат побитового выполнения операции *and* над строкой **s** матрицы **A** столбца **t** матрицы **B** имеет ненулевое значение.

Теперь предположим, что **A** – это матрица смежности орграфа A, и мы используем приведенный выше программный код для вычисления $C = A*A \equiv A^2$ (простая замена в программном коде обозначения **B** на **A**). Если рассматривать этот программный код применительно к матрицам смежности, то сразу становится ясно, что он вычисляет: для каждой пары вершин s и t мы помещаем в C ребро, ведущее из s и t, в том и только том случае, когда имеется такая вершина i, для которой в A существует как путь из s и i, так и путь из i и t. Другими словами, ориентированные ребра в A^2 в точности соответствуют ориентированным путям длиной 2 в A. Если при этом мы учтем петли в каждой вершине графа A, то в A^2 появятся ребра графа A, иначе их там не будет. Эта зависимость между умножением булевых матриц и путей в орграфе показана на рис. 19.14. Она немедленно приводит нас к элегантному методу вычисления транзитивного замыкания любого орграфа.

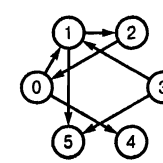
Свойство 19.6. *Транзитивное замыкание орграфа можно вычислить путем построения матрицы смежности A этого графа, добавления петли каждой вершины и вычисления* A^V.

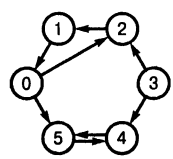
Доказательство: В соответствии с аргументацией предыдущего параграфа, A^3 содержит ребро для каждого пути орграфа длиной меньшего или равного 3, в матрице A^4 содержится каждый путь орграфа, длина которого меньше или равна 4 и т.д. У нас нет необходимости рассматривать пути, длина которых больше V, в силу принципа "картотечного ящика": любой такой путь хотя бы один раз должен повторно пройти через одну из вершин (поскольку в графе всего V вершин), поэтому он не добавляет какую-то новую информацию в транзитивное замыкание, поскольку обе вершины, связанные таким путем, соединены также ориентированным путем, длина которого меньше V (который можно получить за счет удаления цикла из пути в повторно посещенную вершину). ■



	0	1	2	3	4	5
0	0	0	1	0	0	1
1	1	0	0	0	0	0
2	0	1	0	0	0	0
3	0	0	1	0	1	0
4	0	0	0	0	0	1
5	0	0	0	0	1	0

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	0	0	1	0	0	1
2	1	0	0	0	0	0
3	0	1	0	0	0	1
4	0	0	0	0	1	0
5	0	0	0	0	0	1





	0	1	2	3	4	5
0	1	0	1	0	0	1
1	1	1	0	0	0	0
2	0	1	1	0	0	0
3	0	0	1	1	1	0
4	0	0	0	0	1	1
5	0	0	0	0	1	1

	0	1	2	3	4	5
0	1	1	1	0	1	1
1	1	1	1	0	0	1
2	1	1	1	0	0	0
3	0	1	1	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	1	1

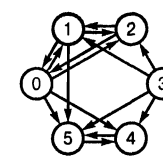


РИСУНОК 19.14. ВОЗВЕДЕНИЕ В КВАДРАТ МАТРИЦЫ СМЕЖНОСТИ

Если мы обнулим главную диагональ матрицы смежности орграфа, квадрат такой матрицы будет представлять собой граф с ребрами, соответствующими каждому пути длиной 2 (вверху). Если установить каждый элемент главной диагонали в 1, то квадрат такой матрицы будет представлять собой граф с ребрами, соответствующий каждому пути длиной 1 или 2 (внизу).

На рис. 19.15 представлены различные степени матрицы смежности для одного и того же, взятого в качестве примера орграфа, постепенно сходящиеся к транзитивному замыканию. Рассматриваемый метод предусматривает V умножений матриц самой на себя, для каждого из которых требуется время, пропорциональное V^3, что в конечном итоге составит V^4. Фактически мы можем вычислить транзитивное замыкание для любого орграфа всего лишь за счет выполнения $\lceil \lg V \rceil$ операций булевого умножения матриц: мы вычисляем A^2, A^4, A^8, ..., до тех пор, пока не достигнем показателя степени, большего или равного V^4. Как следует из доказательства свойства 19.6, $A^t = A^V$ для любого $t > V$; таким образом, результатом этого вычисления, требующего на свое выполнение времени, пропорционального $V^3 \lg V$, будет A^V, т.е. транзитивное замыкание.

И хотя только что описанный подход привлекает своей простотой, тем не менее, существует еще более простой метод. Мы можем вычислить транзитивное замыкание, применив всего лишь одну операцию такого рода, предусматривающую построение транзитивного замыкания матрицы смежности вместо самой матрицы:

```
for (i = 0; i < V; i++)
  for (s = 0; s < V; s++)
    for (t = 0; t < V; t++)
      if (A[s][i] && A[i][t]) A[s][t] = 1;
```

Этот классический метод, предложенный С. Уоршаллом (S. Warshall) в 1962 г., наиболее предпочтителен при вычислении транзитивных замыканий насыщенных орграфов. Приведенный выше программный код подобен коду, который можно использовать для возведения в квадрат булевой матрицы: различие (что очень важно!) заключается в порядке выполнения циклов **for**.

Свойство 19.7. *С помощью алгоритма Уоршалла мы можем вычислить транзитивное замыкание орграфа за время, пропорциональное V^3.*

Доказательство: Оценка времени выполнения рассматриваемого программного кода непосредственно следует из его структуры. Мы докажем, что он способен вычислить транзитивное замыкание методом индукции по **i**. После выполнения первой итерации цикла в матрице на пересечении строки **s** и столбца **t** стоит **1** в том и только том случае, когда существуют пути **s-t** или **s-0-t**. Вторая итерация проверяет все пути между **s** и **t**, которые содержат вершину **1** и, возможно, **0**, такие как, например, **s-1-t**, **s-1-0-t** и **s-0-1-t**. Мы

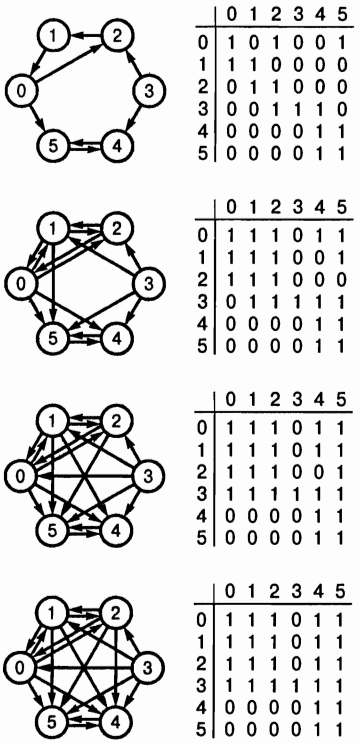


РИСУНОК 19.15. СТЕПЕНИ МАТРИЦЫ СМЕЖНОСТИ И ОРИЕНТИРОВАННЫЕ ПУТИ

В рассматриваемую последовательность включены первая, вторая, третья и четвертая степени (справа, сверху вниз) матрицы смежности, изображенной справа вверху, которая порождает графы с ребрами для каждого из путей длины, меньшей, соответственно, 1, 2, 3 и 4 (слева, сверху вниз) в графе, представленном этой матрицей. Граф в нижней части рисунка представляет собой транзитивное замыкание для этого примера, поскольку в рассматриваем случае не существует путей длиной больше 4, которые соединяют вершины, не соединенные более короткими путями.

приходим к следующей индуктивному предположению: *i*-я итерация цикла устанавливает бит матрицы, находящийся на пересечении строки **s** и столбца **t**, в **1** тогда и только тогда, когда существует ориентированный путь из **s** в **t**, который не содержит никаких вершин с индексами, большими **i** (за исключением, возможно, конечных точек **s** и **t**). Как только что было доказано, это условие выполняется, когда **i** равно **0**. Предположим, что это условие выполняется для *i*-й итерации цикла, тогда путь из **s** в **t**, который не содержит ни одной вершины с индексами, превосходящими **i+1**, существует тогда и только тогда, когда: (*i*) существует путь из **s** в **t**, который не содержит никаких вершин с индексами, меньшими **i**, в этом случае значение **A[s][t]** установлено на предыдущей итерации цикла (в соответствии с индуктивным предположением); либо (*ii*) когда существует путь из **s** в **i+1** и путь из **i+1** в **t**, причем ни один из них не содержит вершин с индексами, большими **i** (за исключением конечных точек), и в этом случае **A[s][i+1]** и **A[i+1][t]** были ранее установлены в **1** (по индуктивному предположению), следовательно, внутренний цикл устанавливает **A[s][t]** в **1**. ■

Мы можем повысить производительность алгоритма Уоршалла путем простого преобразования программного кода: мы вынесем проверку элемента **A[s][i]** из внутреннего цикла, поскольку по мере изменения **t** его значение не меняется. Это действие позволит нам избежать выполнения **t**-го цикла полностью, когда элемент **A[s][i]** равен нулю. Экономия, которую мы получим от данного усовершенствования, зависит от особенностей орграфа и в большинстве случаев весьма существенна (см. упражнения 19.53 и 19.54). Программа 19.3 реализует это усовершенствование и представляет собой метод Уоршалла в виде, позволяющем клиентам сначала выполнить предварительную обработку орграфа (вычислить транзитивное замыкание), затем дать за постоянное время ответ на любой запрос относительно достижимости.

Мы заинтересованы в получении более эффективных решений, в частности, для разреженных графов. Мы хотели бы сократить время предварительной обработки, равно как и пространства памяти, поскольку оба эти фактора предъявляют к системе, использующей алгоритм Уоршалла, такие требования, что стоимость обработки крупных разреженных орграфов по соображениям фактически становится просто запредельной.

В современных приложениях абстрактные типы данных предоставляют нам возможность выделить идею операции из любой конкретной реализации с тем, чтобы можно было сосредоточить свои усилия только на эффективных реализациях. В случае транзитивного замыкания эта точка зрения приводит к признанию того факта, что нам не обязательно вычислять всю матрицу, чтобы предоставить клиентам абстракцию транзитивного замыкания. Вполне вероятно, что одна из возможностей заключается в том, что транзитивным замыканием является крупная разреженная матрица, поэтому возникает необходимость в представлении графа в виде списков смежных вершин, поскольку сохранение графа в матричном виде чересчур накладно. Даже если транзитивное замыкание насыщено, клиентские программы могут проверять только крошечную часть возможных пар вершин, так что вычисление полной матрицы попросту расточительно.

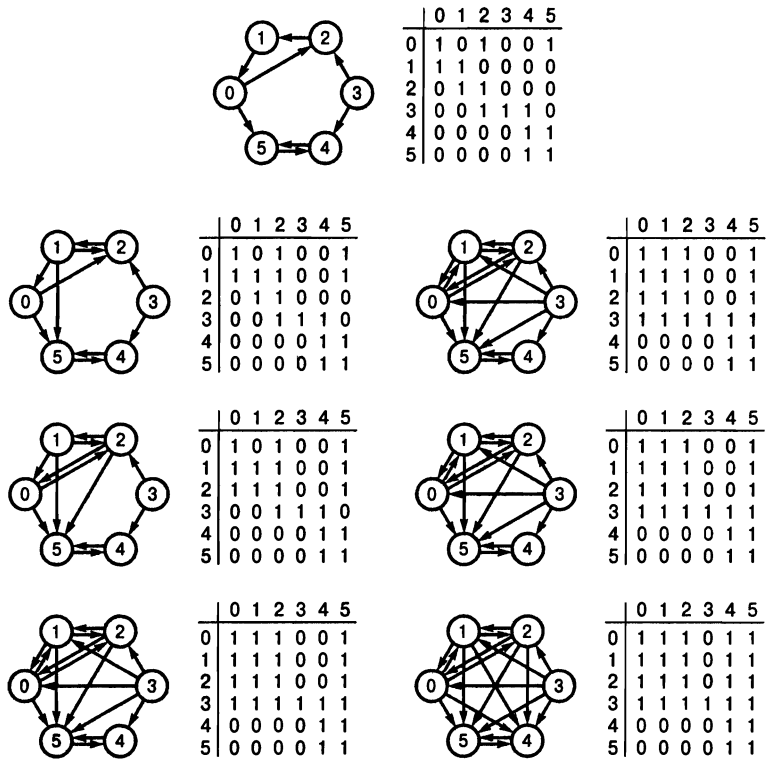


РИСУНОК 19.16. АЛГОРИТМ УОРШАЛЛА

Последовательность графов на данном рисунке показывает, как формируется транзитивное замыкание (внизу) орграфа, выбранного в качестве примера (вверху) путем применения алгоритма Уоршалла. Первая итерация цикла (левая колонка, вверху) добавляет ребра 1-2 и 1-5 в силу существования путей 1-0-2 и 1-0-5, которые содержат вершину 0 (но не содержат вершины с более высокими номерами); вторая итерация цикла (левая колонка, вторая сверху) добавляет ребра 2-0 и 2-5 в силу существования путей 2-1-0 и 2-1-0-5, которые содержат вершину 1 (но ни одной вершины с более высоким номером); третья итерация цикла (левая колонка, внизу) добавляет ребра 0-1, 3-0, 3-1 и 3-5 в силу существования путей 0-2-1, 3-2-1-0, 3-2-1 и 3-2-1-0-5, которые содержат вершину 2 (но ни одной вершины с более высоким номером). В правой колонке показаны ребра, добавленные в результате просмотра вершин 3, 4 и 5. Последняя итерация цикла (правая колонка, внизу) добавляет ребра, ведущие из вершин 0, 1 и 2 в вершину 4, поскольку единственные ориентированные пути из этих вершин в вершину 4, включают 5, т.е. вершину с наивысшим номером.

Программа 19.3 Алгоритм Уоршалла

Конструктор класса **TC** вычисляет транзитивное замыкание графа **G** в приватном элементе данных **T**, так что клиентские программы могут использовать объекты **TC** для проверки, достижима ли заданная вершина орграфа из любой другой вершины. Конструктор инициализирует **T** копией графа **G**, добавляет петли, затем в завершение вычислений использует алгоритм Уоршалла. Класс **tcGraph** должен включать реализацию проверки **edge** на предмет существования ребра.

```
template <class tcGraph, class Graph> class TC
{ tcGraph T;
public:
  TC(const Graph &G) : T(G)
    {
    for (int s = 0; s < T.V(); s++)
      T.insert(Edge(s, s));
    for (int i = 0; i < T.V(); i++)
      for (int s = 0; s < T.V(); s++)
        if (T.edge(s, i))
          for (int t = 0; t < T.V(); t++)
            if (T.edge(i, t))
              T.insert(Edge(s, t));
    }
  bool reachable(int s, int t) const
    { return T.edge(s, t); }
};
```

Мы используем термин *абстрактное транзитивное замыкание (abstract transitive closure)* для обозначения АТД, который предоставляет клиентам возможность проверки после предварительной обработки графа, аналогичной применяемой в программе 19.3. В этом контексте у нас возникает потребность оценки алгоритма не только в аспекте стоимости вычисления транзитивного замыкания (стоимость предварительной обработки), но и в аспектах требуемого пространства памяти и достижимого времени ответа на запросы. То есть, мы предлагаем следующую формулировку свойства 19.7:

Свойство 19.8. *Мы можем поддерживать выполнение проверки заданного орграфа на достижимость (абстрактное транзитивное замыкание) за постоянное время ценой затрат пространства памяти, пропорционального V^2, и времени, пропорционального V^3, затрачиваемого на предварительную обработку.*

Доказательство: Это свойство непосредственно следует из базовых рабочих характеристик алгоритма Уоршалла. ■

Для большинства приложений нашей целью остается не только быстрое вычисление транзитивного замыкания орграфа, но и обеспечение постоянного времени обслуживания запросов относительно абстрактного транзитивного замыкания с гораздо меньшими затратами пространства памяти и намного меньшими затратами времени на предварительную обработку, чем указано в определении 19.8. Можем ли мы найти такую реализацию, которая позволила бы строить клиентские программы, способные выполнять обработку таких графов? Мы вернемся к обсуждению этого вопроса в разделе 19.8.

Существует неформальная внутренняя связь между задачей вычисления транзитивного замыкания орграфа и некоторым числом других фундаментальных вычислительных задач, и эта связь может оказаться полезной для нашего понимания трудностей решения

этой задачи. Мы завершаем этот раздел анализом двух примеров проявления таких проблем.

Прежде всего, мы рассмотрим отношение между транзитивным замыканием и задачей определения *кратчайших путей для всех пар вершин (all-pairs shortest-paths)*. Для орграфов задача заключается в том, чтобы найти для каждой пары вершин ориентированный путь с минимальным числом ребер.

Для заданного орграфа мы инициализируем целочисленную матрицу A размерности V на V, устанавливая элемент **A[s][t]** в **1**, если в орграфе существует ребро, ведущее из **s** в **t**, или, присваивая ему значение сигнальной метки V, если такое ребро не существует. Эту задачу выполняет следующий программный код:

```
for (i = 0; i < V; i++)
   for (s = 0; s < V; s++)
      for (t = 0; t < V; t++)
         if (A[s][i] + A[i][t] < A[s][t])
             A[s][t] = A[s][i] + A[i][t];
```

Этот программный код отличается от алгоритма Уоршалла, который мы рассматривали непосредственно перед тем, как дать формулировку свойства 19.7, только оператором **if** во внутреннем цикле. В самом деле, в соответствующей абстрактной форме эти вычисления ничем не отличаются (см. упражнения 19.55 и 19.56). Преобразование доказательства свойства 19.7 в прямое доказательство того, что этот метод достигает желаемой цели, достаточно просто. Этот метод является частным случаем *алгоритма Флойда (Floid's algorithm)* поиска кратчайших путей во взвешенных графах (см. главу 21). Решение для ориентированных графов, построенное на использовании поиска в ширину, которое мы рассматривали в разделе 18.7, также способно отыскивать кратчайшие пути в орграфах (модифицированных соответствующим образом). Кратчайшие пути подробно рассматриваются в главе 21, поэтому мы отложим детальное сравнение рабочих характеристик обоих алгоритмов до этой главы.

Во-вторых, как мы уже могли убедиться, задача транзитивного замыкания также тесно связана и с задачей перемножения булевых матриц. Базовые алгоритмы решения обоих задач, которые мы рассматривали выше, требуют для своего выполнения с использованием аналогичной схемы вычислений время, пропорциональное V^3. Известно, что умножение булевых матриц является сложной вычислительной задачей: известны алгоритмы, обладающие большим быстродействием, чем простые методы, однако вопрос о том, достаточны ли выгоды, получаемые от их использования, чтобы оправдать усилия, затрачиваемые на их реализацию, остается открытым. Этот факт имеет большое значение в данном контексте, поскольку мы можем воспользоваться быстрым алгоритмом умножения булевых матриц для разработки быстрого алгоритма транзитивного замыкания (он медленнее алгоритма умножения всего лишь в $\lg V$ раз), применяя для этой цели метод многократного возведения в квадрат, представленный на рис. 19.15. И наоборот, мы можем понизить уровень сложности вычисления транзитивного замыкания.

Свойство 19.9. *Мы можем использовать алгоритм транзитивного замыкания для вычисления произведения двух булевых матриц, при этом различие во времени исполнения не превышает некоторого постоянного коэффициента.*

Доказательство: Пусть даны булевы матрицы A и B размерности V на V, построим матрицу размерности $3V$ на $3V$ следующего вида:

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

Здесь 0 означает матрицу размерности V на V, все элементы которой равны **0**, I означает тождественную матрицу размерностью V на V, все элементы которой равны **0**, за исключением элементов, стоящих на главной диагонали, которые равны **1**. Теперь рассмотрим, как эту матрицу можно использовать в качестве матрицы смежности орграфа и вычислить его транзитивное замыкание посредством многократного возведения в квадрат. Для этого потребуется выполнить всего лишь одно действие:

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^2 = \begin{pmatrix} I & A & A*B \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

Матрица в правой части этого уравнения есть транзитивное замыкание, поскольку последующие умножения дают эту же матрицу. В то же время в этой матрице содержится элемент $A*B$ в ее верхнем правом углу. Какой бы алгоритм мы ни использовали для решения задачи получения транзитивного замыкания, мы можем применить его для решения задачи перемножения булевых матриц на том же уровне затрат (т.е. разница в затратах определяется некоторым постоянным множителем). ■

Важность значения этого свойства определяется убежденностью экспертов в том, что задача умножения булевых матриц относится к числу сложных: математики десятилетиями работают над тем, чтобы исследовать, насколько она трудна, и решение этой проблемы еще не найдено; наиболее известные результаты говорят в пользу того, что время выполнения умножения примерно пропорционально $V^{2.5}$ (см. раздел ссылок). Теперь, если мы сможем найти решение задачи транзитивного замыкания с линейной зависимостью по времени (пропорциональное V^2), то мы получим также и решение задачи перемножения булевых матриц с линейной временной зависимостью. Подобная зависимость между задачами известна как *приведение (reduction)*; мы говорим, что задача перемножения булевых матриц *сводится (reduces)* к задаче транзитивного замыкания (см. раздел 21.6 и часть 8). В самом деле, приведенное выше доказательство показывает, что умножения булевых матриц сводится к нахождению в орграфе путей длиной 2.

Несмотря на проведение интенсивных исследований с участием многих математиков, ни один из них не смог предложить алгоритм умножения булевых матриц с линейной временной зависимостью, следовательно, и мы не сможем предложить простого алгоритма транзитивного замыкания с линейной временной зависимостью. С другой стороны, никому еще не удалось доказать, что такие алгоритмы не существуют, так что возможность появления такого алгоритма не исключается. Короче говоря, мы можем понимать свойство 19.9 в том смысле, что, отвергая возможность прорыва в исследованиях, мы не вправе ожидать, что в худшем случае время выполнения любого алгоритма транзитивного замыкания, какой только мы можем придумать, будет пропорционально V^2. Однако вопреки этому заключению, мы можем разработать быстрые алгоритмы для некоторых

специальных классов орграфов. Например, мы уже упоминали простой метод вычисления транзитивного замыкания, быстродействие которого для разреженных графов намного выше, чем у алгоритма Уоршалла.

Свойство 19.10. *С помощью DFS мы можем поддерживать постоянное время ответа на запросы относительно абстрактного транзитивного замыкания орграфа, затрачивая пространство памяти, пропорциональное V^2, и время, пропорциональное $V(E+V)$, на предварительную обработку (вычисление транзитивного замыкания).*

Доказательство: Как отмечалось в предыдущем разделе, поиск в глубину позволяет найти все вершины, достижимые из исходной, за время, пропорциональное E, если мы используем представление графа в виде списков смежных вершин (свойство 19.5 и рис. 19.11). В силу этого обстоятельства, если мы выполняем поиск в глубину V раз, по одному разу на каждую вершину, используя ее как исходную, то мы можем вычислить набор вершин, достижимых из каждой вершины, т.е. транзитивное замыкание, за время, пропорциональное $V(E + V)$. Этот же аргумент справедлив для каждого обобщенного поиска (см. раздел 18.8 и упражнение 19.66). ■

Программа 19.4. Транзитивное замыкание, построенное на основе поиска в глубину

Класс DFS (поиска в глубину) реализует тот же интерфейс, что и программа 19.3. Она вычисляет транзитивное замыкание **T**, запуская DFS в каждой вершине графа **G**, для вычисления набора узлов, достижимых из этой вершины. Каждое обращение к рекурсивной функции добавляет ребро, исходящее из начальной вершины, и генерирует рекурсивные вызовы с целью заполнения соответствующей строки в матрице транзитивного замыкания. Эта матрица используется также для маркирования посещенных вершин в процессе выполнения поиска в глубину, так что он требует, чтобы класс **Graph** поддерживал проверку **edge** на предмет существования ребра.

```
template <class Graph> class tc
{ Graph T; const Graph &G;
   void tcR(int v, int w)
   {
     T.insert(Edge(v, w));
     typename Graph::adjIterator A(G, w);
     for (int t = A.beg(); !A.end(); t = A.nxt())
       if (!T.edge(v, t)) tcR(v, t);
   }
public:
   tc(const Graph &G) : G(G), T(G.V(), true)
     { for (int v = 0; v < G.V(); v++) tcR(v, v); }
   bool reachable(int v, int w)
     { return T.edge(v, w); }
};
```

Программа 19.4 предлагает реализацию алгоритма транзитивного замыкания, построенного на базе поиска. Этот класс реализует тот же интерфейс, что и программа 19.3. Результат прогона этой программы на орграфе, показанном на рис. 19.1, представлен на первом дереве каждого леса на рис. 19.11.

В случае разреженных орграфов этот подход, основанный на поиске, обладает определенными достоинствами. Например, если число ребер E пропорционально числу вершин V, то программа 19.4 вычисляет транзитивное замыкание за время, пропорциональное V^2. Как она это может сделать с учетом возможности приведения к умножению

булевых матриц? Ответ на данный вопрос может быть таким: рассматриваемый алгоритм транзитивного замыкания является оптимальным способом умножения *некоторых типов* булевых матриц (число ненулевых элементов которых составляет $O(V)$). Нижняя граница показывает, что мы не можем надеяться на то, что найдем такой алгоритм транзитивного замыкания, который выполняется за время, пропорциональное V^2 для всех орграфов, однако это не исключает возможности, что мы найдем алгоритмы, подобные рассматриваемому, которые работают быстрее *на некоторых классах* орграфов. Если эти графы суть именно те, которые и требуется обрабатывать, зависимость между транзитивным замыканием и умножением булевых матриц не будет иметь для нас особого значения.

Методы, описанные в данном разделе, легко расширить с таким расчетом, чтобы они снабдили клиентские программы средством, позволяющим находить конкретные пути, соединяющие две вершины, путем отслеживания дерева поиска в соответствии с изложенным в разделе 17.8. Мы рассмотрим специальные реализации АТД этого вида в контексте более общих задач поиска кратчайших путей в главе 21.

В таблице 19.1 представлены эмпирические результаты сравнения элементарных алгоритмов транзитивного замыкания, описанных в настоящем разделе. Реализация решения, основанного на поиске и ориентированного на представление графа в виде списков смежных вершин, на данный момент является наиболее быстрым методом для разреженных графов. Все остальные реализации вычисляют матрицу смежности (размера V^2), следовательно, ни одна из них не подходит для обработки крупных разреженных графов.

В случае разреженных графов, транзитивные замыкания которых тоже разрежены, мы можем воспользоваться для вычисления замыканий реализацией, ориентированной на представление графа в виде списка смежных вершин, так что размер ее выхода пропорционален числу ребер в транзитивном замыкании. Естественно, это число служит нижней границей стоимости вычисления транзитивного замыкания, которое мы можем получить для различных типов орграфов, используя для этой цели различные алгоритмические технологии (см. упражнения 19.64 и 19.65). Несмотря на наличие такой возможности, мы в общем случае полагаем, что результат транзитивного замыкания есть насыщенная матрица. В силу этого обстоятельства, мы можем воспользоваться такой реализацией, как **DenseGRAPH**, которая способна отвечать на запросы о достижимости, при этом мы рассматриваем алгоритмы транзитивного замыкания, вычисляющие матрицу транзитивного замыкания за время, пропорциональное V^2, как оптимальные, поскольку на их выполнение затрачивается время, пропорциональное размерам их выходов.

Таблица 19.1. Эмпирическое исследование алгоритмов транзитивного замыкания

В данной таблице показаны значения времени выполнения различных алгоритмов вычисления транзитивных замыканий случайных орграфов, как насыщенных, так и разреженных. Эти значения разительно отличаются друг от друга. Для всех алгоритмов, за исключением поиска в глубину на орграфе, представленном в виде списков смежных вершин, время выполнения возрастает в 8 раз при увеличении V в два раза. Этот факт подтверждает вывод о том, что по существу это время пропорционально V^3. Поиск в глубину на орграфе, представленном в виде списков смежных вершин, требует для своего выполнения время, пропорциональное VE, что может служить объяснением того факта, что время выполнения этого алгоритма возрастает в примерно в 4 раза с увеличением V и E в два раза каждого (разреженные графы), и примерно в 2 раза, когда мы увеличиваем E в два раза (насыщенные графы) за исключением тех случаев, когда непроизводительные

затраты, связанные с обходом списков, снижают производительность сильно насыщенных графов.

Разреженные (10*V* ребер)					Насыщенные (250 вершин)				
V	W	W*	A	L	*E*	W	W*	A	L
25	0	0	1	0	5000	289	203	177	23
50	3	1	2	1	10000	300	214	184	38
125	35	24	23	4	25000	309	226	200	97
250	275	181	178	13	50000	315	232	218	337
500	2222	1438	1481	54	100000	326	246	235	784

Обозначения:

W Алгоритм Уоршалла (раздел 19.3)

W* Усовершенствованный алгоритм Уоршалла (программа 19.3)

A Поиск в глубину, представление графа в виде матрицы смежности (программы 19.4 и 17.7)

L Поиск в глубину, представление графа в виде списков смежных вершин (программы 19.4 и 17.9)

Если такая матрица смежности симметрична, она эквивалентна неориентированному графу, благодаря чему поиск транзитивного замыкания становится эквивалентным поиску связных компонентов — транзитивное замыкание представляет собой объединение полных графов в вершинах связных компонент (см. упражнение 19.48). Алгоритмы определения связности, которые мы изучали в разделе 18.5, эквивалентны вычислению абстрактного транзитивного замыкания для симметричных орграфов (неориентированных графов), требуют для своего выполнения время, пропорциональное V, и способны давать ответы на запросы о достижимости за постоянное время. Можем ли мы добиваться таких же успешных результатов в случае орграфов общего вида? Для каких типов орграфов можно вычислить транзитивное замыкание за линейное время? Чтобы ответить на эти вопросы нам придется изучить структуру орграфов более подробно, и в первую очередь — структуру графов DAG.

Упражнения

▷ **19.46**. Что собой представляет транзитивное замыкание орграфа, который состоит только из направленного цикла с V вершинами?

19.47. Сколько ребер содержит транзитивное замыкание орграфа, который состоит только из простого ориентированного пути с V вершинами?

▷ **19.48**. Постройте транзитивное замыкание неориентированного графа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

● **19.49**. Покажите, как можно построить орграф с V вершинами и E ребрами, обладающий тем свойством, что число ребер в транзитивном замыкании пропорционально t, для любого t, принимающего значения в диапазоне между E и V^2. Как обычно, предполагаем, что $E > V$.

19.50. Выведите формулу числа ребер в транзитивном замыкании орграфа, представляющего собой ориентированный лес, как функцию структурных свойств леса.

19.51. Представьте в стиле рис. 19.15 процесс вычисления транзитивного замыкания орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

путем многократного возведения в квадрат.

19.52. Представьте в стиле рис. 19.16 процесс вычисления транзитивного замыкания орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4

с применением алгоритма Уоршалла.

○ **19.53**. Найдите семейство разреженных орграфов, для которых прогон усовершенствованной версии алгоритма Уоршалла для вычисления транзитивного замыкания (программа 19.3) производится за время, пропорциональное VN.

○ **19.54**. Найдите разреженный орграф, для которого прогон усовершенствованной версии алгоритма Уоршалла для вычисления транзитивного замыкания (программа 19.3) производится за время, пропорциональное V^3.

○ **19.55**. Разработайте базовый класс, из которого вы можете получить производные классы, которые реализуют как алгоритм Уоршалла, так и алгоритм Флойда. (Это упражнение представляет собой версию упражнения 19.56 для тех, кто лучше знаком с абстрактными типами данных, чем с абстрактной алгеброй).

● **19.56**. Воспользуйтесь аппаратом абстрактной алгебры для разработки родового алгоритма, который заключает в себе как алгоритм Уоршалла, так и алгоритм Флойда. (Данное упражнение есть версия упражнения 19.55 для тех, кто лучше ознакомлен с абстрактной алгеброй, чем с абстрактными типами данных).

○ **19.57**. Представьте в стиле рис. 19.16 разработку матрицы всех кратчайших путей для примера графа, изображенного на этом рисунке, для чего воспользуйтесь алгоритмом Флойда.

19.58. Является ли произведение двух симметричных булевых матриц симметричным? Сопроводите ваш ответ доказательством.

19.59. Введите в программы 19.3 и 19.4 общедоступные функции-элементы, позволяющие клиентским программам использовать объекты **tc** для определения числа ребер в транзитивном замыкании.

19.60. Разработайте способ подсчета числа ребер в транзитивном замыкании и его модификации при добавлении и удалении ребер. Покажите, какие затраты влекут за собой добавления и удаления ребер в соответствии с вашей схемой.

▷ **19.61**. Введите в программы 19.3 и 19.4 общедоступную функцию-элемент, которая возвращает вектор, индексированный именами вершин, показывающий, какие вершины достижимы из данной вершины.

○ **19.62**. Проведите эмпирические исследования с целью определить число ребер в транзитивном замыкании различных типов орграфов (см. упражнения 19.11–19.18).

● **19.63**. Выполните исследование представления графа в виде битовой матрицы, описание которой дано в упражнении 17.23. Какой из методов вы можете ускорить в B

раз (где B есть число бит в слове вашего компьютера): алгоритм Уоршалла или алгоритм, построенный на базе поиска в глубину? Подтвердите ответ разработкой соответствующей программной реализации.

○ **19.64**. Дайте пример программы, вычисляющей транзитивное замыкание орграфа, который представляет собой ориентированный лес, за время, пропорциональное числу ребер в транзитивном замыкании.

○ **19.65**. Реализуйте алгоритм абстрактного транзитивного замыкания для разреженных графов, который используют пространство памяти, пропорциональное T, и может отвечать на запросы о достижимости за постоянное время после предварительной обработки, на которую затрачивается время, пропорциональное $VE + T$, где T – число ребер в транзитивном замыкании. *Указание*: Воспользуйтесь динамическим хэшированием.

▷ **19.66**. Представьте версию программы 19.4, которая основана на обобщенном поиске на графе (см. раздел 18.8), и проведите эмпирические исследования с тем, чтобы выяснить, оказывает ли какое-либо влияние выбор алгоритма поиска на графе на ее производительность.

19.4. Отношения эквивалентности и частичные порядки

В данном разделе обсуждаются фундаментальные понятия теории множеств и их отношение к алгоритмам абстрактного транзитивного замыкания. Цель этого обсуждения связана с тем, чтобы внушить читателям мысль о том, что исследования проводятся в широком контексте, и чтобы показать всю широту применения алгоритмов, которые мы изучаем. Читатели с математическим складом ума, знакомые с теорией множеств, могут сразу перейти к изучению раздела 19.5, поскольку материал, который мы здесь рассматриваем, элементарен (хотя предлагаемы нами краткий обзор терминологии может оказаться весьма полезным); в то же время у читателей, не знакомых с теорией множеств, может возникнуть желание ознакомиться с элементарными понятиям дискретной математики, так как эти сведения представлены в весьма лаконичной форме. Связь между орграфами и фундаментальными математическими понятиями достаточно важна, чтобы ее игнорировать.

Пусть задано некоторое множество. По определению, *отношение (relation)*, в котором находятся его объекты, есть множество упорядоченных пар этих объектов. Если отвлечься от некоторых деталей, таких как параллельные ребра и петли, это определение аналогично определению орграфа: отношения и орграфы суть различные представления одной и той же абстракции. Это математическое понятие обладает большей универсальностью, поскольку множества могут быть бесконечными, в то время как все современные компьютерные программы работают с конечными множествами, но на данный момент мы не принимаем во внимание эти различия.

Обычно мы выбираем символ R и используем выражение sRt для обозначения утверждения "упорядоченная пара (s,t) находится в отношении R". Например, мы используем символ "<" для представления отношения "меньше чем". Пользуясь этой терминологией, мы можем описывать различные свойства отношений. Например, отношение R называется *симметричным*, если из sRt следует tRs для всех s и t; говорят, что отношение *рефлек-*

сивно, если *sRs* справедливо для всех *s*. Симметричные отношения соответствуют неориентированным графам. Рефлексивные отношения соответствуют графам, в которых в каждой вершине есть петли; отношения, соответствующие графам, в котором ни у одной из вершин нет петель, называются *нерефлексивными*.

Говорят, что отношение *транзитивно*, когда из *sRt* и *tRu* следует *sRu* для всех *s*, *t* и *u*. *Транзитивное замыкание (transitive closure)* того или иного отношения представляет собой четко определенное понятие; однако вместо того, чтобы давать его определение в контексте теории множеств, мы обратимся к определению орграфов, данному в разделе 19.3. Любое отношение эквивалентно некоторому орграфу, а транзитивное замыкание этого отношения эквивалентно транзитивному замыканию орграфа. Транзитивное замыкание любого отношения само транзитивно.

В контексте алгоритмов на графе особый интерес вызывает у нас два специальных транзитивных отношения, которые определяются дальнейшими ограничениями. Эти два типа отношений, получивших широкое применение, известны как *отношения эквивалентности (equivalence relations)* и *частичные порядки (partial orders)*.

Отношение эквивалентности ($\equiv$) есть транзитивное отношение, которое к тому же рефлексивно и симметрично. Обратите внимание на тот факт, что симметричное и транзитивное отношение, которое помещает каждый объект в некоторую упорядоченную пару, должно быть отношением эквивалентности: если $s \equiv t$, то $t \equiv s$ (по симметричности) и $s \equiv s$ (по транзитивности). Отношение эквивалентности разносит объекты множества по подмножествам, известным как *классы эквивалентности (equivalence class)*. Два объекта *s* и *t* содержатся в одном и том же классе эквивалентности тогда и только тогда, когда $s \equiv t$. Типичными примерами эквивалентности могут служить следующие отношения:

Арифметика над абсолютными значениями чисел. Любое положительное целое *k* определяет на множестве целых чисел отношение эквивалентности, т.е. $s \equiv t \pmod{k}$ тогда и только тогда, когда остаток от деления *s* на *k* равен остатку от деления *t* на *k*. Очевидно, что это отношение симметрично, несложное доказательство показывает, что оно еще и транзитивно (см. упражнение 19.67) и в силу этого обстоятельства это отношение есть отношение эквивалентности.

Связность в графах. Отношение "содержится в той же связной компоненте, что и...", связывающее вершины, есть отношение эквивалентности, ибо оно симметрично и транзитивно. Классы эквивалентности соответствуют связным компонентам в графах.

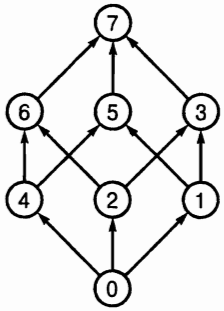
Когда мы выполняем построение АТД графа, который предоставляет клиентам возможность проверки, находятся ли две какие-либо вершины в одной и той же связной компоненте, мы реализуем АТЖ отношения эквивалентности, который предоставляет клиентам возможность проверки эквивалентности двух объектов. На практике это соответствие имеет большое значение, поскольку граф есть сжатая форма представления отношения эквивалентности (см. упражнение 19.71). Фактически, как мы видели в главах 1 и 18, чтобы построить такой АТД, мы должны поддерживать только один вектор, индексированный именами вершин.

Частичный порядок (partial order) $\prec$ есть транзитивное нерефлексивное отношение. Нетрудно доказать, что из нерефлексивности и транзитивности вытекает тот факт, что частичные порядки *асимметричны*. Если $s \prec t$ и $t \prec s$, то $s \prec s$ (по транзитивности), что противоречит нерефлексивности, т.е. $s \prec t$ и $t \prec s$ одновременно выполняться не могут.

Более того, используя ту же аргументацию, легко показать, что в условиях частичного порядка циклы, такие как $s \prec t$, $t \prec u$ и $u \prec s$, невозможны. Следующие примеры представляют собой типичные частичные порядки:

Включение подмножеств. Отношение "включает, но не равно" ($\subset$), определенное на множестве подмножеств данного множества есть частичный порядок – естественно, он нерефлексивен, и если $s \subset t$ и $t \subset u$, то $s \subset u$.

Пути в графах DAG. Отношение "достижим посредством непустого пути из..." есть частичный порядок на множестве вершин графа DAG без петель, поскольку оно транзитивно и нерефлексивно. Подобно отношению эквивалентности и неориентированным графам, этот конкретный частичный порядок важен для многих приложений, поскольку DAG представляет собой неявное представление частичного порядка в сжатой форме. Например, рис. 19.17 служит иллюстрацией графа DAG частичных порядков включения подмножеств, количество ребер которых составляет лишь небольшая часть мощности частичного порядка (см. упражнение 19.73).



0	000	∅
1	001	{*c*}
2	010	{*b*}
3	011	{*b*, *c*}
4	100	{*a*}
5	101	{*a*, *c*}
6	110	{*a*, *b*}
7	111	{*a*, *b*, *c*}

РИСУНОК 19.17. ГРАФ DAG ВКЛЮЧЕНИЯ МНОЖЕСТВ

В графе DAG, показанном в верхней части рисунка, вершины представляют некоторое подмножество множества, состоящего из трех элементов в соответствии с таблицей, приведенной в нижней части рисунка. Транзитивное замыкание этого графа представляет собой порядок включения подмножеств: существует ориентированный путь между двумя узлами в том и только том случае, когда подмножество, представляемое первым узлом, содержится в подмножестве, представленном вторым узлом.

В самом деле, мы редко определяем частичные порядки посредством перечисления всех упорядоченных пар, поскольку таких пар очень много. Вместо этого мы в общем случае определяем нерефлексивное отношение (граф DAG) и рассматриваем его транзитивное замыкание. Такое его использование является для нас основной причиной изучения АТД, реализующих абстрактное транзитивное замыкание графов DAG. Работая с графами DAG, мы рассмотрим примеры частичных порядков в разделе 19.5.

Полный порядок (total order) T есть частичный порядок, при котором выполняется либо sTt, либо tTs, $s \neq t$. Знакомыми нам примерами полного порядка являются отношение "меньше чем" на множествах целых или вещественных чисел или лексикографический порядок на множестве строк символов. Наши исследования алгоритмов поиска и сортировки в частях 3 и 4 основывались на реализации полностью упорядоченных АТД на множествах. В полностью упорядоченном множестве существует один и только один способ упорядочить элементы этого множества так, чтобы выполнялось отношение sTt всякий раз, когда s предшествует t; в то же время, в условиях частичного порядка существует много способов такого упорядочения. В разделе 19.5 мы проведем исследование алгоритмов, решающих эту задачу.

В конечном итоге, приводимые ниже соответствия между множествами и моделями графов помогают нам получить представление, насколько важны фундаментальные алгоритмы на графах и как широко они применяются на практике:

- Отношения и орграфы.
- Симметричные отношения и неориентированные графы.
- Транзитивные отношения и пути в графах.
- Отношения эквивалентности и пути в неориентированных графах.
- Частичные порядки и пути в графах DAG.

Из изложенного выше следует, что все типы графов и алгоритмов, которые мы изучаем, имеют хорошую перспективу и снабжают нас дополнительной мотивацией для изучения базовых свойств графов DAG и алгоритмов их обработки.

Упражнения

19.67. Покажите, что отношение "имеет тот же остаток после деления на k" транзитивно (и в силу этого обстоятельства является отношением эквивалентности) на множестве целых чисел.

19.68. Покажите, что отношение "та же реберно-связанная компонента, что и..." есть отношение эквивалентности на множестве вершин любого графа.

19.69. Покажите, что отношение "та же двусвязная компонента, что и..." не является отношением эквивалентности на множестве вершин всех графов.

19.70. Докажите, что транзитивное замыкание отношения эквивалентности есть отношение эквивалентности, и что транзитивное замыкание частичного порядка есть частичный порядок.

▷ **19.71**. Мощность отношения — это число упорядоченных пар. Покажите, что мощность отношения эквивалентности равна сумме квадратов мощностей классов эквивалентности этого отношения.

○ **19.72**. Используя интерактивный словарь, постройте граф, который представляет отношение эквивалентности "имеет k общих букв с..." на множестве слов. Определите число классов эквивалентности для $k = 1, 2, 3, 4, 5$.

19.73. Мощность отношения частичного порядка есть число упорядоченных пар. Какова мощность частичного порядка включения подмножеств на множестве, состоящем из n элементов?

▷ **19.74**. Покажите, что частичный порядок "является сомножителем" есть частичный порядок на множестве целых чисел.

19.5 Графы DAG

В этом разделе мы рассмотрим различные приложения графов DAG (directed acyclic graph — ориентированный ациклический граф). На то имеются две причины. Во-первых, они служат неявными моделями частичных порядков; во многих приложениях мы имеем дело непосредственно с графами DAG, и в силу этого обстоятельства нуждаемся в эффективных алгоритмах обработки таких графов. Во-вторых, различные приложения дают нам возможность постижения природы графов DAG, а понимание графов DAG существенно упрощает понимание общих свойств орграфов.

Поскольку графы DAG являются особым видом орграфов, все задачи обработки графов DAG элементарно сводятся к задачам обработки орграфов. Хотя мы и надеемся, что

обработка графов DAG проще, чем обработка орграфа общего вида, тем не менее, мы знаем, что если мы столкнемся с задачей, которая трудно решается на графе DAG, то решение этой же задачи на орграфах общего вида едва ли окажется проще. Как мы сможем убедиться далее, задача вычисления транзитивного замыкания попадает именно в эту категорию. И наоборот, осознание сложности обработки графов DAG важно с той точки зрения, что каждый орграф содержит некоторый DAG в качестве своего ядра (см. свойство 19.2), так что мы сталкиваемся с графами DAG даже в тех случаях, когда работаем с орграфами, не являющимися графами DAG.

Приложение-прототип, в котором графы DAG возникают сами по себе, называется приложением *составления расписаний (scheduling)*. В общем случае решение задачи составления расписаний заключается в организации завершения решения некоторого множества *задач (tasks)* в условиях действия некоторого набора *ограничений (constraints)* путем задания условий, когда и как эти задачи должны выполняться. Ограничениями могут быть некоторые функции от времени выполнения или от других ресурсов, потребляемых задачей. Наиболее важным типом ограничений является *ограничения предшествования (precedence constraints)*, которые определяют, что одни задачи должны быть обязательно решены до того, как может быть начато решение других задач, благодаря чему на множество задач устанавливается некоторый частичный порядок. Различные виды дополнительных ограничений приводят к возникновению различных типов задач составления расписаний разной степени сложности. Изучению подверглись буквально тысячи различных задач, при этом исследователи продолжают поиск более совершенных алгоритмов решения для многих из них. Возможно, простейшую нетривиальную задачу составления расписания можно сформулировать следующим образом:

Составление расписаний. Пусть дано множество задач, требующих решения, на котором задан частичный порядок, определяющий, что решение некоторых задач должны быть завершено, прежде чем начнется выполнение некоторых других задач. Каким образом можно организовать решение всех задач множества так, чтобы в конечном итоге все они были успешно завершены с соблюдением частичного порядка?

В своем основном виде задача составления расписания называется *топологической сортировкой (topological sorting)*; при ее решении особых трудностей не возникает, как мы сможем убедиться в следующем разделе, в котором предлагаются два алгоритма ее решения. В более сложных практических приложениях нам, возможно, придется накладывать дополнительные ограничения на то, как следует составлять расписание решения задач, и сложность топологической сортировки, естественно, возрастает. Например, задачи могут соответствовать курсам лекций в расписании занятий студентов, а частичный порядок может определяться некоторыми предварительными условиями. Топологическая сортировка дает реальное расписание курса, удовлетворяющее предварительным условиям, но, по-видимому, не такое, какое учитывало бы другие виды ограничений, которые неплохо было бы добавить в модель, такие как конфликт курсов, ограничения на прием студентов и т.д. Еще один пример: задачи могут быть частью некоторого производственного процесса, при этом частичный порядок представляет собой требования к последовательности, в которой выполняются конкретные процессы. Топологическая сортировка предлагает нам способ планирования задач, но, возможно, существует другой такой способ, требующий меньше времени и денег или других ресурсов, не учтенных в модели. В гла-

вах 21 и 22 мы рассмотрим различные версии задачи составления расписаний, которые учитывают более общие ситуации, подобные описанным выше.

Часто первая наша задача заключается в том, чтобы определить, содержит ли заданный граф DAG направленный цикл. Как было показано в 19.2, мы легко можем реализовать класс, который позволяет клиентским программам проверять, является ли орграф общего типа графом DAG, за линейное время, путем проведения стандартного поиска в глубину и проверки, содержит ли полученный при этом лес DFS обратные ребра (см. упражнение 19.75). Для реализации алгоритмов, ориентированных на работу с графами DAG, мы строим специальные клиентские классы нашего стандартного АТД **GRAPH**, который предполагает, что они производят обработку орграфов без циклов, возлагая обязанность по проверке на присутствие циклов на клиентские программы. Подобная организация допускает возможность того, что алгоритм обработки графа DAG воспроизводит полезные результаты даже в тех случаях, когда выполняется на орграфах с циклами, в чем время от времени возникает потребность. В разделах 19.6 и 19.7 анализируются реализации классов топологической сортировки (**DAGts**) и классов определения достижимости в графах DAG (**DAGts** и **DAGreach**); программа 19.13 представляет собой пример клиента такого класса.

В некотором смысле, с одной стороны DAG – это дерево, с другой – это граф. Естественно, мы воспользуемся преимуществами такой структуры графов DAG во время их обработки. Например, если захотим, мы можем рассматривать DAG как дерево. Предположим, мы хотим совершить обход ориентированного ациклического графа (DAG) **D**, как если бы он был деревом с корнем в вершине **w**, так что, например, результат обхода двух графов DAG на рис. 19.18 с помощью этой программы будет одним и тем же. Следующая простая программа выполняет эту задачу так же, как бы это сделал рекурсивный обход дерева:

```
void traverseR(Dag D, int v)
   {
      visit(v);
         typename Dag::adjIterator A(D, v);
            for (int t = A.beg(); !A.end(); t = A.nxt())
               traverseR(D, t);
   }
```

однако мы редко используем полный обход этого типа, поскольку обычно стремимся воспользоваться теми же средствами, какие применяются в графах DAG для экономии пространства, чтобы сэкономить время при его обходе (например, маркирование посещенных вершин при обычном поиске в глубину). Та же идея применяется к *поиску*, по условиям которого мы выполняем рекурсивные вызовы только для одной связи, инцидентной каждой вершине. В таком алгоритме затраты на поиск в графе DAG и дереве оказываются одними и теми же, но в то же время DAG использует намного меньшее пространство памяти.

Поскольку графы DAG обеспечивают способы более компактного представления деревьев, чем идентичные им поддеревья, мы часто пользуемся графами DAG вместо деревьев для представления различных вычислительных абстракций. В контексте конструкции алгоритма, различия между представлением программы в виде DAG и представлением в виде дерева существенны в плане динамического программирования (см., например, рис.

19.18 и упражнение 19.78). Графы DAG к тому же широко используются в компиляторах в качестве промежуточных представлений арифметических выражений и программ (см., например, рис. 19.19) и в системах расчета электронных схем в качестве промежуточных представлений комбинационных схем.

В соответствии с вышесказанным, при рассмотрении бинарных деревьев возникает важный пример, который находит применение во многих приложениях. Мы можем наложить те же ограничения на графы DAG, какие мы налагали на деревья, чтобы выделить из них бинарные деревья.

Определение 19.6. Двоичный DAG *(binary DAG) есть ориентированный ациклический граф с двумя ребрами, исходящими из каждого узла, которые рассматриваются как левое и правое ребро, при этом каждое из них или оба сразу могут быть нулевыми.*

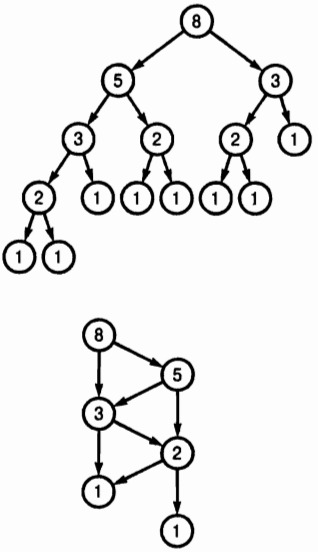


РИСУНОК 19.18. МОДЕЛЬ DAG ВЫЧИСЛЕНИЙ ЧИСЕЛ ФИБОНАЧЧИ

Дерево в верхней части диаграммы указывает на зависимость вычисления очередного числа от вычисления двух его предшественников. Граф DAG, изображенный в нижней части диаграммы, демонстрирует ту же зависимость, используя лишь часть узлов.

Различие между двоичными графами DAG и бинарными деревьями заключается в том, что в двоичном DAG может быть более одной связи, ведущей в конкретный узел. Как и данное нами определение бинарных деревьев, это определение моделирует естественное представление, в рамка которого каждый узел представляет собой структуру с левой связью и с правой связью, указывающими на другие узлы (либо являющимися нулевыми), подчиняются только глобальному ограничению, суть которого заключается в том, что никакие направленные циклы невозможны. Двоичные графы DAG имеют большое значение, поскольку они обеспечивают компактный способ представления бинарных деревьев в некоторых приложениях. Например, как нетрудно видеть из рис. 19.20 и программы 19.5, мы можем сжать trie-дерево существования до двоичного DAG без изменения реализации поиска.

Эквивалентное приложение заключается в том, чтобы рассматривать ключи trie-дерева как соответствующие строкам таблицы истинности булевской функции, на которых эта функция принимает истинное значение (см. упражнения 18.84–18.87). Двоичный DAG есть модель экономичной схемы, которая вычисляет эту функцию. В таком приложении двоичные графы DAG называются *схемами BDD* (binary decision diagram – двоичная схема решений).

Данные приложения заставляют нас обратиться в двух следующих разделах к изучению алгоритмов обработки графов DAG. Эти алгоритмы не только приводят к реализации эффективных и полезных функций АТД графов DAG, но и позволяют оценить трудность обработки орграфов. Как мы увидим далее, даже если графы DAG покажутся значительно более простыми структурами, чем орграфы общего вида, очевидно, что некоторые фундаментальные задачи решаются ничуть не проще.

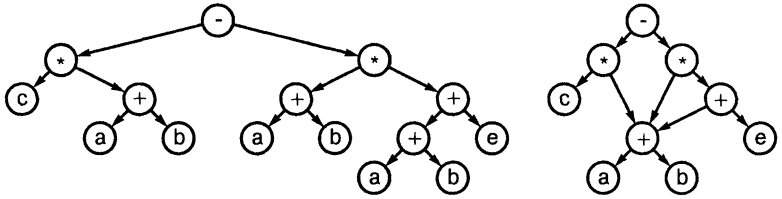


РИСУНОК 19.19. ПРЕДСТАВЛЕНИЕ АРИФМЕТИЧЕСКОГО ВЫРАЖЕНИЕ С ПОМОЩЬЮ ГРАФА DAG

Оба изображенные на диаграмме графы DAG являются представлениями одного и того же арифметического выражения ***(c*(a+b))-((a+b)*(a+b)+e))****. В двоичном дереве грамматического разбора, которое показано в левой части диаграммы, листы представляют операнды, а все внутренние узлы — операции, выполняемые над выражениями, представленными двумя поддеревьями (см. рис. 5.31). Граф DAG, изображенный в правой части диаграммы, — это более компактное представление того же дерева. Более того, мы можем вычислить значение выражения за время, пропорциональное размеру DAG, который обычно значительно меньше, чем размер дерева (см. упражнения 19.112 и 19.113).*

Программа 19.5. Представление бинарного дерева в виде двоичного графа DAG

Представленный здесь фрагмент программного кода реализует обход в обратном направлении, который выполняет построение двоичного DAG, соответствующего структуре двоичного дерева (см. глава 12) путем выявления общих поддеревьев. Он использует индексирующий класс, подобный классу **ST** из программы 17.15 (скорректированный с целью обеспечения ввода пар чисел вместо ввода строк с клавиатуры) с тем, чтобы присвоить уникальное целочисленное значение каждой отдельной древовидной структуре для применения в представлении DAG в виде вектора двоичных целочисленных структур (см. рис. 20). Пустому дереву (нулевая связь) присваивается индекс **0**, дереву с единственным узлом — индекс **1** и т.д.

Индекс, соответствующий каждому поддереву, вычисляется в рекурсивном режиме. Затем создается ключ, обладающий таким свойством, что каждый узел одного и того же поддерева будет иметь тот же индекс, и этот индекс возвращается после заполнения связей соответствующего ребра графа DAG (поддерево).

```
int compressR(link h)
  { STx st;
    if (h == NULL) return 0;
    l = compressR(h->l);
    r = compressR(h->r);
    t = st.index(l, r);
    adj[t].l = l; adj[t].r = r;
    return t;
  }
```

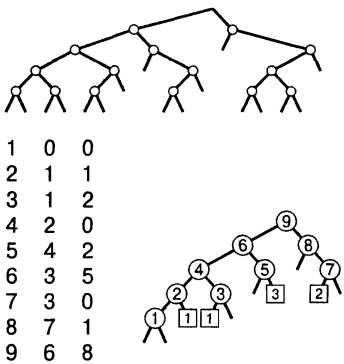


РИСУНОК 19.20. СЖАТИЕ БИНАРНОГО ДЕРЕВА

Таблица из девяти пар целых чисел, приведенная в левой нижней части рисунка, является компактным представлением двоичного DAG (внизу справа), которое представляет собой сжатую версию двоичного DAG двоичной древовидной структуры, показанной в верхней части рисунка. Метки узлов не хранятся в явном виде в рассматриваемой структуре данных: таблица представляет восемнадцать ребер ***1-0, 1-0, 2-1, 2-1, 3-1, 3-2*** *и т.д., однако обозначает левое и правое ребра, исходящие из каждого узла (как в бинарном дереве), при этом исходная вершина каждого ребра табличном индексе выражается неявно.*

Алгоритм, который зависит только от формы дерева, будет эффективно работать на графе DAG. Например, предположим, что рассматриваемое дерево есть trie-дерево существования для двоичных ключей, соответствующих листам дерева, так что оно представляет ключи **0000**, **0001**, **0010**, **0110**, **1100**, **1101**. Успешный поиск ключа **1101** смещается в trie-дереве вправо, вправо, влево и вправо к концу в концевой вершине. В графе DAG тот же поиск проходит от **9** к **8**, к **7**, к **2** и к **1**.

Упражнения

▷ **19.75**. Постройте реализацию класса DFS, предназначенную для использования клиентскими программами с целью проверки на наличие в графе DAG циклов.

○ **19.76**. Напишите программу, которая генерирует случайные графы DAG путем построения случайных орграфов, выполняя поиск в глубину из случайной исходной точки и отбрасывание обратных ребер (см. упражнение 19.40). Проведите эксперименты с целью выбора подходящих значений параметров, обеспечивающих получение графов DAG с E ребрами при заданном числе V.

▷ **19.77**. Сколько узлов содержит дерево и граф DAG, соответствующие рис. 19.18 и представляющие методы вычисления F_N, N-го числа Фибоначчи.

19.78. Постройте DAG, соответствующий примеру динамического программирования, для модели рюкзака из главы 5 (см. рис. 5.17).

○ **19.79**. Разработайте АТД для двоичных графов DAG.

● **19.80**. Может ли каждый DAG быть представлен как двоичный DAG (см. свойство 5.4)?

○ **19.81**. Напишите функцию, которая выполняет неупорядоченный обход двоичного графа DAG с одним источником. То есть, эта функция должна посетить все вершины, которые можно достичь через левое ребро, затем посетить исходную вершину, после чего посетить все вершины, которые могут быть достигнуты через правое ребро.

▷ **19.82**. В стиле рис. 19.20 постройте trie-дерево существования и соответствующий двоичный DAG для ключей **01001010 10010101 00100001 11101100 01010001 00100001 00000111 01010011**.

19.83. Постройте реализацию АТД, основанную на построении trie-дерева существования на базе некоторого набора 32-битовых ключей, сжимающих его до двоичного DAG, с последующим использованием этой структуры данных для поддержки запросов о существовании.

○ **19.84**. Начертите схему BDD для таблицы истинности для функции от четырех переменных проверки на нечетность, которая принимает значение **1** в том и только том случае, когда число переменных, принимающих значение **1** будет нечетным.

19.85. Напишите функцию, которая принимает в качестве аргумента 2^n-разрядную таблицу истинности и возвращает соответствующую схему BDD. Например, для заданного ввода **1110001000001100** программа должна вернуть представление двоичного графа DAG (см. рис. 19.20).

19.86. Напишите функцию, которая принимает в качестве аргумента 2^n-разрядную таблицу истинности, вычисляет каждую перестановку аргументов ее переменных и, пользуясь решением упражнения 19.85, возвращает перестановку, которая приводит к наименьшей схеме BDD.

• **19.87.** Проведите эмпирические исследования с целью определения эффективности стратегии упражнения 19.85 для различных булевых функций, как стандартных, так и случайных.

19.88. Напишите программу, аналогичную программе 19.5, которая поддерживает удаление общих подвыражений: пусть дано бинарное дерево, представляющее арифметическое выражение; вычислите двоичный граф DAG, представляющий то же выражение, из которого удаленны общие подвыражения.

○ **19.89.** Начертите неизоморфные графы DAG с двумя, тремя, четырьмя и пятью вершинами.

•• **19.90.** Сколько существует различных графов DAG с V вершинами E ребрами?

••• **19.90.** Сколько существует различных графов DAG с V вершинами E ребрами, если мы считаем два DAG различными только в тех случаях, когда они не изоморфны?

19.6. Топологическая сортировка

Цель топологической сортировки заключается в том, чтобы подготовить обработку вершин графа DAG таким образом, чтобы каждая вершина была обработана до того, как будут обработаны все вершины, на которые она указывает. Существуют два естественных способа определения этой базовой операции; по сути дела, они эквивалентны. Обе задачи требуют перестановки целых чисел от 0 до V-1, которые мы, как обычно, помещаем в векторы, индексированные именами вершин.

Топологическая сортировка (с переименованием). Пусть задан некоторый DAG, нужно переименовать его вершины таким образом, чтобы каждое ориентированное ребро вело из вершины с меньшим номером в вершину с большим номером (см. рис. 19.21).

Топологическая сортировка (с перегруппировкой). Пусть задан некоторый DAG, нужно перегруппировать его вершины по горизонтали таким образом, чтобы все ориентированные ребра были направлены слева направо (см. рис. 19.22).

Как показывает рис. 19.22, легко установить, что переименование и перегруппировка перестановок обратны по отношению друг к другу: если задана перегруппировка, то переименование можно получить, присвоив **0** первой вершине списка, **1** второй вершине списка и т.д. Например, если в векторе **ts** вершины размещены в порядке топологической сортировки, то цикл

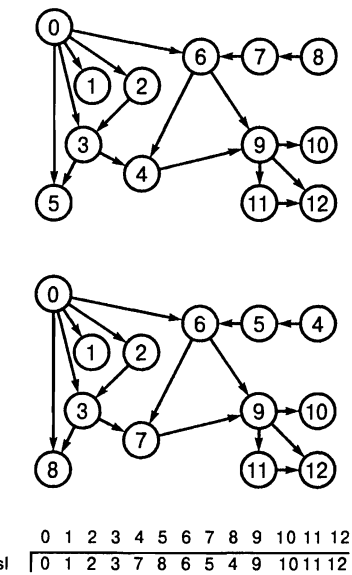
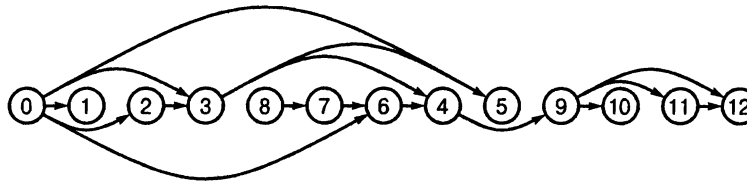


РИСУНОК 19.21. ТОПОЛОГИЧЕСКАЯ СОРТИРОВКА (С ПЕРЕИМЕНОВАНИЕМ)

*Пусть задан произвольный DAG (вверху), топологическая сортировка позволяет нам переименовать его вершины так, что каждое ребро ведет из вершины с меньшим номером в вершину с большим номером (внизу). В этом примере мы переименовываем вершины **4**, **5**, **7** и **8**, соответственно, в **7**, **8**, **5** и **4**, как показано в массиве **tsI**. Существует большое число переименований, позволяющих получить требуемые результаты.*



	0	1	2	3	4	5	6	7	8	9	10	11	12
ts	0	1	2	3	8	7	6	4	5	9	10	11	12
tsI	0	1	2	3	7	8	6	5	4	9	10	11	12

РИСУНОК 19.22. ТОПОЛОГИЧЕСКАЯ СОРТИРОВКА (С ПЕРЕГРУППИРОВКОЙ)

Эта диаграмма может служить другой точкой зрения на топологическую сортировку, представленную на рис. 19.21; в ней мы определяем способ перегруппировки вершин, а не их переименования. Когда мы группируем вершины в порядке, указанном в массиве **ts**, *слева направо, то все ориентированные ребра направлены слева направо. Инверсия перестановки* **ts** *есть перестановка* **tsI**, *которая определяет переименование, описанное на рис. 19.21.*

```
for (i = 0; i < V; i++) tsI[ts[i]] = i;
```

определяет переименование в векторе **tsI**, индексированном именами вершин. И наоборот, мы можем получить перегруппировку из переименования с помощью цикла

```
for (i = 0; i < V; i++) ts[tsI[i]] = i;
```

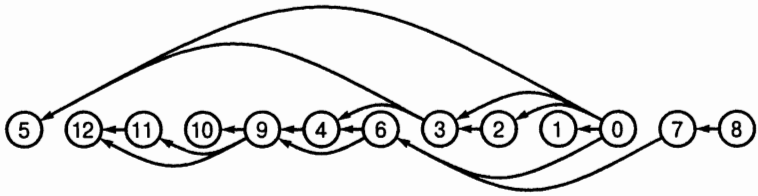
который помещает первой в список вершину, получающую метку **0**, второй вершину, получающую метку **1**, и т.д. Чаще всего мы используем термин *топологическая сортировка (topological sort)* в отношении версии задачи, связанной с перегруппировкой. Обратите внимание на то обстоятельство, что **ts** *не есть* вектор, индексированный именами вершин.

В общем случае, порядок вершин, установленный топологической сортировкой, не уникален. Например,

```
8  7  0  1  2  3  6  4  9  10  11  12  5
0  1  2  3  8  6  4  9  10 11  12  5   7
0  2  3  8  6  4  7  5  9  10  1   11  12
8  0  7  6  2  3  4  9  5  1   11  12  10
```

суть логические сортировки примера DAG, представленного на рис. 19.6 (существует множество других примеров). В приложении составления расписаний подобная ситуация возникает всякий раз, когда одна из задач прямо или косвенно не находится в прямой или непрямой зависимости от другой задачи, и в силу этого обстоятельства она может выполняться перед либо после другой задачи (и даже одновременно). Число возможных расписаний возрастает экспоненциально с увеличением количества таких пар задач.

Как уже отмечалось, иногда бывает полезно рассматривать ребра орграфа в другом аспекте: когда мы говорим, что ребро направлено из *s* в *t*, это означает, что вершина *s* "зависит" от вершины *t*. Например, вершины могут представлять термины, определения которых даются в некоторой книге, при этом ребро направлено из *s* в *t*, если определение *s* использует *t*. В этом случае полезно установить порядок, по условиям которого определение каждого термина дается перед тем, как оно будет использоваться в другом определении. Использование подобной упорядоченности соответствует такому построению вершин в линию, что все ребра направлены справа налево – получаем *обратную топологическую сортировку (reverse topological sort)*. Рисунок 19.23 может служить иллюстрацией обратной топологической сортировки на рассматриваемом нами демонстрационном примере DAG.



	0	1	2	3	4	5	6	7	8	9	10	11	12
ts	5	12	11	10	9	4	6	3	2	1	0	7	8
tsI	10	9	8	7	5	0	6	11	12	4	3	2	1

РИСУНОК 19.23. ОБРАТНАЯ ТОПОЛОГИЧЕСКАЯ СОРТИРОВКА

В условиях обратной топологической сортировки, выполняемой на демонстрационном примере орграфа, все ребра направлены справа налево. Нумерация вершин, задаваемая обратной подстановкой tsI, порождает граф, в котором каждое ребро направлено из вершины с большим номером в вершину с меньшим номером.

Теперь выясняется, что мы уже рассматривали алгоритм обратной топологической сортировки, таковым является наш старый знакомый — стандартный рекурсивный поиск в глубину! Если на вход подать DAG, то *нумерация вершин во время обхода в обратном порядке* размещает вершины в порядке обратной топологии. Иначе говоря, мы выполняем нумерацию каждой вершины как завершающее действие рекурсивной функции DFS аналогично вектору **post** в программе 19.2, которая представляет собой реализацию поиска в глубину (DFS). Как видно из рис. 19.24, использование такой нумерации эквивалентно нумерации узлов в лесе DFS при обходе в обратном порядке и эквивалентно топологической сортировке: вектор **post**, индексированный по именам вершин, позволяет получить переименование и его инверсию, показанные на рис. 19.23, – получается обратная топологическая сортировка графа DAG.

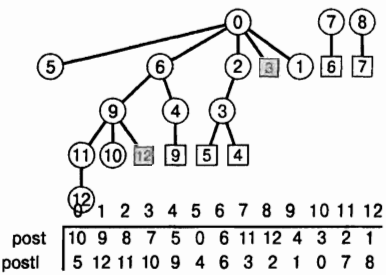


РИСУНОК 19.24. ЛЕС DFS ДЛЯ ГРАФА DAG

Лес DFS заданного орграфа не имеет обратных ребер (ребер, ведущих в узлы с большими номерами в обратном порядке обхода вершин графа) тогда и только тогда, когда этот орграф есть DAG. Ребра, не принадлежащие деревьям в этом DFS лесе для графа DAG, представленного на рис. 19.21, – это либо прямые ребра (заштрихованные квадратики), либо поперечные ребра (незаштрихованные квадратики). Последовательность, в которой посещаются вершины при обходе леса в обратном порядке, показанная в нижней части диаграммы, представляет собой обратную топологическую сортировку (см. рис. 19.23).

Свойство 19.11. *Нумерация при обходе в обратном порядке при поиске в глубину представляет собой обратную топологическую сортировку для любого графа DAG.*

Доказательство: Предположим, что **s** и **t** – две вершины, такие, что **s** появляется раньше **t** при нумерации в обратном порядке, даже если в графе существует направленное ребро **s-t**. Поскольку в момент, когда мы присваиваем **s** ее номер, мы уже выполнили рекурсивный поиск в глубину для вершины **s**, мы, в частности, проверили и ребро **s-t**. Но если бы **s-t** было деревом, прямым или поперечным ребром, рекурсивный поиск в глубину для **t** был бы уже выполнен и **t** имела бы меньший номер. Однако, **s-t** не может быть обратным ребром, поскольку это оз-

начало бы наличие в графе цикла. Полученное противоречие доказывает невозможность существования ребра **s-t**. ■

Таким образом, мы легко можем заставить стандартный поиск в глубину выполнять топологическую сортировку, о чем свидетельствует программа 19.6. Эта программа выполняет обратную топологическую сортировку: она выполняет перестановку во время обхода в обратном порядке и ее инверсию, благодаря чему клиентские программы получают возможность переименовать или перегруппировать вершины.

Программа 19.6. Обратная топологическая сортировка

Этот класс DFS вычисляет нумерацию леса DFS (обратная топологическая сортировка). Клиентские программы могут использовать объект **TS** для переименования вершин графа DAG с таким расчетом, чтобы каждое ребро указывало из вершины с некоторым номером на вершину с более низким номером или чтобы вершины были упорядочены в такой последовательности, что исходная вершина каждого ребра появлялась после вершины назначения (см. рис. 19.23).

```
template <class Dag> class dagTS
{ const Dag &D;
  int cnt, tcnt;
  vector<int> pre, post, postI;
  void tsR(int v)
    {
      pre[v] = cnt++;
      typename Dag::adjIterator A(D, v);
      for (int t = A.beg(); !A.end(); t = A.nxt())
        if (pre[t] == -1) tsR(t);
      post[v] = tcnt; postI[tcnt++] = v;
    }
public:
  dagTS(const Dag &D) : D(D), tcnt(0), cnt(0),
    pre(D.V(), -1), post(D.V(), -1), postI(D.V(), -1)
      { for (int v = 0; v < D.V(); v++)
        if (pre[v] == -1) tsR(v); }
  int operator[](int v) const { return postI[v]; }
  int relabel(int v) const { return post[v]; }
};
```

С точки зрения вычислений, различия между топологической сортировкой и обратной топологической сортировкой не критичны. Мы просто можем изменить операцию **[]** так, чтобы она возвращала значение **postI[G.V()-1-v]**, либо модифицировать реализацию одним из следующих способов:

- Выполнить обратную топологическую сортировку на обращении заданного графа DAG.
- Вместо того чтобы использовать номер вершины в качестве индекса при нумерации в обратном порядке, следует затолкнуть номер вершины в стек как завершающее действие рекурсивной процедуры. После завершения просмотра вытолкните вершины из стека. Они выходят из стека в топологическом порядке.
- Пронумеруйте вершины в обратном порядке (начните с $V-1$ и выполните отсчет до 0). При желании вычислите обратную нумерацию вершин, чтобы получить топологический порядок.

Доказательство того, что эти вычисления позволяют получить правильное топологическое упорядочение, оставляем вам на самостоятельную проработку (см. упражнение 19.97).

Чтобы реализовать первый из вариантов, перечисленных в предыдущем параграфе в контексте разреженных графов (представленных в виде списков смежных вершин), нам может потребоваться программа 19.1 для вычисления обратного графа. Решение этой задачи требует увеличения используемого пространства памяти в два раза, что становится обременительным в случае крупных графов. Что касается насыщенных графов (представленных в виде матриц смежности), то как было отмечено в разделе 19.1, мы можем выполнить поиск в глубину на обратном графе, не прибегая к использованию дополнительного пространства памяти или без выполнения дополнительных работ, а только путем простой замены строк на столбцы при обращении к матрице смежности, как показывает программа 19.7.

Далее мы рассмотрим альтернативный классический метод топологической сортировки, которые имеет определенное сходство с BFS (breadth-first search — поиск в ширину) (см. раздел 18.7.). Он основан на следующем свойстве графа DAG.

Свойство 19.12. *У каждого графа DAG имеется, по меньшей мере, один исток и, по меньшей мере, один сток.*

Доказательство: Предположим, что задан DAG, у которого нет стоков. Тогда, отправляясь из любой вершины, мы можем построить ориентированный путь произвольной длины, следуя из этой вершины вдоль любого ребра в любую другую вершину (существует, по меньшей мере, одно такое ребро, поскольку у графа DAG нет стоков), с последующим следованием из этой вершины вдоль другого ребра и т.д. Но как только мы посетим $V+1$ вершину, мы должны попасть в ориентированный цикл в соответствии с принципом картотечного ящика (см. свойство 19.6), что противоречит предположению о том, что имеется DAG. Таким образом, что в графе DAG существует, по меньшей мере, один сток. Отсюда также следует, что в каждом графе DAG присутствует, по меньшей мере, один исток, ибо исток представляет собой обращение стока. ■

Из этого факта мы можем вывести алгоритм топологической сортировки: пометим любой исток наименьшей неиспользованной меткой, затем удалим его и пометим остальную часть графа DAG, применив тем же алгоритм. На рис. 19.25 показана трассировка этого алгоритма, примененного к рассматриваемому нами примеру графа DAG.

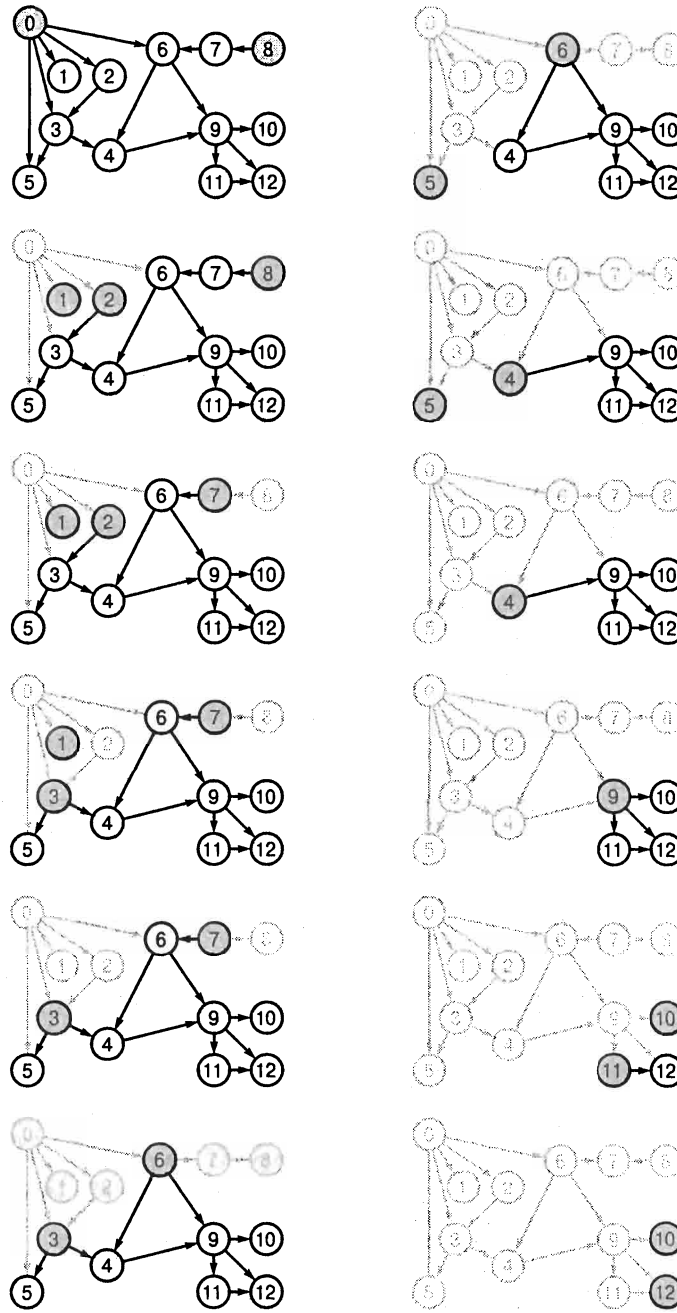
Программа 19.7. Топологическая сортировка

Если использовать данную реализацию функции **tsR** из программы 19.6, конструктор вычисляет топологическую сортировку, но не ее обращение (для любого DAG это реализация, которая поддерживает функцию **edge**), поскольку она замещает вызов функции **edge(v, w)** при поиске в глубину на вызов **edge(w, v)**, тем самым выполняя обработку обратного графа (см. текст).

```
void tsR(int v)
  {
    pre[v] = cnt++;
    for (int w = 0; w < D.V(); w++)
      if (D.edge(w, v))
        if (pre[w] == -1) tsR(w);
    post[v] = tcnt; postI[tcnt++] = v;
  }
```

РИСУНОК 19.25. ТОПОЛОГИЧЕСКАЯ СОРТИРОВКА ГРАФА DAG МЕТОДОМ УДАЛЕНИЯ ИСТОКОВ

Вершина 0, будучи истоком (на нее не указывает ни одно ребро), может оказаться первой при топологической сортировке этого графа (слева вверху). Если удалить 0 (а также все ребра, которые ведут из этой вершины в другую вершину), то истоками в получающемся при этом графе DAG становятся вершины 1 и 2 (слева, вторая диаграмма сверху), которые можно сортировать при помощи того же алгоритма. Этот рисунок служит иллюстрацией работы программы 19.8, которая производит выбор истоков (заштрихованные вершины в каждой диаграмме), используя для этой цели дисциплину обслуживания FIFO, хотя на каждом шаге может быть выбран любой из источников. Посмотрите также на рис. 19.26, на котором представлено содержимое структур данных, способное повлиять на выбор, который выполняет рассматриваемый алгоритм. В результате топологической сортировки, показанной на этом рисунке, получается следующий порядок узлов: 0 8 2 1 7 3 6 5 4 9 11 10 12.



Эффективная реализация этого алгоритма есть классический пример алгоритма топологической сортировки (*см. раздел ссылок*). Во-первых, возможно существование некоторого множества истоков, и в силу этого обстоятельства для их отслеживания мы должны поддерживать очередь (для этого подойдет любая обобщенная очередь). Во-вторых, мы должны выявить в заданном графе DAG истоки, которые остаются после удаления того или иного истока. Мы можем решить эту задачу, поддерживая вектор, проиндексированный именами вершин, который отслеживает полустепень захода каждой вершины. Вершины с полустепенью захода 0 суть истоки, поэтому мы можем инициализировать очередь с одним просмотром графа DAG (используя поиск в глубину или какой-либо другой метод, обеспечивающий просмотр всех ребер). Затем, пока очередь истоков не опустеет, мы выполняем следующие операции:

- Удаляем исток из очереди и присваиваем ему соответствующую метку.
- Уменьшаем на единицу значения элементов вектора полустепеней захода, соответствующих вершинами назначения каждого из ребер удаленной вершины.
- Если в результате уменьшения значения какой-либо элемент принимает значение 0, вставляем соответствующую вершину в очередь истоков.

Программа 19.8 содержит реализацию этого метода, в которой применяется очередь FIFO, а рис. 19.26 служит иллюстрацией ее работы на примере рассматриваемого нами DAG, при этом динамика примера, представленного на рис. 19.25, обрастает дополнительными подробностями.

Очередь истоков не опустеет до тех пор, пока не будет помечена каждая вершина графа DAG, поскольку подграф, индуцированный еще не помеченными вершинами, всегда есть DAG, и каждый DAG имеет, по меньшей мере, один исток. В самом деле, мы можем использовать алгоритм для проверки, является ли заданный граф графом DAG, предполагая, что должен существовать цикл в подграфе, индуцированном еще непомеченными вершинами, если очередь опустеет до того как все вершины будут помечены (см. упражнение 19.104).

0	1	2	3	4	5	6	7	8	9	10	11	12	
0	1	1	2	2	2	2	1	0	2	1	1	2	0 8
0	0	0	1	2	1	1	1	0	2	1	1	2	8 2 1
0	0	0	1	2	1	1	0	0	2	1	1	2	2 1 7
0	0	0	0	2	1	1	0	0	2	1	1	2	1 7 3
0	0	0	0	2	1	1	0	0	2	1	1	2	7 3
0	0	0	0	2	1	0	0	0	2	1	1	2	3 6
0	0	0	0	1	0	0	0	0	2	1	1	2	6 5
0	0	0	0	0	0	0	0	0	1	1	1	2	5 4
0	0	0	0	0	0	0	0	0	1	1	1	2	4
0	0	0	0	0	0	0	0	0	0	1	1	2	9
0	0	0	0	0	0	0	0	0	0	0	0	1	11 10
0	0	0	0	0	0	0	0	0	0	0	0	0	10 12
0	0	0	0	0	0	0	0	0	0	0	0	0	12

РИСУНОК 19.26. ТАБЛИЦА ПОЛУСТЕПЕНЕЙ ЗАХОДА И СОДЕРЖИМОЕ ОЧЕРЕДИ

Данная последовательность описывает содержимое таблицы полустепеней захода (слева) и очередь истоков (справа) во время выполнения программы 19.8 на примере DAG, соответствующего рис. 19.25. В любой заданный момент времени очередь истоков содержит узлы с полустепенью захода, равной 0. Производя считывания сверху вниз мы удаляем из очереди истоков крайний левый узел, уменьшаем на единицу значения полустепени захода элементов, соответствующих каждому ребру, исходящему из этого узла, и добавляем в очередь истоков все вершины, элементы таблицы которых принимают значения 0. Например, вторая строка таблицы отражает результат удаления вершины 0 из очереди истоков, с последующим (поскольку DAG содержит ребра 0-1, 0-2, 0-3, 0-5 и 0-6) уменьшением на единицу значений элементов, соответствующих вершинам 1, 2, 3, 5 и 6, и добавлением вершин 2 и 1 в очередь истоков (поскольку значения их полустепеней захода уменьшилось до 0). Считывание крайних слева элементов в очереди истоков сверху вниз дает топологическое упорядочение рассматриваемого графа.

Программа 19.8. Топологическая сортировка, основанная на очереди истоков

Этот класс реализует тот же интерфейс, что и программы 19.7 и 19.8. Он поддерживает очередь истоков и использует таблицу, которая отслеживает полустепени захода каждой вершины графа DAG, индуцированного вершинами, которые еще не удалены из очереди.

Когда мы удаляем исток из очереди, мы уменьшаем значения полустепеней захода, соответствующих каждой вершине в его списке смежных вершин (и помещаем в очередь каждую вершину, соответствующую элементу таблицы, принимающему значение 0). Вершины покидают очередь в порядке топологической сортировки.

```
#include "QUEUE.cc"
template <class Dag> class dagTS
{ const Dag &D;
  vector<int> in, ts, tsI;
public:
  dagTS(const Dag &D) : D(D),
    in(D.V(), 0), ts(D.V(), -1), tsI(D.V(), -1)
  { QUEUE<int> Q;
    for (int v = 0; v < D.V(); v++)
      {
        typename Dag::adjIterator A(D, v);
        for (int t = A.beg(); !A.end(); t = A.nxt())
          in[t]++;
      }
    for (int v = 0; v < D.V(); v++)
      if (in[v] == 0) Q.put(v);
    for (int j = 0; !Q.empty(); j++)
      {
        ts[j] = Q.get(); tsI[ts[j]] = j;
        typename Dag::adjIterator A(D, ts[j]);
        for (int t = A.beg(); !A.end(); t = A.nxt())
          if (--in[t] == 0) Q.put(t);
      }
  }
  int operator[](int v) const { return ts[v]; }
  int relabel(int v) const { return tsI[v]; }
};
```

Обработка вершин в порядке, образованном топологической сортировкой, представляет собой базовую технологию обработки графов DAG. Классическим примером может служить задача определения длины самого длинного пути в графе DAG. Рассматривая вершины в порядке, обратном устанавливаемому топологической сортировкой, легко вычислить самый длинный путь, начинающийся в каждой вершине v. Для этого потребуется добавить единицу к максимальной из длин путей, начинающихся в каждой из вершин, достижимых из v через одно ребро. Благодаря топологической сортировке все такие длины известны во время обработки вершины v, так что никакие другие пути из v после этого выявляться не будут. Например, выполняя сканирование слева направо обратной топологической сортировки, показанной на рис. 19.23, мы можем достаточно быстро вычислить следующую таблицу длин максимальных путей, возникающих в каждой вершине демонстрационного графа, который показан на рис. 19.21.

```
5  12  11  10  9  4  6  3  2  1  0  7  8
0   0   1   0  2  3  4  4  5  0  6  5  6
```

Например, **6**, соответствующая **0** (третий столбец справа), показывает, что существует путь длиной **6**, начинающийся в **0**, о чем мы знаем, поскольку существует ребро **0-2**; ранее мы определили, что длина самого длинного пути из вершины **2** равна **5** и что ни одно из ребер, исходящих из **0**, не ведет в узел, имеющий более длинный путь.

Всякий раз, когда мы используем топологическую сортировку для подобного рода приложений, перед нами встает проблема выбора одного из следующих способов разработки реализации:

- Использование класса **DAGts** в АТД DAG с последующей обработкой вершин в очередности, задаваемой вектором, который он вычисляет.
- Обработка вершин после рекурсивных вызовов в рамках DFS.
- Обработка вершин по мере того, как они выбираются из очереди в процессе топологической сортировки на базе очереди истоков.

В литературе показано, что все эти методы используются в реализациях, выполняющих обработку DAG, в этой связи следует подчеркнуть, что все они эквивалентны. Мы рассмотрим другие приложения топологической сортировки в упражнениях 19.111 и 19.114 и в разделах 19.7 и 21.4.

Упражнения

▷ **19.92**. Напишите функцию, которая проверяет, является ли заданная перестановка вершин графа DAG подходящей топологической сортировкой этого графа DAG.

19.92. Сколько возможно различных топологических сортировок на графе DAG, изображенном на рис. 19.6?

▷ **19.94**. Приведите пример леса DFS и обратной топологической сортировки, которые получаются в результате выполнения стандартного поиска в глубину на представлении в виде списков смежных вершин (с нумерацией при обходе в обратном порядке) следующего графа DAG:

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4 4-3 2-3.

○ **19.95**. Приведите пример леса DFS и обратной топологической сортировки, которые получаются в результате построения стандартного представления в виде списков смежных вершин графа DAG

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4 4-3 2-3,

с последующим применением программы 19.1 для построения обратной сортировки, после чего проведите поиск в ширину на представлении в виде списков смежных вершин с нумерацией, установленной при обходе в обратном порядке.

▷ **19.96**. Программа 19.6 использует нумерацию при обходе в обратном порядке с целью выполнения обратной топологической сортировки – почему нельзя воспользоваться нумерацией, установленной при обходе в прямом порядке? Обоснуйте свою аргументацию на примере графа с тремя вершинами.

● **19.97**. Докажите правильность каждого из трех приведенных в тексте предположений, касающихся модификации поиска в глубину с применением такой нумерации, установленной при обходе в обратном порядке, при которой вычисляется топологическая сортировка, но не обратная топологическая сортировка.

▷ **19.98**. Приведите пример леса DFS и обратной топологической сортировки, которые получаются в результате выполнения стандартного поиска в глубину с неявным обращением на представлении в виде списков смежных вершин (с нумерацией при обходе в обратном порядке) следующего графа DAG:

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4 4-3 2-3

(см. программу 19.7).

• **19.99**. Пусть задан некоторый DAG, существует ли топологическая сортировка, которую нельзя получить путем применения алгоритма, построенного на базе поиска в глубину, независимо от порядка выбора вершин, смежных с заданной? Обоснуйте свой ответ доказательством.

▷ **19.100**. Покажите в стиле рис. 19.26 процесс топологической сортировки графа DAG

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4 4-3 2-3

с помощью алгоритма, использующего очередь истоков (программа 19.8).

▷ **19.101**. Приведите пример топологической сортировки, которая получается, когда в примере, представленном на рис. 19.25, в качестве структуры данных используется стек, а не очередь.

• **19.102**. Пусть задан некоторый граф DAG, существует ли топологическая сортировка, которую нельзя получить путем применения алгоритма, построенного на базе поиска в глубину, независимо от дисциплины, реализуемой очередью? Обоснуйте свой ответ доказательством.

19.103. Измените алгоритм топологической сортировки на базе очереди истоков с таким расчетом, чтобы можно было пользоваться обобщенной очередью. Воспользуйтесь модифицированным алгоритмом с очередью LIFO (Last In First Out – последним прибыл, первым обслужен), со стеком и рандомизированной очередью.

▷ **19.104**. Воспользуйтесь программой 19.8 для построения реализации класса, осуществляющего проверку наличия циклов в заданном графе DAG (см. упражнение 19.75).

○ **19.105**. Преобразуйте алгоритм топологической сортировки с использованием очереди истоков в алгоритм, использующий очередь стоков для выполнения обратной топологической сортировки.

19.106. Напишите программу, которая обобщает все возможные топологические упорядочения заданного графа DAG либо, если число таких упорядочений превышает границу, заданную в качестве аргумента, печатает это число.

19.107. Напишите программу, которая преобразует любой орграф с V вершинами и E ребрами в графе DAG путем выполнения топологической сортировки на основе поиска в глубину и изменения ориентации любого встреченного обратного ребра. Докажите, что эта стратегия всегда приводит к созданию DAG.

•• **19.108**. Напишите программу, которая осуществляет построение любого DAG с V вершинами и E ребрами с равной вероятностью (см. упражнение 17.70).

19.109. Сформулируйте необходимые и достаточные условия для того, чтобы для конкретного графа DAG существовало только одно возможное топологически отсортированное упорядочение его вершин.

▷ **19.110**. Прогоните эмпирические тесты с целью сравнения алгоритмов топологической сортировки, заданной в этом разделе для различных графов DAG (см. упражнение 19.2, упражнение 19.76, упражнение 19.107 и упражнение 19.108). Протестируйте свою программу в соответствии с изложенным в упражнении 19.11 (для разреженных графов) и в соответствии с изложенным в упражнении 19.12 (для насыщенных графов).

▷ **19.111**. Модифицируйте программу 19.8 таким образом, чтобы она могла вычислять число различных простых путей из любого истока в каждую вершину графа DAG.

○ **19.112**. Разработайте класс, который дает оценку графу DAG, представляющему арифметические выражения (см. рис. 19.19). Воспользуйтесь вектором, проиндексированным по именам вершин, для хранения значений, соответствующих каждой вершине. Предполагается, что значения, соответствующие листьям, устанавливаются заранее.

○ **19.113**. Дайте описание семейства арифметических выражений, обладающих тем свойством, что размер дерева выражения экспоненциально больше, чем размер соответствующего DAG (в связи с чем время выполнения вашей программы из упражнения 19.112 для этого DAG пропорционально логарифму от времени ее выполнения для дерева).

○ **19.114**. Разработайте метод обнаружения простейшего ориентированного пути максимальной длины на графе DAG, время выполнения которого пропорционально V. Воспользуйтесь полученным методом для реализации класса, выполняющего распечатку гамильтонова пути в заданном DAG, если таковой имеется.

19.7 Достижимость в графе DAG

Мы завершим наше изучение графов DAG рассмотрением задачи вычисления транзитивного замыкания DAG. Можно ли разработать такие алгоритмы для графов DAG, которые обладают большей эффективностью, чем алгоритмы для обобщенных орграфов, которые исследовались в разделе 19.3?

Любой метод топологической сортировки может служить основой алгоритмов транзитивного замыкания DAG, например, мы осуществляем обход вершин в обратном топологическом порядке, вычисляя при этом вектор достижимости для каждой вершины (он является строкой матрицы транзитивного замыкания) из строк, соответствующих смежным с ней вершинам. Обратная топологическая сортировка гарантирует, что все эти строки будут вычислены заблаговременно. В итоге мы проверяем каждый из V элементов вектора, соответствующего вершине назначения каждого их E ребер, выполняется ли его обработка за время, пропорциональное EV. И хотя этот метод достаточно прост для программной реализации, по эффективности обработки графов DAG он ничуть не лучше, чем для обработки орграфов общего типа.

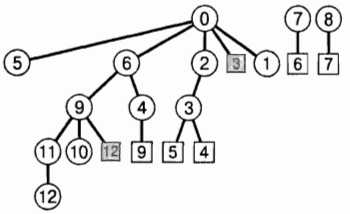
Когда мы используем стандартный поиск в глубину для топологической сортировки (см. программу 19.7), мы можем повысить ее производительность для некоторых видов графов DAG, как показывает программа 19.9. Поскольку в графах DAG нет циклов, то ни в каких поисках в глубину нет обратных ребер. Что еще важнее, как поперечные, так и прямые ребра ведут в узлы, в которых DFS завершен. Чтобы воспользоваться этим обстоятельством, мы разработаем рекурсивную функцию, вычисляющую все вершины, достижимые из заданной исходной вершины, тем не менее (как это характерно для поиска в глубину), мы не производим рекурсивных вызовов для вершин, для которых множество достижимых вершин уже было вычислено. В этом случае достижимые верши-

ны представлены одной из строк транзитивного замыкания, а рекурсивная функция выполняет операцию логического **or** над всеми строками, ассоциированными со смежными вершинами. Что касается древесных ребер, то мы производим рекурсивный вызов для вычисления этой строки; в случае поперечных ребер мы можем отказаться от рекурсивного вызова, поскольку мы знаем, что эта строка уже была вычислена в результате предыдущего вызова; в случае прямых ребер мы может полностью отказаться от вычислений, поскольку любые достижимые узлы, которые они могут добавить, уже включены в множество узлов, достижимых из вершины назначения (ниже и раньше в дереве DFS).

Использование этой версии поиска в глубину можно охарактеризовать как применение динамического программирования для вычисления транзитивного замыкания, поскольку мы используем результаты, которые уже были вычислены (и сохранены в строках матриц смежности, соответствующих вершинам, которые обрабатывались раньше) с тем, чтобы не совершать ненужных рекурсивных вызовов. Рисунок 19.27 служит иллюстрацией вычисления транзитивного замыкания графа DAG, представленного в качестве примера на рис. 19.6.

Свойство 19.13. *С помощью динамического программирования и поиска в глубину мы можем обеспечить постоянное время ответа на запрос на абстрактное транзитивное замыкание графа DAG, при этом на предварительную обработку (для подготовки вычисления транзитивного замыкания) затрачивается пространство памяти, пропорциональное V^2, и время, пропорциональное $V^2 + VX$, где X – это число поперечных ребер в лесе DFS.*

Доказательство: Доказательство непосредственно следует по индукции из рекурсивной функции, используемой в программе 19.9. Мы посещаем вершины в обратном топологическом порядке. Каждое ребро указывает на вершину, для которой мы уже вычислили множество всех достижимых вершин, в силу этого обстоятельства мы можем вычислить множество достижимых вершин для любой вершины путем слияния множеств достижимых вершин, ассоциированных с вершинами назначения каждого ребра. Завершает это слияние




```
 5:  0 0 0 0 0 1 0 0 0 0 0 0 0
12:  0 0 0 0 0 0 0 0 0 0 0 0 1
11:  0 0 0 0 0 0 0 0 0 0 0 1 1
10:  0 0 0 0 0 0 0 0 0 0 1 0 0
 9:  0 0 0 0 0 0 0 0 0 1 1 1 1
 4:  0 0 0 0 1 0 0 0 0 1 1 1 1
 6:  0 0 0 0 1 0 1 0 0 1 1 1 1
 3:  0 0 0 1 1 1 0 0 0 1 1 1 1
 2:  0 0 1 1 1 1 0 0 0 1 1 1 1
 1:  0 1 0 0 0 0 0 0 0 0 0 0 0
 0:  1 1 1 1 1 1 1 0 0 1 1 1 1
 7:  0 0 0 0 1 0 1 1 0 1 1 1 1
 8:  0 0 0 0 1 0 1 1 1 1 1 1 1
```

РИСУНОК 19.27. ТРАНЗИТИВНОЕ ЗАМЫКАНИЕ ГРАФА DAG

*Рассматриваемая последовательность векторов-строк есть транзитивное замыкание графа DAG, показанного на рис. 19.21. Эти строки были построены во время выполнения обратной топологической сортировки, вычисленной как завершающее действие рекурсивной функции DFS (см. программу 19.9). Каждая строка является результатом логической операции **or** (или) над строками смежных вершин, которые заранее появились в рассматриваемом списке в результате предшествовавших вычислений. Например, чтобы вычислить строку для вершины **0**, мы выполняем логическую **or** над строками для вершин **5**, **2**, **1** и **6** (и подстановку **1** вместо самой вершины **0**), поскольку ребра **0-5**, **0-2**, **0-1** и **0-6** приводят нас из вершины **0** в любую вершину, которая достижима их каждой из этих вершин. Мы можем игнорировать прямые ребра, поскольку они не добавляют новой информации. Например, мы игнорируем ребро, ведущее из **0** в **3**, поскольку вершины, достижимые из **3** уже фигурируют в строке, соответствующей **2**.*

выполнение операции логического **or** над конкретными строками матрицы смежности. Мы осуществляем доступ к строке размера V каждого древесного ребра и каждого поперечного ребра. В рассматриваемом случае обратные ребра отсутствуют, и мы можем игнорировать прямые ребра, поскольку мы определили все вершины, в которые они приводят, когда проводили обработку предшественников обоих вершин этих ребер на более ранних стадиях поиска. ■

Программа 19.9. Транзитивное замыкание графа DAG

Конструктор в этом классе вычисляет транзитивное замыкание DAG при помощи одного поиска в глубину. Он вычисляет в рекурсивном режиме вершины, достижимые из каждой вершины из числа достижимых вершин ее потомков в дереве DFS.

```
template <class tcDag, class Dag> class dagTC
{ tcDag T; const Dag &D;
   int cnt;
   vector<int> pre;
   void tcR(int w)
   {
     pre[w] = cnt++;
     typename Dag::adjIterator A(D, w);
     for (int t = A.beg(); !A.end(); t = A.nxt())
      {
         T.insert(Edge(w, t));
         if (pre[t] > pre[w]) continue;
         if (pre[t] == -1) tcR(t);
         for (int i = 0; i < T.V(); i++)
           if (T.edge(t, i)) T.insert(Edge(w, i));
      }
   }
public:
   dagTC(const Dag &D) : D(D), cnt(0),
     pre(D.V(), -1), T(D.V(), true)
     { for (int v = 0; v < D.V(); v++)
         if (pre[v] == -1) tcR(v); }
   bool reachable(int v, int w) const
     { return T.edge(v, w); }
};
```

Если рассматриваемый DAG не имеет прямых ребер (см. упражнение 19.42), время выполнения программы 19.9 пропорционально VX, и в этом аспекте она никак не превосходит алгоритмы транзитивного замыкания, которые мы анализировали применительно к орграфам общего вида в разделе 19.3 (как, например, программа 19.4), равно как и подход, основанный на применении топологической сортировки, описание которой изложено в начале данного раздела. С другой стороны, если число прямых вершин велико (или, что одно и то же, число поперечных ребер мало), то программа 19.9 проявляет намного большее быстродействие, чем указанные выше методы.

Задача поиска оптимального алгоритма (такого, чтобы гарантировал решение задачи за время, пропорциональное V^2) вычисления транзитивного замыкания насыщенного DAG все еще не решена. Широко известная граница производительности для худшего случая достигает уровня VE. В то же время нас больше устраивает алгоритм, который выполняется быстрее на некотором обширном классе графов DAG, такой как, например, алгоритм, реализованный программой 19.9, чем алгоритм, который, подобно программе

19.4, в любом случае выполняется за время, пропорциональное VE. В разделе 19.9 мы убедимся, что подобного рода увеличение производительности для графов DAG оказывает также непосредственное влияние на нашу способность вычислять транзитивные замыкания орграфов общего вида.

Упражнения

- **19.115**. Покажите в стиле рис. 19.27, какой вид принимают векторы достижимости, когда для вычисления транзитивного замыкания графа DAG

  3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4 4-3 2-3

  мы используем программу 19.9.

- **19.116**. Разработайте такую версию программы 19.9, которая использовала бы представление транзитивного замыкания, которая не поддерживает функцию проверки ребра **edge** и которая выполняется за время, пропорциональное $V^2 + \sum_e v(e)$; здесь суммирование производится по всем ребрам графа DAG, а $v(e)$ есть число вершин, достижимых из вершины назначения ребра e. Такие затраты будут значительно ниже, чем VE, для некоторых видов разреженных графов DAG (см. упражнение 19.65).

- **19.117**. Постройте программную реализацию класса, вычисляющего абстрактное транзитивное замыкание графов DAG, который использует дополнительное пространство памяти максимум пропорциональное V (и пригодный для работы с крупными графами DAG). Воспользуйтесь топологической сортировкой для выдачи быстрого ответа в случае, когда вершины не связны, а также реализацией очереди истоков с тем, чтобы получить длину пути, когда вершины связны.

- **19.118**. Разработайте реализацию транзитивного замыкания, основанную на применении обратной топологической сортировки на базе очереди стоков (см. упражнение 19.105).

- **19.119**. Требует ли найденное вами решение задачи 19.118 проверки всех ребер DAG или есть ребра, которые можно проигнорировать, как, например, прямые ребра в поиске в ширину? Дайте пример, когда требуется анализ всех ребер, или опишите ребра, которые можно пропустить.

19.8. Сильные компоненты в орграфах

Неориентированные графы и графы DAG суть более простые структуры данных, чем орграфы общего типа в силу структурной симметрии, которая характеризует отношения достижимости среди вершин: из того факта, что в неориентированном графе существует путь из s в t, мы знаем, что существует путь из t в s; если в графе DAG существует путь из s в t, то мы знаем, что путь из t в s *не существует*. В случае орграфов общего вида из того факта, что вершина t доступна из s, отнюдь не означает, что s доступна из t.

Чтобы изучить структуру орграфов, рассмотрим *сильную связность (strong connectivity)*, обладающую интересующей нас симметрией. Если s и t являются сильно связными вершинами (каждая из них достижима из другой), то, по определению, таковыми являются и t и s. Как уже отмечалось в разделе 19.1, из такой симметрии следует, что вершины орграфа разбиваются на классы сильных компонент, состоящие из взаимно достижимых компонент. В этом разделе мы рассмотрим три алгоритма поиска сильных компонент в орграфах.

В алгоритмах поиска на графах общего вида применительно к неориентированным графам мы используем тот же интерфейс, что и для случая связности (см. программу 18.4). Назначение наших алгоритмов состоит в том, чтобы присвоить номер компоненты каждой вершине вектора, индексированного по именам вершин, используя с этой целью метки 0, 1, ... для обозначения сильных компонент. Наибольший из присваиваемых номеров есть номер, который на единицу меньше количества сильных компонент, а мы можем использовать номер компоненты для проведения проверки, входят ли две заданные вершины в одну и ту же сильную компоненту, за постоянное время.

Алгоритм решения этой задачи "в лоб" построить нетрудно. Используя АТД абстрактного транзитивного замыкания, проанализируйте каждую пару вершин s и t с тем, чтобы проверить, достижима ли вершина t из s и достижима ли s из t. Определите неориентированный граф, в котором содержатся ребра для каждой такой пары: связные компоненты этого графа суть сильные компоненты орграфа. Этот алгоритм нетрудно описать и реализовать, а время его выполнения в основном затрачивается на реализацию абстрактно-транзитивного замыкания таким, каким оно описано, скажем, свойством 19.10.

Алгоритмы, которые мы рассмотрим в данном разделе, являются воплощением последних достижений в области построения алгоритмов, они способны обнаружить сильные компоненты любого графа за *линейное* время, т.е. в V раз быстрее, чем алгоритм решения "в лоб". Для графов, содержащих 100 вершин быстродействие этих алгоритмов в 100 раз быстрее, чем алгоритм решения этой задачи "в лоб"; для графа с 1000 вершин они работают в 1000 раз быстрее; и у нас появляется возможность решать задачи такого рода для графов с миллионами вершин. Эта задача является красноречивым примером возможностей хорошего алгоритма, она стала мощным побудительным мотивом на проведение исследований в области алгоритмов на графах. В каких других областях мы можем рассчитывать на сокращение используемых ресурсов в миллион и более раз за счет выбора элегантного алгоритма решения практически важной задачи?

История этой задачи сама по себе достаточно поучительна (*см. раздел ссылок*). В пятидесятые и шестидесятые годы математики и специалисты по вычислительной технике приступили к серьезному изучению алгоритмов на графах в контексте, в котором анализ алгоритмов сам развивался как отдельная область исследований. В условиях широкого разнообразия алгоритмов на графах, требовавшего исследований на фоне непрерывного и стремительного развития компьютерных систем, языков программирования и нашего понимания того, что означает выполнить эффективные вычисления, многие задачи оставались нерешенными. По мере того, как теоретики вычислительных систем стали постигать многие из базовых принципов анализа алгоритмов, они стали понимать, какие задачи на графах могут быть решены эффективно и для каких задач это невозможно, и приступили к разработке алгоритмов решения прежнего набора задач и к повышению их эффективности. И действительно, Р. Тарьян (R. Tarjan) предложил линейные по времени выполнения алгоритмы решения задачи сильной связности и других задач на графах в 1972 году, в том самом, когда Р. Карп (R. Karp) доказал невозможность эффективного решения задачи коммивояжера и многих других задач на графах. Алгоритм Тарьяна оставался главным направлением анализа алгоритмов в течение многих лет, поскольку он решает важную практическую задачу, используя для этой цели простые структуры данных. В восьмидесятых годах Р. Косарайю (R. Kosaraju) предложил оригинальный подход к ре-

шению этой задачи и разработал ее новое решение; несколько позднее было установлено, что статья, в которой был описан этот же метод, была опубликована в научной печати в России намного раньше, а именно, в 1972 г. Позже, в 1999 г., Г. Габову (H. Gabov) удалось получить простую реализацию одного из первых подходов, предложенных в шестидесятых года, что дает третье линейное по времени решение этой задачи.

Суть сказанного выше состоит не только в том, что трудные задачи обработки графов могут иметь простые решения, но и в том, что абстракции, которыми мы пользуемся (поиск в глубину и списки смежных вершин) таят в себе гораздо большие возможности, чем мы можем предполагать. По мере того как мы освоим эти и подобные им инструментальные средства, нас не должны также удивлять случаи обнаружения простых решений других важных задач на графах. Исследователи продолжают поиски компактных реализаций, подобных рассматриваемым выше, других многочисленных важных алгоритмов на графах; многие из таких алгоритмов еще предстоит открыть.

Метод Косарайю прост для понимания и реализации. Чтобы найти сильные компоненты заданного графа, сначала выполняется поиск в глубину в обратном порядке, что означает вычисление различных перестановок вершин, определенных посредством нумерации при обходе в обратном порядке. (Такой процесс представляет собой топологическую сортировку, если орграф есть DAG.) Затем производится прогон DFS на этом графе, но на этот раз с целью обнаружить следующую вершину для поиска (при вызове рекурсивной функции поиска в начале прогона и каждый раз, когда рекурсивная функция поиска возвращает результат в функцию поиска более высокого уровня) *используется непосещенная вершина с максимальным номером в обратном порядке.*

Привлекательность такого алгоритма заключается в том, что когда проверка непосещенных вершин производится в соответствии с топологической сортировкой таким образом, деревья в лесе DFS определяют сильные компоненты так же, как деревья в лесе DFS определяют связные компоненты в неориентированных графах – две вершины содержатся в одной и той же сильной компоненте тогда и только тогда, когда они принадлежат одному и тому же дереву в этом лесе. Рисунок 19.28 служит иллюстрацией этого факта в условиях рассматриваемого примера, что мы и докажем немного позже. В силу этого обстоятельства мы можем обозначать компоненты номерами, как это делалось в случае неориентированных графов, увеличивая номер компоненты на единицу всякий раз, когда рекурсивная функция возвращает результат в функцию поиска более высокого уровня. Программа 19.10 предлагает полную реализацию этого метода.

Программа 19.10. Сильные компоненты (алгоритм Косарайю)

Клиенты могут использовать объекты этого класса для определения числа сильных компонент орграфа (**count**) и выполнять проверку на принадлежность сильной компоненте (**strongly connected**). Конструктор **SC** сначала строит обратный орграф и выполняет поиск в глубину с целью вычисления нумерации при обходе в обратном порядке. Далее он выполняет поиск в глубину на исходном орграфе, используя для этой цели обращение обратного порядка, полученного в результате выполнения первого поиска в глубину в цикле поиска, в котором производятся вызовы этой рекурсивной функции. Каждый рекурсивный вызов в рамках второго поиска в глубину посещает все вершины сильной компоненты.

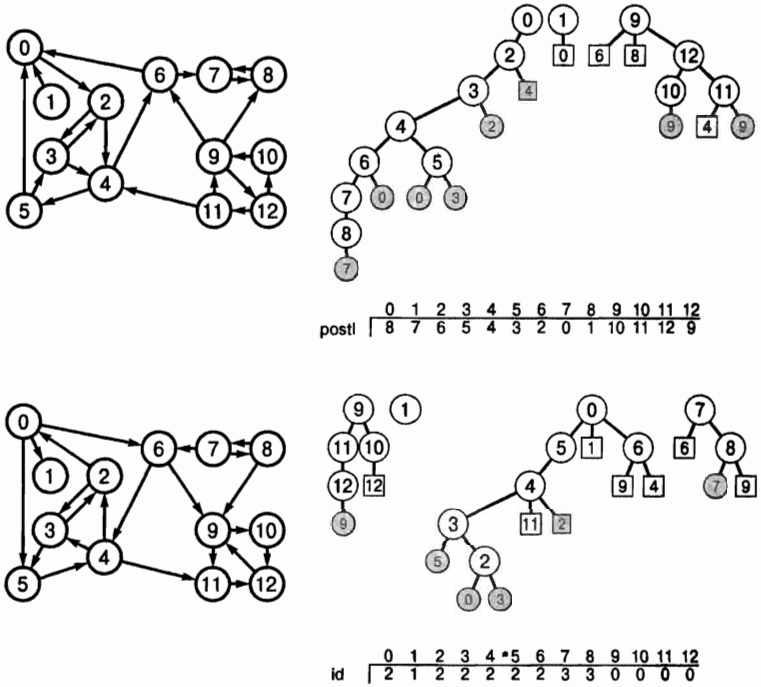


РИСУНОК 19.28. ВЫЧИСЛЕНИЕ СИЛЬНЫХ КОМПОНЕНТ (АЛГОРИТМ КОСАРАЙЮ)

Чтобы вычислить сильные компоненты орграфа, изображенного внизу слева, мы сначала выполняем поиск в глубину на его обращении (вверху слева), вычисляя вектор обратного порядка обхода, который присваивает вершинам индексы в порядке, в каком завершаются рекурсивные DFS (сверху). Этот порядок эквивалентен обратному порядку обхода леса DFS (вверху справа). Затем мы используем обращение этого порядка, чтобы выполнить поиск в глубину на исходном графе (внизу). Сначала мы проверяем все узлы, доступные из вершины 9, затем осуществляем просмотр этого вектора справа налево, обнаруживая при этом, что 1 есть крайняя справа непосещенная вершина, поэтому мы выполняем рекурсивный вызов для вершины 1 и т.д. Деревья в лесе DFS, которые выбираются в результате этого процесса, представляют собой сильные компоненты: все вершины каждого дерева имеют те же значения, что и вектор id, индексированный именами вершин (внизу).

```
template <class Graph> class SC
{ const Graph &G;
  int cnt, scnt;
  vector<int> postI, postR, id;
  void dfsR(const Graph &G, int w)
  {
    id[w] = scnt;
    typename Graph::adjIterator A(G, w);
    for (int t = A.beg(); !A.end(); t = A.nxt())
      if (id[t] == -1) dfsR(G, t);
    postI[cnt++] = w;
  }
public:
  SC(const Graph &G) : G(G), cnt(0), scnt(0),
    postI(G.V()), postR(G.V()), id(G.V(), -1)
    { Graph R(G.V(), true);
      reverse(G, R);
      for (int v = 0; v < R.V(); v++)
        if (id[v] == -1) dfsR(R, v);
      postR = postI; cnt = scnt = 0;
      id.assign(G.V(), -1);
      for (int v = G.V()-1; v >= 0; v--)
        if (id[postR[v]] == -1)
          { dfsR(G, postR[v]); scnt++; }
    }
  int count() const { return scnt; }
  bool stronglyreachable(int v, int w) const
    { return id[v] == id[w]; }
};
```

Свойство 19.14. *Метод Косарайю обеспечивает обнаружение сильных компонент графа ценой линейных затрат времени и пространства памяти.*

Доказательство: Этот метод состоит из двух процедур поиска в глубину, подвергнутых незначительным изменениям, благодаря чему время его выполнения, как обычно, пропорционально V^2 в случае насыщенных графов и $V + E$ в случае разреженных графов (если графы представлены в виде списков смежных вершин). Чтобы проверить, что он правильно вычисляет сильные компоненты, мы должны доказать на втором просмотре, что две вершины s и t содержатся в одном и том же дереве леса DFS тогда и только тогда, когда они взаимно достижимы.

Если вершины s и t взаимно достижимы, они обязательно будут находиться в одном и том же дереве DFS, поскольку, когда просматривается первая из них, вторая остается непосещенной и достижимой из первой и непременно будет просмотрена, прежде чем завершится рекурсивный вызов из корня.

Чтобы доказать противное, предположим, что s и t включены в одно и то же дерево, и пусть r есть корень этого дерева. Из того факта, что s достижима из r (через ориентированный путь, состоящий из ребер дерева), следует, что существует ориентированный путь из s в r в обратном орграфе. Теперь остается доказать, что должен существовать путь из r в s в обратном орграфе, поскольку r имеет более высокий номер в обратном порядке обхода, чем s (из-за того, что при втором поиске в глубину r была выбрана первой в то время, когда обе вершины еще не посещались), и существует путь из s в r. Если бы пути из s в r не было, то путь из s в r в обратном орграфе остав-

лял бы s с более высоким номером в результате обратного обхода. В силу сказанного выше, существуют ориентированные пути из s в r и из r в s в как в орграфе, так и в его обращении, отсюда следует, что s и r сильно связаны. Те же рассуждения доказывают, что t и r сильно связаны, и в силу этого обстоятельства s и t также сильно связаны. ■

Реализация алгоритма Косарайю для орграфа, представленного в виде матрицы смежности, даже проще, чем программа 19.10, поскольку в этом случае не нужно вычислять обратный граф в явном виде; решение этой задачи мы оставляем читателю на самостоятельную проработку (см. упражнение 19.125).

Программа 19.11. Сильные компоненты (алгоритм Тарьяна)

Класс DFS представляет собой еще одну реализацию того же интерфейса, что применялся в программе 19.10. Он использует стек **S** для хранения каждой вершины до тех пор, пока не выяснится, что все вершины в верхней части стека до определенного уровня принадлежат одной и той же сильной компоненте. Вектор **low**, индексированный именами вершин, отслеживает вершину с наименьшим номером в прямом порядке обхода, достижимую из каждого узла через некоторую последовательность прямых связей, за которыми следует одна восходящая связь (*см. рассуждения по тексту раздела*).

```
#include "STACK.cc"
template <class Graph> class SC
{ const Graph &G;
  STACK<int> S;
  int cnt, scnt;
  vector<int> pre, low, id;
  void scR(int w)
  { int t;
    int min = low[w] = pre[w] = cnt++;
    S.push(w);
    typename Graph::adjIterator A(G, w);
    for (t = A.beg(); !A.end(); t = A.nxt())
     {
       if (pre[t] == -1) scR(t);
       if (low[t] < min) min = low[t];
     }
    if (min < low[w]) { low[w] = min; return; }
    do
     { id[t = S.pop()] = scnt; low[t] = G.V(); }
    while (t != w);
    scnt++;
  }
public:
  SC(const Graph &G) : G(G), cnt(0), scnt(0),
    pre(G.V(), -1), low(G.V()), id(G.V())
    { for (int v = 0; v < G.V(); v++)
        if (pre[v] == -1) scR(v); }
  int count() const { return scnt; }
  bool stronglyreachable(int v, int w) const
    { return id[v] == id[w]; }
};
```

Программа 19.10 представляет собой оптимальное решение задачи сильной связности, аналогичную решениям задачи связности, рассмотренным в главе 18. В разделе 19.9 мы проведем исследование расширения этого решения на вычисление задачи транзитивного замыкания и решим задачу достижимости (абстрактно-транзитивного замыкания) в орграфах.

Сначала, однако, мы рассмотрим алгоритм Тарьяна и алгоритм Габова – оригинальные методы, требующие внесения всего лишь небольших исправлений в нашу базовую процедуру поиска в глубину. Они предпочтительнее, чем алгоритм Косарайю, поскольку используют только один проход по графу и в силу того, что не требуют обращения для разреженных графов.

Алгоритм Тарьяна – это аналог программы, которую мы изучали в главе 17 с целью обнаружения мостов в неориентированных графах (см. программу 18.7). Этот метод основан на двух наблюдениях, которые мы сделали в других контекстах. Во-первых, мы рассматриваем вершины в обратном топологическом порядке, поэтому, когда мы достигнем конца рекурсивной функции для вершины, которую знаем, мы не встретим ни одной вершины из той же сильной компоненты (в силу того, что все вершины, достижимые из этой вершины, уже подверглись обработке). Во-вторых, обратные связи в дереве обеспечивают второй путь из одной вершины в другую и связывают между собой сильные компоненты.

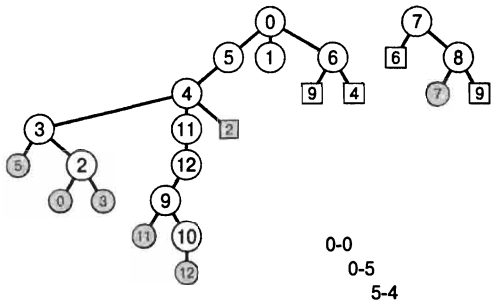
Рекурсивная функция DFS использует те же вычисления, что и программа 18.7, с целью определения достижимой вершины с максимальным номером (через обратную связь) из любого потомка в каждой вершине. Она использует также вектор, индексированный именами вершин, для хранения сильных компонент и стек для отслеживания текущего пути поиска. Она заталкивает имена вершин в стек на входе в рекурсивную функцию, затем выталкивает их из стека и назначает номера компонентам после посещения завершающего элемента каждой сильной компоненты. Этот алгоритм построен на нашей способности фиксировать этот момент при помощи простой проверки (в основе которой лежит отслеживание предка с максимальным номером, достижимого через одну восходящую связь из всех потомков каждого узла) в конце рекурсивной процедуры, которая сообщает, что все вершины, встреченные с момента вхождения (за исключением тех, которые уже были назначены компоненте) принадлежат той же сильной компоненте.

Реализация в виде программы 19.11 представляет собой полное описание рассматриваемого алгоритма в сжатой форме, содержащее все необходимые подробности, которых не хватает в данном выше общем описании. Рисунок 19.29 служит иллюстрацией работы рассматриваемого алгоритма на демонстрационном примере орграфа, показанного на рис. 19.1.

Свойство 19.15. *Алгоритм Тарьяна находит сильные компоненты орграфа за линейное время.*

Набросок доказательства: Если у вершины s нет потомков или восходящих связей в дереве DFS, либо если у нее есть потомок в дереве DFS с восходящей связью, которая указывает на s, и нет потомков с восходящими связями, которые направлены в верхнюю часть дерева, тогда она и все ее потомки (за исключением тех вершин, которые удовлетворяют тому же свойству, и их потомки) составляют сильную компоненту. Чтобы установить этот факт, заметим, что каждый потомок t вершины s, который не удовлетворяет сформулированному свойству, имеет потомка, обладающего восходящей связью, указывающей на вершину, которая в дереве выше t. Существует путь из s в t вниз по дереву, а мы можем найти путь из t в s следующим образом: спустимся вниз по дереву из t в вершину с восходящей связью, которая ведет в вершину, расположенную выше t, затем продолжаем тот же процесс из этой вершины, пока не достигнем s.

Как обычно, этот метод линеен по времени, поскольку в данном случае к стандартному DFS добавляется несколько операций, выполняющихся за постоянное время. ■



	0	1	2	3	4	5	6	7	8	9	10	11	12
pre	0	9	4	3	2	1	10	11	12	7	8	5	6
low	0	9	0	0	0	0	0	11	11	5	6	5	5
id	2	1	2	2	2	2	2	3	3	0	0	0	0

0-0	0		0
0-5	0 5		0 5
5-4	0 5 4		0 5 4
4-3	0 5 4 3		0 5 4 3
3-5	0 5 4 3		0 5
3-2	0 5 4 3 2		0 5 2
2-0	0 5 4 3 2		0
2-3	0 5 4 3 2		0
4-11	0 5 4 3 2 11		0 11
11-12	0 5 4 3 2 11 12		0 11 12
12-9	0 5 4 3 2 11 12 9		0 11 12 9
9-11	0 5 4 3 2 11 12 9		0 11
9-10	0 5 4 3 2 11 12 9 10		0 11 10
10-12	0 5 4 3 2 11 12 9 1 0		0 11
11	0 5 4 3 2		0
4-2	0 5 4 3 2		0
0-1	0 5 4 3 2 1		0 1
1	0 5 4 3 2		0
0-6	0 5 4 3 2 6		0 6
6-9	0 5 4 3 2 6		0 9
6-4	0 5 4 3 2 6		0
0			
7-7	7		7
7-6	7		7
7-8	7 8		7 8
8-7	7 8		7
8-9	7 8		7
7			

РИСУНОК 19.29. ВЫЧИСЛЕНИЕ СИЛЬНЫХ КОМПОНЕНТ (АЛГОРИТМЫ ТАРЬЯНА И ГАБОВА)

В основе алгоритма Тарьяна лежит рекурсивный поиск в глубину, в который добавлена операция проталкивания вершин в стек. Он вычисляет индекс компоненты для каждой вершины в векторе ***id****, индексированном именами вершин, используя векторы* ***pre*** *и* ***low*** *(в центре). Дерево DFS (дерево поиска в глубину) для рассматриваемого примера графа показано в верхней части рисунка. Трассировка ребер показана внизу слева. В центре нижней части находится содержимое главного стека: мы заталкиваем в него вершины, достижимые через древесные ребра. Воспользовавшись поиском в глубину с тем, чтобы рассматривать вершины в обратном топологическом порядке, мы вычисляем для каждой вершины* ***v*** *максимальную точку, достижимую через обратную связь из предшественника (****low[v]****). Когда для вершины* ***v*** *выполняется* ***pre[v]= low[v]*** *(в данном случае это вершины* ***11****,* ***1****,* ***0*** *и* ***7****), мы выталкиваем ее из стека, а также все вершины выше ее и всем им присваиваем номер следующей компоненты.*

В алгоритме Габова мы снова заталкиваем вершины в главный стек точно так же, как и в алгоритме Тарьяна, но в то же время мы поддерживаем и второй стек (внизу справа), в который заталкиваются вершины, лежащие на пути поиска (о последних известно, что они находятся в различных сильных компонентах), и выталкиваем все вершины после достижения каждого обратного ребра. Если мы завершаем обработку вершины v*, когда* v *находится на верхушке второго стека (заштриховано), мы знаем, что все вершины, расположенные над* v *в главном стеке, находятся в одной и той же сильной компоненте.*

В 1999 году Габов обнаружил в программе 19.12 версию алгоритма Тарьяна. Этот алгоритм поддерживает такой же стек вершин и тем же способом, что и алгоритм Тарьяна, но в то же время он использует второй стек (вместо вектора, индексированного именами вершин, в котором хранились номера вершин при обходе в прямом порядке) с тем, чтобы знать, когда выталкивать из главного стека все вершины каждой сильной компоненты. Во втором стеке содержатся вершины, входящие в траекторию поиска. Когда обратное ребро показывает, что некоторая последовательность таких вершин целиком принадлежит одной и той же сильной компоненте, мы освобождаем этот стек, оставляя в нем только вершину назначения обратного ребра, которая ближе к корню дерева, чем любая другая вершина. По завершении обработки всех ребер каждой вершины (при этом выполняются рекурсивные вызовы всех ребер дерева, освобождается стек пути для обратных ребер и игнорируются прямые ребра), мы проверяем, находится ли текущая вершина в верхушке стека пути. Если она там, то она и все вершины, расположенные сверху нее в главном стеке, составляют сильную компоненту; мы выталкиваем их из стека и присваиваем им номер следующей сильной компоненты, как это делалось в алгоритме Тарьяна.

Пример на рис. 19.29 показывает содержимое этого второго стека. Следовательно, данная диаграмма может служить иллюстрацией работы алгоритма Габова.

Свойство 19.16. *Алгоритм Габова находит сильные компоненты орграфа за линейное время.*

Формализацию только что изложенных аргументов и доказательство отношений, связывающих содержимое стеков, на котором оно основано, мы оставляем на самостоятельную проработку (см. упражнение 19.132) для читателей, имеющих склонность к математике. Этот метод также линеен по времени, поскольку в данном случае к стандартному поиску в глубину добавляется несколько операций, выполняющихся за постоянное время. ■

Программа 19.12. Сильные компоненты (алгоритм Габова)

Данная альтернативная реализация рекурсивной функции-элемента из программы 19.11 использует второй стек **path** вместо вектора **low**, индексированного номерами вершин, чтобы принимать решение относительно того, когда выталкивать вершины каждой сильной компоненты из главного стека (см. текст).

```
void scR(int w)
{ int v;
  pre[w] = cnt++;
  S.push(w); path.push(w);
  typename Graph::adjIterator A(G, w);
  for (int t = A.beg(); !A.end(); t = A.nxt())
    if (pre[t] == -1) scR(t);
    else if (id[t] == -1)
      while (pre[path.top()] > pre[t]) path.pop();
  if (path.top() == w) path.pop(); else return;
  do { id[v = S.pop()] = scnt; } while (v != w);
  scnt++;
}
```

Все алгоритмы выявления сильных компонент, которые мы рассмотрели в этом разделе, оригинальны и обманчиво просты. Мы рассмотрели все три алгоритма, поскольку они демонстрируют богатые возможности фундаментальных структур данных и представ-

лены искусно разработанными рекурсивными программами. С практической точки зрения время выполнения всех этих алгоритмов пропорционально числу ребер орграфа, и различие в производительности, по-видимому, зависит от деталей реализации. Например, операции АТД над стеком магазинного типа составляют внутренний цикл алгоритмов Тарьяна и Габова. В нашей реализации используется реализация класса скелетного стека из главы 4; реализации, использующие класс **stack** из библиотеки STL (Standard Template Library — стандартная библиотека шаблонов), которые производят проверку на наличие ошибок и требуют других непроизводительных расходов, могут обладать меньшим быстродействием. Реализация алгоритма Косарайю – это, возможно, простейшая из всех трех, тем не менее, и для нее характерен небольшой недостаток (для разреженных графов), поскольку он требует выполнения трех проходов для просмотра ребер (один проход для построения обратного графа и два прохода поиска в глубину).

Далее мы рассмотрим ключевое приложение, вычисляющее сильные компоненты: построение АТД эффективной достижимости (абстрактно-транзитивное замыкание) в орграфах.

Упражнения

▷ **19.120**. Опишите, что произойдет, когда вы воспользуетесь алгоритмом Косарайю, чтобы найти сильные компоненты графа DAG.

▷ **19.121**. Опишите, что произойдет, когда вы воспользуетесь алгоритмом Косарайю, чтобы найти сильные компоненты орграфа, который состоит из одного цикла.

●● **19.122**. Можно ли избежать вычислений обращения орграфа в версии метода Косарайю, ориентированной на представление графов в виде списка смежных вершин (программа 19.10), за счет использования одной из трех технологий, отмеченных в разделе 19.4, с целью избежать вычисления обращения при выполнении топологической сортировки? Для каждой технологии дайте либо доказательство того, что она работает, либо контрпример, показывающий, что она не работает.

○ **19.123**. Покажите в стиле рис. 19.28 леса DFS и содержимое вспомогательных векторов, индексированных именами вершин, которые получаются, когда вы используете алгоритм Косарайю для вычисления сильных компонент обращения орграфа, представленного на рис. 19.5. (Вы должны получить те же сильные компоненты.)

19.124. Покажите в стиле рис. 19.28 леса DFS и содержимое вспомогательных векторов, индексированных именами вершин, которые получаются, когда вы используете алгоритм Косарайю для вычисления сильных компонент орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

○ **19.125**. Реализуйте алгоритм Косарайю с целью определения сильных компонент орграфа в представлении, которое поддерживает проверку **edge** существования ребра. Явное вычисление обращения графа исключается. *Указание*: Рассмотрите возможность использования двух различных рекурсивных функций DFS.

▷ **19.126**. Опишите, что произойдет, когда вы воспользуетесь алгоритмом Тарьяна, чтобы найти сильные компоненты графа DAG.

▷ **19.127**. Опишите, что произойдет, когда вы воспользуетесь алгоритмом Тарьяна, чтобы найти сильные компоненты орграфа, который состоит из одного цикла.

○ **19.128**. Покажите в стиле рис. 19.28 леса DFS, содержимое стека во время выполнения алгоритма и завершающее состояние вспомогательных векторов, индексированных именами вершин, которые получаются, когда используется алгоритм Тарьяна для вычисления сильных компонент обращения орграфа, представленного на рис. 19.5. (Вы должны получить те же сильные компоненты.)

19.129. Покажите в стиле рис. 19.29 леса DFS, содержимое стека во время выполнения алгоритма и окончательное содержимое векторов, индексированных именами вершин, которые получаются, когда вы используете алгоритм Тарьяна для вычисления сильных компонент орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

○ **19.130**. Внесите такие изменения в реализацию алгоритма Тарьяна в программе 19.11 и алгоритма Габова в программе 19.12, чтобы они могли использовать сигнальные значения с тем, чтобы избежать необходимости явной проверки поперечных связей.

19.131. Покажите в стиле рис. 19.29 леса DFS, содержимое обоих стеков во время выполнения алгоритма и окончательное содержимое вспомогательных векторов, индексированных именами вершин, которые получаются, когда вы используете алгоритм Габова для вычисления сильных компонент орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

• **19.132**. Дайте полное доказательство свойства 19.16.

○ **19.133**. Разработайте версию алгоритма Габова, который находит мосты и реберно-связные компоненты в неориентированных графах.

• **19.134**. Разработайте версию алгоритма Габова, который находит точки сочленения и двусвязные компоненты в неориентированных графах.

19.135. Разработайте таблицу в духе табл. 18.1 с целью изучения сильных компонент в случайных графах (см. табл. 19.2). Пусть S есть множество набор в наибольшей сильной компоненте. Следите за изменением размера набора S проведите анализ процентного отношения ребер следующих четырех классов: те, которые соединяют две вершины из S; те, которые указывают на вершины, не входящие в S; те, которые указывают на вершины, входящие в S, и соединяют две вершины, не входящие в S.

19.136. Проведите эмпирические исследования с тем, чтобы сравнить метод решения "в лоб" задачи вычисления сильных компонент, описанный в начале этого раздела, с алгоритмом Косарайю, с алгоритмом Тарьяна и с алгоритмом Габова для различных типов орграфов.

••• **19.137**. Разработайте линейный по времени алгоритм для *сильной двусвязности* (strong 2-connectivity): Определите, обладает ли сильно связанный орграф тем свойством, что он остается сильно связным после удаления любой из вершин (и всех инцидентных ей ребер).

19.9. Еще раз о транзитивном замыкании

Объединяя результаты двух предыдущих разделов, мы можем разработать алгоритм решения задачи вычисления абстрактно-транзитивного замыкания для орграфов, который хотя и не вводит никаких усовершенствований в решение для худшего случая, основанное на поиске в глубину, но в то же время обеспечивает оптимальное решение во многих других случаях.

В основе этого алгоритма лежит предварительная обработка орграфа, назначение которой состоит в построении базового графа DAG рассматриваемого орграфа (см. свойство 19.2). Алгоритм эффективен в том случае, когда размер базового графа DAG меньше размеров исходного орграфа. Если исходный граф сам является графом DAG (и в силу этого обстоятельства идентичен базовому DAG), или если он содержит всего лишь несколько циклов, мы не можем рассчитывать на сколь либо заметную экономию затрат. В то же время, если орграф содержит большие циклы или крупные сильные компоненты (и поэтому содержит базовые DAG небольших размеров), мы можем построить оптимальные либо близкие к оптимальным алгоритмы. Для ясности предположим, что базовый DAG достаточно мал, благодаря чему мы можем воспользоваться представлением графа в виде матрицы смежности, хотя основная идея сохраняет свою актуальность и для базовых DAG больших размеров.

Для реализации абстрактного транзитивного замыкания мы выполним предварительную обработку графа в следующем объеме:

- Найдем его сильные компоненты.
- Построим его базовый DAG.
- Вычислим транзитивное замыкание базового DAG.

Мы можем воспользоваться алгоритмом Косарайю, алгоритмом Тарьяна либо алгоритмом Габова, чтобы найти сильные компоненты; выполнить один просмотр ребер с целью построения базового DAG (описание будет дано в следующем параграфе); и провести поиск в глубину (программа 19.9) с целью вычисления его транзитивного замыкания. По завершении этой предварительной обработки мы сразу можем обращаться к информации, необходимой для определения достижимости.

Если в нашем распоряжении имеется вектор, индексированный именами вершин с сильными компонентами орграфа, построение представления его базового DAG в виде матрицы смежности не представляет трудностей. Вершины графа DAG суть номера компонент орграфа. Для каждого ребра *s-t* исходного графа мы просто устанавливаем **D->adj[sc[s]][sc[t]]** в **1**. Нам пришлось бы решать проблему дубликатов ребер в базовом DAG, если бы мы воспользовались представлением графа в виде списков смежных вершин, в то же время в матрице смежности дубликаты ребер просто соответствуют установке элемента матрицы в **1**, который уже имел значение **1**. Это небольшой нюанс приобретает важное значение, поскольку потенциально в рассматриваемом приложении число дубликатов ребер огромно (по отношению к размеру базового DAG).

Свойство 19.17. *Заданы две вершины s и t в орграфе D, пусть sc(s) и sc(t) – соответствующие им вершины в базовом DAG K орграфа D. Тогда t достижима из s в D тогда и только тогда, когда sc(t) достижима из sc(s) в K.*

Этот простой факт следует из определений. В частности, это свойство предполагает, что действует соглашение, согласно которому вершина достижима сама из себя (все вершины имеют петли). Если эти вершины находятся в одной и той же сильной компоненте ($sc(s) = sc(t)$), то они взаимно достижимы. ■

Мы определим, достижима ли вершина t из заданной вершины s тем же способом, каким мы строили базовый DAG: воспользуемся вектором, индексированным именами вершин, вычисленным с помощью алгоритма сильных компонент, чтобы получить числа $sc(s)$ и $sc(t)$ (за постоянное время), которые мы будем рассматривать как индексы абстрактных вершин (abstract vertex) базового DAG. Их использование в качестве индексов для транзитивного замыкания базового DAG даст искомый результат.

Программа 19.13. Транзитивное замыкание на основе сильных компонент

Данный класс реализует интерфейс (абстрактного) транзитивного замыкания орграфов путем вычисления сильных компонентов (с использованием, скажем, программы 19.11), базового DAG и транзитивного замыкания базового DAG (с использованием программы 19.9). Он предполагает, что класс **SC** обладает общедоступной функцией-элементом **ID**, которая возвращает индекс сильной компоненты (из массива **id**) для любой заданной вершины. Эти числа суть индексы вершин в базовом DAG. Вершина **t** орграфа достижима из вершины **s** тогда и только тогда, когда **ID(t)** достижима из **ID(s)** в базовом DAG.

```
template <class Graph> class TC
{ const Graph &G; DenseGRAPH *K;
  dagTC<DenseGRAPH, Graph> *Ktc;
  SC<Graph> *Gsc;
public:
  TC(const Graph &G) : G(G)
    {
      Gsc = new SC<Graph>(G);
      K = new DenseGRAPH(Gsc->count(), true);
      for (int v = 0; v < G.V(); v++)
      { typename Graph::adjIterator A(G, v);
        for (int t = A.beg(); !A.end(); t = A.nxt())
          K->insert(Edge(Gsc->ID(v), Gsc->ID(t))); }
      Ktc = new dagTC<DenseGRAPH, Graph>(*K);
    }
  ~TC() { delete K; delete Ktc; delete Gsc; }
  bool reachable(int v, int w)
  { return Ktc->reachable(Gsc->ID(v), Gsc->ID(w));}
};
```

Программа 19.13 представляет собой реализацию АТД абстрактно-транзитивного замыкания, которое воплощает эти идеи. Мы используем интерфейс абстрактно-транзитивного замыкания также и в базовом DAG. Время прогона этой реализации зависит не только от числа вершин и ребер орграфа, но и от свойств базового DAG. В целях анализа предположим, что мы используем матрицу смежности для представления базового DAG, поскольку мы ожидаем, что базовый DAG имеет небольшие размеры, если только он еще и не насыщен.

Свойство 19.18. *Мы можем поддерживать постоянное время запроса АТД абстрактного транзитивного замыкания орграфа при затратах пространства памяти, пропорциональных $V + v^2$, и времени, пропорциональных $E + v^2 + vx$, на предварительную обработку (на вычисление транзитивного замыкания); здесь v есть число вершин в базовом DAG, а x — число поперечных ребер в его лесе DFS.*

Доказательство: Непосредственно следует из свойства 19.13. ■

Если орграф сам является DAG, то вычисления сильных компонент не дают никакой новой информации, а этот алгоритм тот же, что реализованный программой 19.9; однако в орграфах общего вида, в которых имеются циклы, этот алгоритм, по-видимому, обладает значительно более высоким быстродействием, чем алгоритм Уоршалла или решение на базе поиска в ширину. Например, из свойства 19.18 немедленно вытекает следующий результат.

Свойство 19.19. *Мы можем поддерживать постоянное время запроса для транзитивного замыкания любого графа, в базовом DAG которого содержится менее* $\sqrt[3]{V}$ *вершин, при затратах на предварительную обработку пространства памяти, пропорционального V, и времени, пропорционального E + V.*

Доказательство: Установим неравенство $v < \sqrt[3]{V}$ в свойстве 19.18 и примем во внимание, что $x < v2$. ■

Мы можем рассматривать другие вариации этих границ. Например, если мы хотим использовать пространство, пропорциональное E, мы можем достичь тех же временных границ, когда в базовом DAG содержатся до $\sqrt[3]{E}$ вершин. Более того, эти временные границы консервативны, поскольку они предполагают, что базовый DAG насыщен поперечными ребрами, – разумеется, это совсем не обязательно.

Главный ограниченный фактор применимости этого метода – это размер базового DAG. Чем больше рассматриваемый нами орграф приближается по своим характеристикам к графу DAG (чем больше его базовый DAG), тем с большими трудностями приходится сталкиваться при вычислении его транзитивного замыкания. Обратите внимание на тот факт, что мы (естественно) не нарушили нижней границы, устанавливаемой по свойству 19.9, поскольку рассматриваемый алгоритм выполняется для насыщенных DAG за время, пропорциональное V^3, однако мы существенно расширили класс графов, для которых можно избежать условий функционирования для худшего случая. В самом деле, построение модели случайного орграфа, генерирующей орграфы, на которых рассматриваемый алгоритм показывает низкое быстродействие, – задача не из легких (см. упражнение 19.142).

В таблицу 19.2 сведены результаты эмпирических исследований; она показывает, что случайные орграфы обладают небольшими базовыми графами даже для графов со средней насыщенностью и моделей с серьезными ограничениями на расположение ребер. И хотя в худшем случае нет никаких гарантий, на практике мы можем рассчитывать на получение орграфов с малыми базовыми DAG. Когда в нашем распоряжении имеются такие орграфы, мы можем надеяться, что получим эффективную реализацию АТД абстрактно-транзитивного замыкания.

Таблица 19.2. Свойства случайных орграфов

В данной таблице показано число вершин и ребер в базовых DAG случайных орграфов, построенных на базе двух различных моделей (ориентированные версии моделей из таблицы 18.1). В обоих случаях базовый граф DAG уменьшается (будучи разреженным) по мере возрастания насыщенности.

		Случайные ребра		Случайная из 10 соседних вершин	
	E	*v*	*e*	*v*	*e*
1000 вершин					
	1000	983	981	916	755
	2000	424	621	713	1039
	5000	13	13	156	313
	10000	1	1	8	17
	20000	1	1	1	1
10000 вершин					
	50000	144	150	1324	150
	100000	1	1	61	123
	20000	1	1	1	1

Обозначения:

v – число вершин в базовом DAG
e – число ребер в базовом DAG

Упражнения

• **19.138**. Разработайте версию реализации абстрактного транзитивного замыкания для орграфов, основанных на использовании представлений разреженных графов для базового DAG. Основная ваша задача состоит в устранении дубликатов из списка без излишних затрат времени или пространства памяти (см. упражнение 19.65).

▷ **19.139**. Покажите базовый DAG, вычисленный с помощью программы 19.13, и его транзитивное замыкание для орграфа

3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.

○ **19.140**. Преобразуйте реализацию абстрактно-транзитивного замыкания, основанную на вычислении сильных компонент (программа 19.13) в эффективную программу, которая вычисляет матрицы смежности транзитивного замыкания орграфа, представленного в виде матрицы смежности, используя для этой цели алгоритм Габова для вычисления сильных компонент и усовершенствованный алгоритм Уоршалла для вычисления транзитивного замыкания графа DAG.

19.141. Выполните эмпирические исследования с целью оценки ожидаемых размеров базовых DAG для различных типов орграфов (см. упражнения 19.11–19.18).

•• **19.142**. Разработайте модель случайных орграфов, которая генерирует орграфы, имеющие большие базовые DAG. Ваш генератор должен генерировать ребра по одному за раз, но он не должен использовать какие-либо свойства полученного графа.

19.143. разработайте абстрактного транзитивного замыкания орграфа путем выявления сильных компонент и построения базового DAG с последующими утвердительными ответами на запросы о достижимости, если заданные две вершины находятся в одной и той же сильной компоненте, и выполнением в противном случае поиска в глубину в DAG с тем, чтобы определить их достижимость.

19.10. Перспективы

В настоящей главе мы рассматривали алгоритмы решения задач топологической сортировки, транзитивного замыкания и нахождения кратчайшего пути для орграфов и для графов DAG, в том числе и фундаментальные алгоритмы выявления циклов и сильных компонент в орграфах. Эти алгоритмы находят многочисленные важные приложения сами по себе, а также служат основой решения более сложных задач, включая взвешенные графы, которые мы будем рассматривать в двух следующих главах. Значения времени прогона этих алгоритмов в худших случаях суммируются в таблице 19.3.

Таблица 19.3. Затраты на выполнение операций обработки орграфов в худшем случае

В эту таблицу сведены итоговые данные затратах (время выполнения в худшем случае) на выполнение алгоритмов решения различных задач обработки орграфов, рассмотренных в этой главе, для случайных графов и графов, в которых ребра случайным образом соединяет каждую вершину с одной из 10 заданных соседних вершин. Все затраты определялись на представлении графов в виде списков смежных вершин; в случае представления графа в виде матрицы смежности E элементов становятся V^2 элементами, поэтому, например, затраты на вычисление всех кратчайших путей составляют V^3. Линейные по времени алгоритмы суть оптимальные алгоритмы, так что затраты позволяют с достаточной уверенностью предсказывать время выполнения при любом вводе; другие алгоритмы дают оценку с чрезмерным запасом затрат, таким образом, время выполнения для некоторых типов графов может оказаться ниже. Рабочие характеристики алгоритма вычисления транзитивного замыкания орграфа, обладающего наибольшим быстродействием, зависит от структуры орграфа, в частности, от размеров его базового DAG.

	Задача	Затраты	Алгоритмы
Орграфы			
	Определение циклов	E	Поиск в глубину
	Транзитивное замыкание	V(E+V)	Поиск в глубину из каждой вершины
	Кратчайший путь одного источника	E	Поиск в глубину
	Все кратчайшие пути	V(E+V)	Поиск в глубину из каждой вершины
	Сильные компоненты	E	Косарайю, Тарьяна или Габова
	Транзитивное замыкание	E+v(v+x)	Базовый DAG
Графы DAG			
	Ациклическая проверка	E	Поиск в глубину или очередь истоков
	Топологическая сортировка	E	Поиск в глубину или очередь истоков
	Транзитивное замыкание	V(E+V)	Поиск в глубину
	Транзитивное замыкание	V(E+X)	Поиск в глубину/динамическое программирование

В частности, общей темой на протяжении данной главы является решение задачи абстрактно-транзитивного замыкания, при этом мы хотели обеспечить поддержку АТД, который, после предварительной обработки, может определить, существует ли ориентированный путь из одной заданной вершины в другую. Несмотря на то что нижняя граница определяет, что затраты на предварительную обработку в худшем случае существенно превосходят V^2, метод, обсуждавшийся в разделе 19.7, показывает, что базовые методы, исследуемые в этой главе, сводятся к простому решению, которое обеспечивает оптимальную производительность на многих типах орграфах; более-менее заметное исключение составляют лишь насыщенные графы DAG. Нижняя граница предполагает, что более высокая гарантированная производительность едва ли может быть достигнута на всех графах, в то же время мы можем пользоваться этими методами, чтобы получить приемлемую производительность на графах, встречающихся на практике.

Цель разработки алгоритмов с рабочими характеристиками, аналогичными характеристиками алгоритма объединения-поиска, который изучался в главе 1, для насыщенных орграфов остаются призрачными. В идеальном случае мы могли бы определять АТД, посредством которого мы могли бы добавлять ориентированные ребра или проверять, достижима ли одна вершина из другой, и разработать реализацию, в которой мы могли бы поддерживать выполнение всех операций за постоянное время (см. упражнения 19.153–19.155). Как отмечалось в главе 1, мы можем достаточно близко подойти к этой цели в случае неориентированных графов, но сравнимые с этим решения для орграфов или графов DAG до сих пор не известны. (Обратите внимание на тот факт, что удаление ребер представляет собой сложную проблему даже для неориентированных графов.) Эта задача *динамической достижимости (dynamic reachability)* не только вызывает интерес у математиков и имеет широкое практическое применение, но и играет исключительно важную роль в разработке алгоритмов на более высоком уровне абстракции. Например, достижимость лежит в основе задачи реализации сетевого симплексного алгоритма для определения потоков с минимальной стоимостью, представляющей собой модель решения задач, которая получила широкое применение и которую мы будем рассматривать в главе 22.

Множество других алгоритмов обработки орграфов и графов DAG находят важное практическое применение и подвергаются подробным исследованиям, в то же время многие задачи обработки орграфов ждут своих исследователей, способных разработать эффективные алгоритмы их решения. Список таких задач достигает внушительных размеров.

Доминаторы. Пусть задан DAG все, вершины которого достижимы из одного истока r, вершина v *доминирует (dominates)* над вершиной t, если каждый путь из r в t содержит s. (В частности, каждая вершина доминирует сама над собой.) Каждая вершина v, отличная от источника, имеет *непосредственного доминатора*, который доминирует над вершиной v, но в то же время не доминирует над другим доминатором v и над самим собой. Множество непосредственных доминаторов представляет собой дерево, которое охватывает все вершины, достижимые из источника. Эта структура имеет важное значение для построения компиляторов. Доминаторное дерево может быть вычислено за линейное время путем применения подхода, основанного на поиске в глубину, который использует несколько вспомогательных структур данных, хотя обычно на практике применяется несколько более медленная его версия.

Транзитивное сокращение. Пусть задан орграф, требуется найти орграф, который имеет то же транзитивное замыкание и минимальное число ребер среди орграфов этого класса. Эта задача поддается решению (см. упражнение 19.150); однако если наложить на нее дополнительное условие, потребовав, чтобы результатом был подграф исходного графа, она сразу же становится NP-трудной.

Ориентированный эвклидов путь. Существует ли на орграфе путь, соединяющий две заданных вершины, который использует каждое ребро орграфа в точности один раз? Эта задача решается довольно просто в силу тех же причин, что и соответствующая задача для неориентированных графов, которую мы рассматривали в разделе 17.7 (см. упражнение 17.92).

Ориентированный маршрут почтальона. Пусть задан граф, существует ли ориентированный цикл с минимальным числом ребер, который использует каждое ребро графа, по меньшей мере, один раз (при этом он имеет возможность воспользоваться каждым конкретным ребром многократно). Как мы увидим в разделе 22.7, эта задача сводится к задаче выявления потоков с минимальной стоимостью, и в силу этого обстоятельства она поддается решению.

Ориентированный гамильтонов путь. Найти в орграфе простой ориентированный путь максимальной длины. Эта задача относится к числу NP-трудных, однако она легко решается, если граф представляет собой DAG (см. упражнение 19.114).

Односвязный подграф. Говорят, что граф *односвязный (uniconnected)*, если существует самое большее один ориентированный путь между любой парой вершин. Если задан орграф и целое число k, определить, существует ли односвязный подграф, имеющий, по меньшей мере, k ребер. Известно, что для произвольного k эта задача принадлежит к категории NP-трудных.

Множество вершин обратной связи. Определить, имеет ли заданный орграф подмножество, состоящее максимум из k вершин, которое содержит, по меньшей мере, одну вершину из каждого направленного цикла в G. Известно, это задача принадлежит к категории NP-трудных.

Четные циклы. Определите, содержит ли заданный орграф цикл четной длины. Как было показано в разделе 17.8, это проблема не принадлежит к числу трудно разрешимых, тем не менее, никому не удалось получить алгоритм, который можно было бы использовать на практике.

Так же как и в случае неориентированных графов, было исследовано великое множество задач, при этом часто определение того, является ли рассматриваемая задача легко или трудно решаемой, сама по себе является сложной проблемой (см. раздел 17.8). Как подчеркивалось на протяжении всей этой главы, некоторые обнаруженные нами факты, касающиеся орграфов, суть выражения более общих математических закономерностей, а многие из используемых нами алгоритмов имеют применение на уровнях абстракций, отличающихся от тех, на которых мы работаем. С другой стороны, понятие трудно решаемой задачи говорит нам о том, что в своем стремлении получить эффективные алгоритмы, обеспечивающие эффективное решение тех или иных задач, мы можем столкнуться с фундаментальными препятствиями. С другой стороны, классические алгоритмы, описанные в настоящей главе, имеют фундаментальное значение и находят широкое применение на практике, поскольку они предлагают эффективные решения задач, которые часто возникают в реальной жизни и не могут быть решены каким-либо другим способом.

Упражнения

19.144. Внесите соответствующие изменения в программы 17.18 и 17.19, позволяющие реализовать функцию АТД распечатки эйлерова пути в орграфе, если таковой существует. Поясните назначение каждого изменения и дополнения, которые придется внести в программный код.

▷ **19.145**. Начертите дерево доминаторов орграфа

3-7 1-4 7-8 0-5 5-2 3-0 2-9 0-6 4-9 2-6
6-4 1-5 8-2 9-0 8-3 4-5 2-3 1-6 3-5 7-6

●● **19.146**. Постройте класс, использующий поиск в глубину для создания представления доминаторного дерева заданного орграфа в виде родительских связей (*см. раздел ссылок*).

○ **19.147**. Найдите транзитивное сокращение орграфа

3-7 1-4 7-8 0-5 5-2 3-0 2-9 0-6 4-9 2-6
6-4 1-5 8-2 9-0 8-3 4-5 2-3 1-6 3-5 7-6

● **19.148**. Найдите подграф графа

3-7 1-4 7-8 0-5 5-2 3-0 2-9 0-6 4-9 2-6
6-4 1-5 8-2 9-0 8-3 4-5 2-3 1-6 3-5 7-6,

имеющего то же транзитивное замыкание и минимальное число ребер среди всех таких подграфов.

○ **19.149**. Докажите, что каждый DAG имеет уникальное транзитивное сокращение, и дайте эффективную реализацию функции АТД для вычисления транзитивного сокращения DAG.

● **19.150**. Постройте эффективную функцию АТД для орграфов, которая вычисляет транзитивное сокращение.

19.151. Предложите алгоритм, который определяет, принадлежит ли заданный орграф к категории односвязных. Предлагаемый вами алгоритм в худшем случае должен характеризоваться временем выполнения, пропорциональным VE.

19.152. Найдите односвязный подграф максимальных размеров орграфа

3-7 1-4 7-8 0-5 5-2 3-0 2-9 0-6 4-9 2-6
6-4 1-5 8-2 9-0 8-3 4-5 2-3 1-6 3-5 7-6.

▷ **19.153**. Разработайте реализацию класса орграфа, который поддерживает операции вставки ребра, удаления ребра и проверку, находятся ли заданные вершины в одной и той же сильной компоненте, такого, что в худшем случае для выполнения всех операций построения, вставки ребер и удаления требуется линейное время, а на выполнение запросов о сильной связности – постоянное время.

○ **19.147**. Решите упражнение 19.153 таким образом, чтобы запросы на вставку ребра, удаления ребра и принадлежность к сильной компоненте выполнялись за время, пропорциональное $\log V$ в худшем случае.

●●● **19.146**. Решите упражнение 19.153 таким образом, чтобы запросы на вставку ребра, удаления ребра и принадлежность к сильной компоненте выполнялись за время, близкое к постоянному (как это имеет место для алгоритмов поиска-объединения для определения связности в неориентированных графах).

Глава 20

Минимальные остовные деревья

Модели графов, в которых мы связываем *веса (weights)* или *стоимости (costs)* с каждым ребром, используются во многих приложениях. В картах авиалиний, в которых ребрами отмечены авиарейсы, такие веса означают расстояния или стоимость проезда. В электронных схемах, где ребра представляют электрические соединения, веса могут означать длину соединения, его стоимость или время, необходимое для распространения по ней сигнала. В задачах календарного планирования веса могут представлять время или расходы, затрачиваемые на выполнение задач, либо время ожидания завершения соответствующей задачи.

Естественно, в таких ситуациях возникают вопросы, касающиеся различных аспектов минимизации стоимости. Мы будем изучать алгоритмы решения двух задач такого рода: (*i*) определение пути с наименьшей стоимостью, соединяющего все точки, и (*ii*) отыскание пути наименьшей стоимости, соединяющего две заданных точки. Первый тип алгоритма применяется для решения задач на неориентированных графах, которые представляют такие объекты как электрические цепи, находит *минимальное остовное дерево (minimum spanning tree)*; это дерево является основной темой данной главы. Второй тип алгоритма, применяемый для решения задач на орграфах, которые представляют такие объекты, как карты авиарейсов, определяет *кратчайшие пути (shortest path)*, и они являются темой обсуждений в главе 21. Эти алгоритмы находят широкое применение не только в приложениях, связанных со схемами авиарейсов и электрическими схемами, они используются также и при решении задач, возникающих во взвешенных графах.

Когда мы изучаем алгоритмы обработки взвешенных графов, наша интуиция часто отождествляет веса с расстояни-

ями: мы употребляем выражение "ближайшая к *x* вершина" и ему подобные. Действительно, термин "кратчайший путь" оказывается в русле этой тенденции. Однако, несмотря на многочисленные приложения, в которых мы по существу имеем дело с расстояниями, и несмотря на все преимущества геометрической интуиции в понимании базовых алгоритмов, важно помнить, что веса не обязательно должны быть пропорциональны расстоянию, они могут представлять время и стоимость или совсем другую переменную величину. В самом деле, как мы убедимся в главе 21, веса в условиях задач определения кратчайшего пути могут принимать даже *отрицательные* значения.

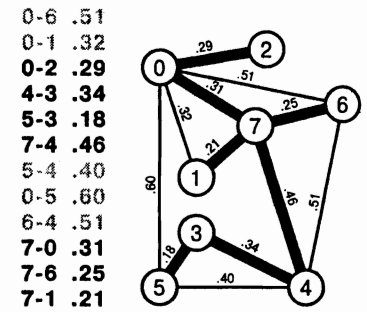


РИСУНОК 20.1. ВЗВЕШЕННЫЙ НЕОРИЕНТИРОВАННЫЙ ГРАФ И ЕГО ДЕРЕВО MST

Взвешенный неориентированный граф представляет собой набор взвешенных ребер. Дерево MST есть множество ребер минимального общего веса, которые соединяют все вершины (выделены черным в списке ребер, утолщенные ребра на чертеже графа). В рассматриваемом графе веса пропорциональны расстояниям между вершинами, однако базовые алгоритмы, которые мы исследуем, соответствуют графам общего вида и не накладывают никаких условий на веса (см. рис. 20.2).

Прибегая к помощи интуиции при описании алгоритмов и примеров и сохраняя при этом общую применимость, мы используем неоднозначные термины, когда попеременно ссылаемся на длины и веса. В большей части примеров, приводимых в настоящей главе, мы используем веса, пропорциональные расстоянию между вершинами, как это нетрудно видеть на рис. 20.1. Такие графы удобно рассматривать в качестве примеров, поскольку нет необходимости указывать метки ребер, ибо достаточно одного взгляда, дабы уяснить, что более длинные ребра имеют большие веса по сравнению с короткими ребрами. Когда веса представляют расстояния, мы получаем возможность рассматривать алгоритмы, которые становятся более эффективными, когда учитывают геометрические свойства графов (разделы 20.7 и 21.5). За этим исключением, алгоритмы, которые мы будем исследовать, просто выполняют обработку ребер и не извлекают никаких преимуществ из содержащейся в графах геометрической информации (см. рис. 20.2).

Задача поиска минимального остовного дерева произвольного взвешенного неориентированного графа получила множество важных применений, и алгоритмы ее решения были известны, по меньшей мере, с двадцатых годов прошлого столетия. Тем не менее, эффективность ее реализации колеблется в широких пределах, а исследователи до сих пор заняты поисками более совершенных методов. В этом разделе мы будем изучать три классических алгоритма, которые легко понять на концептуальном уровне; в разделах 20.3–20.5 мы подробно изучим реализации каждого из них, в разделе 20.6 проведем сравнение этих фундаментальных подходов и внесем в них некоторые усовершенствования.

Определение 20.1. Дерево MST *(minimal spanning tree — минимальное остовное дерево) взвешенного графа есть остовное дерево, вес которого (сумма весов его ребер) не превосходит вес любого другого остовного дерева.*

Если все веса положительны, достаточно определить дерево MST как множество ребер с минимальным общим весом, которые соединяют все вершины, ибо такое множе-

ство и должно образовать остовное дерево. Условие остовного дерева включено в определение для того, чтобы его можно было применять к графам, которые могут иметь отрицательные веса ребер (см. упражнение 20.2 и 20.3).

Если ребра могут иметь равные веса, минимальное остовное дерево может не быть единственным. Например, на рис. 20.2 показан граф, который имеет два различных MST. Возможность равных весов усложняет описание и доказательство правильности некоторых рассматриваемых нами алгоритмов. Мы должны внимательно изучать случаи с равными весами, поскольку они довольно часто встречаются на практике, в то же время мы хотим, чтобы наши алгоритмы работали правильно и в условиях равновесных ребер.

Возможно существование сразу нескольких MST, это обозначение не отражает достаточно четко тот факт, что мы минимизируем вес, а не само дерево. Наиболее подходящее прилагательное для описания свойства такого дерева есть *минимальное* (т.е., дерево, имеющее минимальный вес). В силу этих причин многие авторы употребляют более точные понятия, такие как *минимальное остовное дерево* или *остовное дерево с минимальным весом*. Аббревиатура *MST*, которую мы будем употреблять, по всеобщему мнению, довольно часто достаточно точно отражает это базовое понятие.

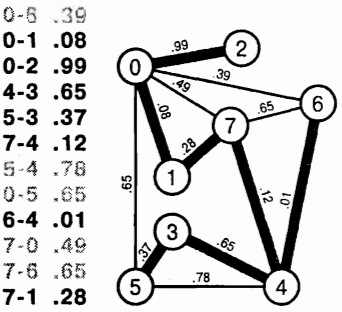


РИС. 20.2. ПРОИЗВОЛЬНЫЕ ВЕСА

*В этом примере веса ребер выбраны произвольно и не имеют никакого отношения к геометрии изображенного здесь представления графа. Этот пример также служит иллюстрацией того факта, что дерево MST не обязательно уникально: мы получаем одно дерево MST, включая ребро **3-4** (показано на рисунке), и другое дерево MST, включая вместо него **0-5** (хотя ребро **7-6** имеет тот же вес, что эти два ребра, оно не появляется ни в одном из этих MST).*

И опять-таки, во избежание путаницы при описании алгоритмов на сетях, в которых вполне могут быть ребра с равными весами, мы должны очень осторожно использовать термин "минимальный" для обозначения "ребра минимального веса" (среди всех ребер конкретного множества) и термин "максимальный" для обозначения "ребра максимального веса". Иначе говоря, если ребра различны, минимальным ребром является самое короткое ребро (такое ребро будет единственным); но если существует несколько ребер минимального веса, любое из них может быть минимальным.

В данной главе мы будем работать исключительно с неориентированными графами. Задача поиска на орграфе ориентированного остового дерева с минимальным весом не принадлежит к этому классу, к тому же она намного труднее.

Для решения задачи MST были разработаны несколько классических алгоритмов. Эти методы принадлежат к числу самых старых и самых известных алгоритмов из множества опубликованных в этой книге. Как мы уже могли убедиться раньше, классические методы предлагают общий подход, в то же время современные алгоритмы и структуры данных позволяют получать компактные и эффективные программные реализации этих алгоритмов. В самом деле, такие реализации представляют собой убедительные примеры эффективности тщательного проектирования АТД и правильного выбора фундаментальных структур данных АТД и реализаций алгоритмов при решении алгоритмических задач, сложность которых неуклонно возрастает.

Упражнения

20.1. Предположим, что веса в графе положительны. Докажите, что вы можете изменить их вес путем добавления постоянной величины к каждому из них или путем умножения на некоторую постоянную величину, и при этом деревья MST не изменятся при условии, что новые веса принимают положительные значения.

20.2. Покажите, что если веса ребер положительны, то множество ребер, соединяющих все вершины, суммарный вес которых не больше суммы весов любого другого множества ребер, соединяющих все эти вершины, есть дерево MST.

20.3. Покажите, что свойство, сформулированное в упражнении 20.2, выполняется для графов с отрицательными весами при условии, что не существуют циклы, все ребра которых имеют неположительные веса.

○ **20.4.** Как найти *максимальное* остовое дерево взвешенного графа?

▷ **20.5.** Покажите, что если все ребра графа имеют различные веса, то дерево MST уникально.

▷ **20.6.** Выполните анализ утверждения, что граф обладает уникальным деревом MST, только когда веса его ребер различны. Дайте доказательства или приведите противоположный пример.

● **20.7.** Предположим, что граф имеет $t < V$ ребер с равными весами и что все другие веса различны. Какими будут верхняя и нижняя границы числа различных деревьев MST, которые могут быть у графа?

20.1. Представления

В этой главе мы сосредоточим свое внимание на изучении взвешенных неориентированных графов – наиболее естественная постановка задачи о MST. Возможно, проще всего начать это изучение с расширения базовых представлений графа, предложенных в главе 17, и представить граф в следующем виде: в матрице смежности вместо булевых значений могут быть проставлены веса ребер; в представление графа в виде списков смежных вершин мы можем добавить поле весов в список элементов, представляющих ребра. Этот классический подход привлекает своей простотой, но мы будем пользоваться другим методом, который не намного сложнее, зато позволяет использовать наши программы в более общих условиях. Сравнение некоторых рабочих характеристик обоих подходов будет проведено далее в данной главе.

Для исследования всех проблем, возникающих при переходе от графов, где нас интересует только наличие или отсутствие ребер, к графам, в которых наш интерес сосредоточен на информации, связанной с ребрами, полезно составить сценарий, по которому вершины и ребра суть объекты неизвестной сложности; возможно, они являются некоторой частью крупной клиентской базы данных, которая построена и поддерживается другим приложением. Например, возможно, имеет смысл рассматривать улицы, дороги, шоссе в базе данных географической карты как абстрактные ребра, интерпретируя их длину как вес. В то же время, запись в базе данных, содержащей информацию о дороге, может содержать и другую информацию, например, название дороги и ее тип, детальное описание физических свойств ее покрытия, транспортные потоки, проходящие через нее ежедневно, и т.п. Клиентская программа может брать из нее нужную инфор-

мацию, строить соответствующий граф, выполнять его обработку, а затем интерпретировать полученные результаты в контексте базы данных, в то же время все это может оказаться трудным и дорогостоящим процессом. В частности, он требует (по меньшей мере) построения копии обрабатываемого графа.

Программа 20.1 Интерфейс АТД графов со взвешенными ребрами

Этот программный код определяет интерфейс для графов с весами и другую информацию, имеющую отношение к ребрам. Он содержит интерфейс АТД **EDGE** и шаблонный интерфейс **GRAPH**, который может использоваться в любой реализации интерфейса **EDGE**. Реализации интерфейса **GRAPH** манипулируют указателями ребра (клиенты снабжают ее функцией **insert**), но не ребрами. Класс ребер содержит также функции, которые предоставляют информацию об ориентации ребра: либо **e->from(v)** есть **true**, **e->v()** есть **v** и **e->other(v)** есть **e->w()**; либо **e->from(v)** есть **false**, **e->w()** есть **v** и **e->other(v)** есть **e->v()**.

```
class EDGE
  {
    public:
      EDGE(int, int, double);
      int v() const;
      int w() const;
      double wt() const;
      bool from(int) const;
      int other(int) const;
  };
template <class Edge> class GRAPH
  {
    public:
      GRAPH(int, bool);
      ~GRAPH();
      int V() const;
      int E() const;
      bool directed() const;
      int insert(Edge *);
      int remove(Edge *);
      Edge *edge(int, int);
      class adjIterator
        {
          public:
            adjIterator(const GRAPH &, int);
            Edge *beg();
            Edge *nxt();
            bool end();
        };
  };
```

Более приемлемая для клиентов альтернатива предусматривает объявление типа данных **Edge** и, в условиях нашей реализации, манипулирование *указателями* на ребра. В программе 20.1 содержатся подробности использования интерфейсов **EDGE**, **GRAPH** и итератора АТД, который применяется для поддержки этого подхода. Для использования в других контекстах клиенты легко могут выполнить минимальные требования типа данных **Edge**, сохраняя при этом гибкость, позволяющую давать определения типов данных, отражающих специфику конкретных приложений. Предлагаемые нами реализации могут манипулировать указателями на ребра и использовать интерфейс с целью извлечения не-

обходимой им информации из интерфейсов **EDGE** независимо от их представления. Программа 20.2 представляет собой пример клиентской программы, которая использует этот интерфейс.

Программа 20.2. Пример клиентской функции обработки графа

Данная функция служит иллюстрацией использования интерфейса взвешенного графа, предложенного нами в программе 20.1. При любой реализации этого интерфейса функция **edges** возвращает вектор, содержащий указатели на все ребра графа. Как и в главах 17–19, мы в общем случае используем функцию итератора только так, как показано здесь.

```
template <class Graph, class Edge>
vector <Edge *> edges(const Graph &G)
{ int E = 0;
   vector <Edge *> a(G.E());
   for (int v = 0; v < G.V(); v++)
     {
       typename Graph::adjIterator A(G, v);
       for (Edge* e = A.beg(); !A.end(); e = A.nxt())
          if (e->from(v)) a[E++] = e;
     }
   return a;
}
```

Программа 20.3. Класс взвешенного графа (представление в виде матрицы смежности)

Для насыщенных взвешенных графов мы используем матрицу указателей на данные типа **Edge** с указателем на ребро в **v-u** в строке **v** и столбце **w**. Для неориентированных графов мы устанавливаем другой указатель на ребро в строке **w** и столбце **v**. Нулевой указатель свидетельствует об отсутствии соответствующего ребра; когда мы удаляем ребро с помощью функции **remove()**, мы удаляем указатель на него. Рассматриваемая реализация не производит проверки на наличие параллельных ребер, в то же время клиенты могут для этой цели воспользоваться функцией **edge**.

```
template <class Edge> class DenseGRAPH
{ int Vcnt, Ecnt; bool digraph;
  vector <vector <Edge *> > adj;
public:
  DenseGRAPH(int V, bool digraph = false) :
    adj(V), Vcnt(V), Ecnt(0), digraph(digraph)
    {
      for (int i = 0; i < V; i++)
        adj[i].assign(V, 0);
    }
  int V() const { return Vcnt; }
  int E() const { return Ecnt; }
  bool directed() const { return digraph; }
  void insert(Edge *e)
    { int v = e->v(), w = e->w();
      if (adj[v][w] == 0) Ecnt++;
      adj[v][w] = e;
      if (!digraph) adj[w][v] = e;
    }
  void remove(Edge *e)
    { int v = e->v(), w = e->w();
      if (adj[v][w] != 0) Ecnt--;
      adj[v][w] = 0;
```

```
    if (!digraph) adj[w][v] = 0;
   }
  Edge* edge(int v, int w) const
   { return adj[v][w]; }
  class adjIterator;
  friend class adjIterator;
};
```

В алгоритмах тестирования и в базовых приложениях мы используем класс **EDGE** (ребро), имеющий два приватных элемента данных типа **int** и **double**, которые инициализируются аргументами конструктора и являются возвращаемыми значениями, соответственно, функций-элементов **v()**, **w()** и **wt()** (см. упражнение 20.8). Во избежание распространения простых типов данных, на протяжении данной главы и главы 21 мы будем использовать для представления веса ребер тип данных **double**. В наших примерах в качестве весов ребер мы будем использовать вещественные числа из диапазона от 0 до 1. Это решение не противоречит различным альтернативам, которые могут понадобиться в приложениях, поскольку мы можем явно или неявно изменить веса таким образом, чтобы они соответствовали этой модели (см. упражнения 20.1 и 20.10.). Например, если весами являются целые числа, меньшие некоторого известного максимального значения, то поделив значения весов на это максимальное значение, мы преобразуем их в вещественные числа, принимающие значения в диапазоне от 0 до 1.

Программа 20.4. Класс итератора, ориентированный на представление графа в виде матрицы смежности

Данный программный код, возвращающий указатели на ребра, является простой адаптацией программы 17.8.

```
template <class Edge>
class  DenseGRAPH<Edge>::adjIterator
{ const DenseGRAPH<Edge> &G;
  int i, v;
public:
  adjIterator(const DenseGRAPH<Edge> &G, int v) :
   G(G), v(v), i(0) { }
  Edge *beg()
   { i = -1; return nxt(); }
  Edge *nxt()
   {
    for (i++; i < G.V(); i++)
      if (G.edge(v, i)) return G.adj[v][i];
    return 0;
   }
  bool end() const
   { return i >= G.V(); }
};
```

При желании можно было бы разработать интерфейс АТД более общего характера, и использовать для весов ребер любой тип данных, который поддерживает операции сложения, вычитания и сравнения, поскольку мы совершаем более сложные манипуляции с весами, чем просто накопление их сумм и принятие решений в зависимости от значений этих сумм. В алгоритмах, построенных в главе 22, мы выполняем сравнения линейных комбинаций весов ребер, а время выполнения некоторых алгоритмов зависит от ариф-

метических свойств весов, вследствие чего мы переходим к использованию целочисленных весов, тем самым упрощая анализ алгоритмов.

Программы 20.2 и 20.3 реализуют абстрактный тип данных взвешенного графа, представленного в виде матрицы смежности, который впервые был использован в программе 20.1. Как и ранее, вставка ребра в неориентированный граф сопряжена с хранением указателей на него в двух местах матрицы – по одному для каждой ориентации ребра. Как это характерно для алгоритмов, предполагающих применение представлений неориентированных графов в виде матриц смежности, их время выполнения пропорционально V^2 (на инициализацию матрицы) и выше.

Имея такое представление, мы проводим тест на существование ребра **v-w** путем проверки, принимает ли указатель, расположенный на пересечении строки **v** и столбца **w**, нулевое значение. В некоторых ситуациях мы можем избежать проверок подобного рода, воспользовавшись сигнальными значениями для весов, однако в своих реализациях мы не будем использовать сигнальные значения.

Программа 20.5. Класс взвешенного графа (списки смежных вершин)

Данная реализация интерфейса из программы 20.1 основана на представлении графа в виде списков смежных вершин и поэтому хорошо подходит для разреженных взвешенных графов. Как и в случае невзвешенных графов, мы представляем каждое ребро узлом списка, но в рассматриваемом случае каждый узел содержит указатель на ребро, которое он представляет, а не только на вершину назначения. Класс итератора представляет собой непосредственную адаптацию программы 17.10 (см. упражнение 20.13).

```
template <class Edge> class SparseMultiGRAPH
{ int Vcnt, Ecnt; bool digraph;
   struct node
    { Edge* e; node* next;
       node(Edge* e, node* next): e(e), next(next) {}
    };
   typedef node* link;
   vector <link> adj;
public:
   SparseMultiGRAPH(int V, bool digraph = false) :
    adj(V), Vcnt(V), Ecnt(0), digraph(digraph) { }
   int V() const { return Vcnt; }
   int E() const { return Ecnt; }
   bool directed() const { return digraph; }
   void insert(Edge *e)
    {
      adj[e->v()] = new node(e, adj[e->v()]);
      if (!digraph)
        adj[e->w()] = new node(e, adj[e->w()]);
      Ecnt++;
    }
   class adjIterator;
   friend class adjIterator;
};
```

В программе 20.5 содержатся подробности реализации АТД взвешенного графа, который использует указатели на ребра в представлении графа в виде матрицы смежности. Вектор, проиндексированный именами вершин, ставит в соответствие каждой вершине связный список инцидентных ей ребер. Каждый узел списка содержит указатель на не-

которое ребро. Как и в случае матрицы смежности, мы можем при желании сэкономить пространство памяти, поместив вершину назначения и вес в узел списка (оставляя неназванной соответствующую вершину-источник) ценой усложнения итератора (см. упражнения 20.11 и 20.14).

На этом этапе полезно сравнить эти представления с простыми представлениями, о которых шла речь в начале этого раздела (см. упражнения 20.11 и 20.12). Если бы мы строили граф с нуля, то использование указателей потребовало бы гораздо большего пространства памяти. Это пространство нужно не только для размещения указателей, но и для размещения индексов (имен вершин), которые в простых реализациях представлены неявно. Чтобы иметь возможность пользоваться указателями на ребра в представлении графа в виде матрицы смежности, требуется дополнительное пространство памяти для размещения V^2 указателей на ребра и E пар индексов. Аналогично, чтобы иметь возможность пользоваться указателями на ребра в условиях представления графа в виде списков смежных вершин, требуется дополнительное пространство памяти для размещения E указателей на ребра и E индексов.

С другой стороны, использование указателей на ребра, по-видимому, позволит получить более высокопроизводительный программный код, поскольку скомпилированный код клиентской программы использует указатель с тем, чтобы получить прямой доступ к весу соответствующего ребра, в отличие от простой реализации, в рамках которой требуется построить тип данных **Edge** (ребро) и получить доступ к его полям. Если стоимость требуемого пространства памяти непомерно высока, то применение минимальных представлений (и, возможно, выбор оптимальной организации итераторов с целью экономии времени) может оказаться разумной альтернативой; в противном случае, гибкость, обеспечиваемая применением указателей, по всем признакам, стоит дополнительных затрат памяти.

Во избежание путаницы, на всех рисунках будут применяться простые представления. То есть, вместо того, чтобы показывать матрицы указателей на реберные структуры, мы просто показываем матрицы весов, а вместо того, чтобы показывать узлы списков, содержащие указатели на реберную структуру, мы будем показывать узлы, которые содержат вершины назначения ребер. Представления одного и того же графа в виде матриц смежности и в виде списков смежных вершин, приведены на рис. 20.3.

Что касается реализаций неориентированных графов, то ни в одной из реализаций мы не выполняем проверку на наличие параллельных ребер. В зависимости от приложения, мы можем внести такие изменения в представление графа в виде матрицы смежности, которые позволили бы содержать параллельные ребра с наименьшим или наибольшим весом либо эффективно сливать параллельные ребра в единое ребро, устанавливая ему вес, равный сумме весов параллельных ребер. В представлении графа в виде списков смежных вершин мы оставляем параллельные ребра в структуре данных, но мы могли бы построить более совершенные структуры данных, позволяющие отказаться от них, используя одно из только что упомянутых правил, применяемых к матрицам смежности (см. упражнение 17.49).

Как мы должны представить само дерево MST? Дерево MST графа G есть подграф графа G, который сам по себе является деревом, в связи с чем появляется множество вариантов, основными из которых являются:

	0	1	2	3	4	5	6	7
0	*	.32	.29	*	*	.60	.51	.31
1	.32	*	*	*	*	*	*	.21
2	.29	*	*	*	*	*	*	*
3	*	*	*	*	.34	.18	*	*
4	*	*	*	.34	*	.40	.51	.46
5	.60	*	*	.18	.40	*	*	*
6	.51	*	*	*	.51	*	*	.25
7	.31	.21	*	*	.46	*	.25	*

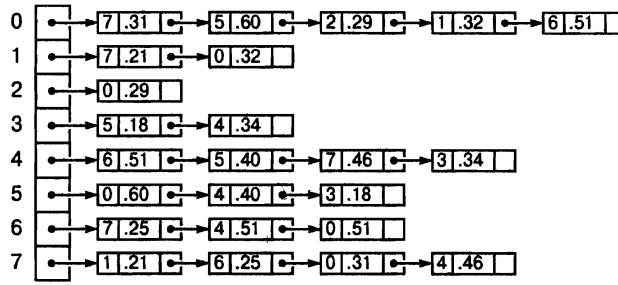


РИСУНОК 20.3. ПРЕДСТАВЛЕНИЯ ВЗВЕШЕННОГО ГРАФА (НЕОРИЕНТИРОВАННОГО)

Два стандартных представления взвешенных неориентированных графов содержат веса в каждом представлении ребра, как видно из представлений графа, изображенного на рис. 20.1, в виде матрицы смежности (слева) и в виде списков смежных вершин (справа). Для простоты мы показываем веса на этих рисунках в элементах матрицы и в узлах списков; в наших программах мы используем указатели на ребра клиентов. Матрица смежности симметрична, а списки смежных вершин содержат два узла для каждого ребра, как и в случае невзвешенных ориентированных графов. Несуществующие ребра представлены в матрице нулевыми указателями (обозначены на рисунке звездочками) и в списках попросту отсутствуют. Петли отсутствуют в обоих показанных здесь представлениях, поскольку алгоритмы MTS без них оказываются проще, тогда как другие алгоритмы обработки взвешенных графов их используют (см. главу 21).

- Граф.
- Связный список ребер.
- Вектор указателей на ребра.
- Вектор, индексированный именами вершин с родительскими связями.

Рисунок 20.4 служит иллюстрацией перечисленных вариантов на примере дерева MST, изображенного на рис. 20.1. Другая альтернатива заключается в том, чтобы дать определение и использовать АТД для представления деревьев.

Одно и то же дерево допускает различные представления в любой из указанных выше схем. В каком порядке должны быть приведены ребра в представлении в виде списков ребер? Какой узел должен быть выбран в качестве корня в представлении в виде родительских связей (см. упражнение 20.21)? По существу, во время выполнения алгоритма MST конкретное представление дерева MST, какое мы получаем при этом, есть артефакт используемого алгоритма и не отражает каких-либо важных свойств дерева MST.

Выбор того или иного представления дерева MST не оказывает заметного влияния на алгоритм, поскольку мы легко можем преобразовать каждое из этих представлений в любое другое. Чтобы выполнить преобразование представления дерева MST в виде графа в вектор ребер, мы можем воспользоваться функцией **GRAPHedges** из программы 20.2. Чтобы преобразовать представление в виде родительских связей, зафиксированных в векторе **st** (при этом веса представлены в векторе **wt**) в вектор **mst** указателей на ребра, включенные в дерево MST, можно воспользоваться циклом:

```
for (k = 1; k < G.V(); k++)
    mst[k] = new EDGE (k, st[k], wt[k]);
```

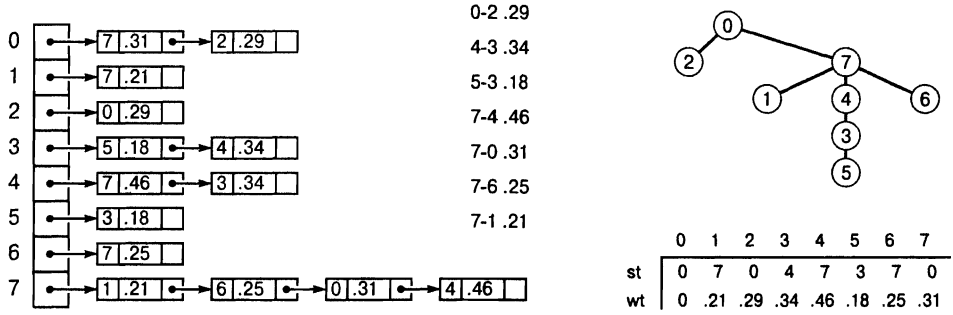


РИСУНОК 20.4. ПРЕДСТАВЛЕНИЕ ДЕРЕВА MST

На этом рисунке показаны различные представления дерева MST, показанного на рис. 20.1. Наиболее простым из них является список его ребер, при этом какой-либо порядок отсутствует (диаграмма слева). К тому же изображенное дерево MST есть разреженный граф, и оно может быть представлено в виде списков смежных вершин (диаграмма в центре). Наиболее компактным является представление в виде родительских связей: в качестве корня мы выбираем одну из вершин и поддерживаем два вектора, индексированные именами вершин, в одном из них фиксируются родитель каждой вершины дерева, во втором указан вес ребра, ведущего из каждой вершины к ее родителю (диаграмма справа). Ориентация дерева (выбор корневой вершины) произвольна и не является свойством дерева MST. Мы можем выполнить преобразование любого из этих представлений в какое-либо другое за линейное время.

Этот программный код представляет типичный случай, когда в качестве корня дерева MST выбирается вершина **0**, при этом он не помещает фиктивное ребро **0-0** в список ребер дерева MST.

Оба эти преобразования тривиальны, в то же время возникает вопрос, как выполняется преобразование представления дерева MST в виде вектора указателей на ребра в представление в виде родительских связей? В нашем распоряжении имеются базовые инструментальные средства, позволяющие с такой же легкостью решить и эту задачу: мы преобразуем указанное выше представление в граф, воспользовавшись циклом, подобным предложенному выше (измененным таким образом, что функция **insert** вызывается для каждого ребра), с последующим выполнением поиска в глубину из любой вершины с целью вычисления представления дерева DFS в виде родительских связей за линейное время.

Короче говоря, несмотря на то, что выбор представления дерева MST производится из соображений удобства, мы заключаем все наши алгоритмы в класс **MST** обработки графов, который вычисляет приватный вектор **mst** указателей на ребра. В зависимости от потребностей приложений, мы можем реализовать для этого класса функции-элементы, которые возвращают этот вектор или предоставляют клиентским программам другую информацию о деревьях MST, однако мы не будем вдаваться в дальнейшие детали этого интерфейса, а отметим лишь, что в число инструментальных средств мы включаем функцию-элемент **show**, которая вызывает аналогичную функцию для каждого ребра дерева MST (см. упражнение 20.8).

Упражнения

▷ **20.8**. Разработайте класс **WeightedEdge** (взвешенное ребро), который реализует интерфейс **EDGE** (ребро) из программы 20.1 и содержит функцию-элемент **show**, которая производит распечатку ребер и их весов в формате, используемом в рисунках этой главы.

▷ **20.9**. Реализуйте класс **io** для взвешенных графов, в котором имеются функции-элементы **show**, **scan** и **scanEZ** (см. программу 17.4).

▷ **20.10**. Постройте АТД графа, который использует целочисленные веса и в то же время отслеживает минимальные и максимальные веса в графе и содержит функцию АТД, которая всегда возвращает веса, представляющие собой вещественные числа в диапазоне от 0 до 1.

▷ **20.11**. Спроектируйте интерфейс, подобный используемому в программе 20.1, который позволил бы клиентам и реализациям манипулировать типами данных **Edge** (но не указателями на них).

○ **20.12**. Разработайте реализацию интерфейса, предложенного вами в упражнении 20.11, который использует представление графа в виде минимальной матрицы весов, в которой функция итератора **nxt** использует информацию, неявно содержащуюся в индексах строк и столбцов, для создания типа данных **Edge**, что обеспечивает возможность возвращать его значения клиентской программе.

20.13. Реализуйте класс итератора с целью его использования в программе 20.5 (см. программу 20.4).

○ **20.14**. Разработайте реализацию интерфейса, построенного вами в упражнении 20.11, которая использует минимальное представление графа в виде списков смежных вершин, где узлы списка содержат вес и вершину назначения (но не вершину-источник), а функция итератора **nxt** использует неявную информацию для создания типа данных **Edge**, что обеспечивает возможность возвращать его значения клиентской программе.

20.15. Внесите изменения в генератор разреженных случайных графов из программы 17.12, которые позволили бы присваивать ребрам случайные веса (из диапазона от 0 до 1).

▷ **20.16**. Внесите изменения в генератор насыщенных случайных графов из программы 17.13, которые позволили бы присваивать ребрам случайные веса (из диапазона от 0 до 1).

20.17. Напишите программу, которая генерирует случайные взвешенные графы путем соединения вершин, упорядоченных в виде решетки $\sqrt{V}$ на $\sqrt{V}$, с соседними вершинами (как изображенная на рис. 19.3, но для случая неориентированного графа), при этом каждому ребру присваивается случайный вес (принимающий значение из диапазона от 0 до 1).

20.18. Напишите программу генерации случайных полных графов, ребрам которых присвоены веса, выбранные по распределению Гаусса.

● **20.19**. Напишите программу, которая генерирует V случайных точек на плоскости во время построения взвешенного графа за счет соединения каждой пары точек, расположенных друг от друга на расстоянии d и ближе, ребрами, весом которых является это расстояние (см. упражнение 17.74). Определите, какое должно быть установлено расстояние d, чтобы ожидаемое число ребер было равно E.

• **20.20**. Найдите крупный взвешенный динамический граф, возможно, карту с расстояниями, телефонную сеть со стоимостями переговоров или расписание авиарейсов с указанными стоимостями билетов.

20.21. Составьте матрицу размерами 8 на 8, содержащую представления родительских связей всех ориентаций дерева MST графа, который показан на рис. 20.1. Поместите представление дерева в виде родительских связей с корнем в вершине *i* в *i*-тую строку этой матрицы.

▷ **20.22**. Предположим, что конструктор класса **MST** генерирует представление дерева MST в виде вектора указателей на ребра с элементами от **mst[1]** до **mst[V]**. Добавьте функцию-элемент **ST** для клиентских программ (например, как в программе 18.3) такую, что **ST(v)** возвращает родителя вершины **v** в этом дереве (**v**, если она является корнем).

▷ **20.23**. В условиях предположений, сформулированных в упражнении 20.22, напишите функцию-элемент, которая возвращает суммарный вес дерева MST.

○ **20.24**. Предположим, что конструктор класса MST генерирует представление дерева MST в виде родительских связей в векторе **st**. Напишите программный код, который необходимо добавить в этот конструктор, чтобы можно было вычислять представление этого дерева в виде указателей на векторы ребер в элементах с индексами 1, ..., *V* приватного вектора **mst**.

▷ **20.25**. Определите класс **TREE** (дерево). Затем, исходя из предположений упражнения 20.22, напишите функцию-элемент, которая возвращает **TREE**.

20.2. Принципы, положенные в основу алгоритмов построения дерева MST

Задача определения дерева MST относится к одной из тех, которым в этой книге уделяется наибольшее внимание. Основные подходы к ее решению были получены задолго до разработки современных структур данных и современных технологий анализа рабочих характеристик алгоритмов, еще в те времена, когда определение дерева MST конкретного графа, содержавшего, скажем, тысячу ребер, представляло собой невыполнимую задачу. Как будет показано далее, некоторые новые алгоритмы определения MST отличаются от старых, главным образом, способами использования и реализации современных алгоритмов и структур данных в рамках базовых задач, которые (в совокупности с мощью современных компьютеров) дают возможность вычислять деревья MST, состоящие из миллионов и даже миллиардов ребер.

Одно из определяющих свойств дерева (см. раздел 5.4) заключается в том, что добавление ребра в дерево порождает уникальный цикл. Это свойство является определяющим для доказательства двух фундаментальных свойств деревьев MST, к изучению которых мы сейчас перейдем. Все алгоритмы, с которыми мы сталкиваемся, основаны на одном или двух таких свойствах.

Первое из этих свойств, на которое мы далее будем ссылаться как на *свойство сечения*, относится к идентификации ребер, которые должны входить в дерево MST заданного графа. Несколько основных термином из теории графов, определение которых мы дадим ниже, позволяют получить компактную формулировку этого свойства, что мы и сделаем.

Определение 20.2. Сечение (cut) *графа есть разделение множества всех вершин графа на два непересекающихся множества.* **Пересекающее ребро (crossing edge)** *есть ребро, которое соединяет вершину одного множества с вершиной другого множества.*

Иногда мы описываем сечение графа путем описания некоторого множества вершин графа, подразумевая под этим, что сечение содержит само это множество и его дополнение. По существу, мы используем сечения графа в тех случаях, когда оба указанных выше множества не пусты — в противном случае пересекающие ребра отсутствуют.

Свойство 20.1. (Свойство сечения). *При любом сечении графа каждое минимальное пересекающее ребро принадлежит некоторому дереву MST и каждое дерево MST содержит минимальное пересекающее ребро.*

Доказательство: Проведем доказательство от противного. Предположим, что e – минимальное пересекающее ребро, которое не содержится ни в одном MST, и пусть T есть некоторое дерево MST, которое не содержит минимального пересекающего ребра e. В любом случае T есть MST, которое не содержит минимального пересекающего ребра e. Теперь рассмотрим граф, полученный путем добавления ребра e в T. В этом графе имеется цикл, который содержит ребро e, и этот цикл должен, по меньшей мере, содержать еще одно пересекающее ребро, скажем, f, которое имеет вес, равный или больший веса e (поскольку e имеет минимальный вес). Мы можем получить остовное дерево равного или меньшего веса, если удалим f и добавим e, что противоречит условию минимальности T или предположению, что e не содержится в T. ■

Если все веса ребер графа различны, он обладает единственным деревом MST; свойство сечения свидетельствует о том, что кратчайшее пересекающее дерево для каждого сечения должно быть внутри дерева MST. Если имеют место равные веса ребер, мы можем получить несколько минимальных пересекающих ребер. По меньшей мере, одно из них входит в состав любого заданного дерева MST, в то время как другие могут в нем быть, а могут и не быть.

На рис. 20.5 представлены несколько примеров рассмотренного свойства сечения. Обратите внимание на то обстоятельство, что в рассматриваемом случае отсутствует требование того, что минимальное ребро должно быть *единственным* ребром MST, которое соединяет два эти множества; в самом деле, для обычных сечений существует несколько ребер, соединяющих вершину одного множества с вершиной другого. Если мы

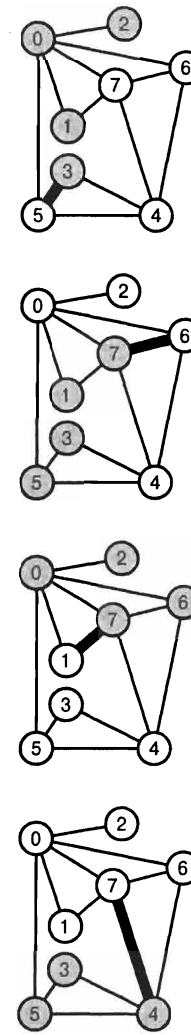


РИСУНОК 20.5. СВОЙСТВО СЕЧЕНИЯ

Приведенные здесь четыре примера служат иллюстрацией свойства 20.1. Если для одного множества вершин мы выберем серый цвет, а для другого такого множества белый, то самое короткое ребро, соединяющее серую вершину с белой, принадлежит дереву MST.

уверены в том, что существует одно такое ребро, мы можем разработать алгоритмы типа "разделяй и властвуй", в основу которых положен походящий выбор множеств, но сейчас нас интересуют совершенно другие проблемы.

Мы используем свойство сечения в качестве основы для алгоритмов поиска деревьев MST, оно может также служить *условием оптимальности*, которое характерно для деревьев MST. В частности, из выполнения условия оптимальности следует, что каждое ребро дерева MST есть минимальное ребро, пересекающее сечение, определенный вершинами двух поддеревьев, соединенных этим ребром.

Второе свойство, которое мы будем называть *свойством цикличности*, применяется с целью выявления ребер, которые не должны входить в дерево MST графа. Другими словами, если мы игнорируем эти ребра, то, тем не менее, не теряем возможности отыскать дерево MST.

Свойство 20.2. (Свойство цикличности). *Пусть задан граф G, рассмотрим граф G', который получается в результате добавления к графу G ребра e. Добавление ребра e в дерево MST графа G и удаление максимального ребра из полученного в результате этой операции цикла дает дерево MST графа G'.*

Доказательство: Если ребро *e* длиннее всех других ребер цикла, то оно не должно содержаться в дереве MST графа G' по свойству 20.1: удаление e из любого такого дерева MST разделит последнее на две части, а *e* не будет самым коротким ребром, соединяющим вершины каждой из полученных двух частей, поскольку это должно делать какое-то другое ребро цикла. И наоборот, пусть *t* есть максимальное ребро цикла, построенного в результате добавления ребра *e* в дерево MST графа *G*. Удаление ребра *t* приводит к разбиению исходного дерева MST на две части, а ребра графа *G*, соединяющие эти две части, не короче *t*; следовательно, *e* является минимальным ребром в G', которое соединяет вершины этих двух частей. Подграфы, индуцированные этими двумя подмножествами вершин, идентичны *G* и G', таким образом, дерево MST для G' состоит из ребра *e* и из деревьев MST этих двух подмножеств.

В частности, обратите внимание на то, что если ребро *e* есть максимальное в цикле, то мы показали, что существует дерево MST графа G', которое не содержит *e* (дерево MST графа *G*). ■

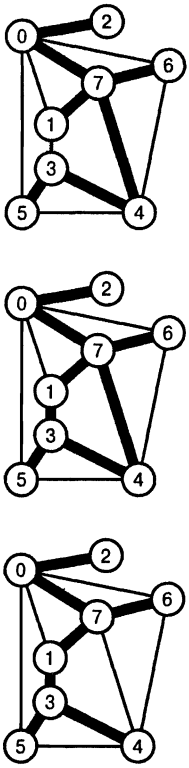


РИС. 20.6. СВОЙСТВО ЦИКЛА
Добавление ребра 1-3 в граф, показанный на рис. 20.1, приводит к тому, что дерево MST перестает быть таковым (диаграмма вверху). Чтобы найти дерево MST нового графа, мы добавляем новое ребро в MST старого графа, которое порождает цикл (диаграмма в центре). Удаляя самое длинное ребро цикла (4-7), получаем MST нового графа (внизу). Одним из способов проверить, что остовное дерево минимально, заключается в проверке того факта, что каждое ребро, не входящее в это дерево MST, имеет наибольший вес в цикле, который оно образует с тремя другими ребрами. Например, на диаграмме в нижней части рисунка ребро 4-6 имеет максимальный вес в цикле 4-6-7-1-3-4.

Рисунок 20.6 служит иллюстрацией рассматриваемого свойства цикла. Обратите внимание на то обстоятельство, что процесс, предусматривающий выбор произвольного остовного дерева, добавление ребра, обеспечивающего построение цикла, и последующее удаление максимального в этом цикле ребра, приводит к получению остовного дерева, вес которого меньше или равен весу исходного остовного дерева. Вес нового дерева будем меньше веса исходного в том и только том случае, когда добавляемое ребро короче, чем некоторое из ребер построенного цикла.

Свойство цикличности служит также основой условия оптимальности, которое характерно для деревьев MST: из него следует, что каждое ребро любого графа, не содержащегося в заданном MST, есть максимальное ребро цикла, который оно образует с ребрами MST.

Свойства сечения и цикличности образуют основу для классических алгоритмов, которые будут исследоваться применительно к задаче определения дерева MST. Мы будем рассматривать ребра по одному за раз, используя сечения с целью проверки их пригодности для использования в качестве ребер дерева MST, или свойство цикличности с целью определения возможности их отбрасывания за ненадобностью. Эти алгоритмы различаются по их способности эффективно идентифицировать сечения и циклы.

Первый подход к поиску дерева MST, который мы намерены исследовать во всех подробностях, заключается в постепенном построении MST, добавляя одно ребро за раз: мы начинаем построение с произвольной вершины и рассматриваем ее как дерево MST, состоящее из одной вершины, затем добавляем к нему $V-1$ вершин, при этом каждый раз выбираем минимальное ребро, которое соединяет вершину, уже включенную в дерево MST, с вершиной, которая еще не содержится в MST. Этот метод известен как *алгоритм Прима (Prim's algorithm)*, и он будет рассматриваться в разделе 20.3.

Свойство 20.3. *Алгоритм Прима вычисляет дерево MST любого связного графа.*

Доказательство: В соответствии с подробным описанием, изложенным в разделе 20.2, рассматриваемый метод есть обобщенный метод поиска на графе. Непосредственно из свойства 18.12 вытекает тот факт, что выбранные ребра образуют остовное дерево. Чтобы показать, что они представляют собой дерево MST, применим механизм сечения с использованием для этих целей вершин, входящих в MST, в качестве первого подмножества и вершин, не входящих в MST, в качестве второго подмножества. ■

Еще один подход к вычислению дерева MST предусматривает многократное применение свойства цикличности: мы добавляем ребра по одному за раз во мнимое дерево MST, с удалением максимального ребра из цикла, если таковой был образован (см. упражнения 20.33 и 20.71). Этому методу уделяется меньше внимание, чем другим рассмотренным нами алгоритмам, в силу трудностей, связанных с поддержкой структуры данных, которые обеспечивают реализацию операции "удалить самое длинное ребро цикла".

Второй подход отыскания дерева MST, который будет изучаться во всех подробностях, предусматривает обработку ребер в порядке возрастания их длины (первым обрабатывается наиболее короткое) с добавлением в MST каждого ребра, которое не образует цикла с ребрами, включенными в MST раньше; при этом процесс останавливается после того, как будут добавлены $V-1$ ребер. Этот метод известен как *алгоритм Крускала (Kruskal's algorithm)*, который подробно рассматривается в разделе 20.4.

Свойство 20.4. *Алгоритм Крускала вычисляет дерево MST любого связного графа.*

Доказательство: Мы покажем методом индукции, что этот алгоритм поддерживает лес поддеревьев MST. Если следующее рассматриваемое ребро приводит к образованию цикла, то это дерево представляет собой максимальное дерево цикла (поскольку все другие ребра были выбраны раньше в порядке, установленном сортировкой), так что даже если его проигнорировать, дерево MST все равно сохраняется в соответствии со свойством цикла. Если следующее рассматриваемое ребро не приводит к образованию цикла, примените свойство сечения, воспользовавшись сечением, определенным множеством вершин, связанных с одной из вершин этого ребра ребрами дерева MST (и их дополнениями). Поскольку рассматриваемое ребро не образует цикла, это всего лишь пересекающее ребро, а поскольку мы рассматриваем ребра в порядке, установленном сортировкой, это ребро минимальное и, в силу этого обстоятельства, содержится в дереве MST. Основанием для индукции служат V отдельных вершин; как только мы выберем $V-1$ ребер, мы получаем одно дерево (дерево MST). Ни одно из неисследованных ребер не короче никакого ребра MST, и все это вместе образует цикл, следовательно, отбросив все остальных ребра, по свойству цикличности мы получаем дерево MST. ■

Третий подход к построению дерева MST, который мы сейчас рассмотрим подробно, известен как *алгоритм Борувки (Boruvka's algorithm)*, он подробно исследуется в разделе 20.4. На первом шаге в дерево MST добавляются ребра, которые соединяют каждую вершину с ее ближайшим соседом. Если веса ребер различны, этот шаг порождает лес из поддеревьев дерева MST (мы сейчас докажем этот факт и рассмотрим усовершенствование, которое выполняет эту задачу, даже когда в какой-то момент появляются ребра с равными весами). Затем мы добавляем в дерево MST ребра, которые соединяют каждую вершину с ее ближайшим соседом (минимальное ребро, соединяющее вершину одного дерева с вершиной другого), и повторяем этот процесс до тех пор, пока у нас не останется только одно единственное дерево.

Свойство 20.5. *Алгоритм Борувки вычисляет дерево MST для любого связного графа.*

Сначала предположим, что веса всех ребер различны. В этом случае у каждой вершины имеется единственный ближайший сосед, дерево MST уникально, и мы знаем, что каждое ребро добавлено в соответствие со свойством сечения (это самое короткое ребро из тех, что пересекают сечение и соединяют некоторую вершину одного множества со всеми вершинами другого). Поскольку каждая вершина выбрана из уникального дерева MST, циклов в этом случае может и не быть, каждое добавленное ребро соединяет два дерева из леса, образуя при этом дерево больших размеров, и процесс продолжается до тех пор, пока не останется единственное дерево, а именно – дерево MST.

Если же имеются одинаковые ребра, может оказаться, что ближайших соседей будет несколько, и, возможно, появится цикл, когда мы добавляем ребро к ближайшему соседу (см. рис. 20.7). Другими словами, мы можем включить для некоторой вершины два ребра из множества минимальных пересекающих ребер, в то время как дереву MST принадлежит одно. Во избежание подобных ситуаций, нам необходимо соответствующее правило разрыва связей. Одной из возможностей является выбор на множестве минимальных соседей вершины с наименьшим номером. Тогда существование любого цикла приводит к противоречию: если бы v была вершиной с наибольшим номером в цикле, то ни одна из вершин, соседних с вершиной v, не выбрала бы ее как ближайшего соседа, в этом случае вершина v должна выбрать только одного из своих соседей с минимальным номером, а не двух. ■

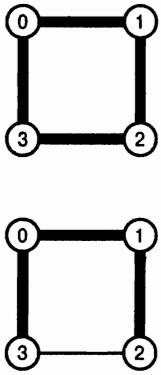


РИСУНОК 20.7. ЦИКЛЫ АЛГОРИТМА БОРУВКИ

В представленном на рисунке графе имеется четыре вершины, и все четыре показанных здесь ребра имеют одну и ту же длину. Когда мы соединяем каждую вершину с ближайшим соседом, мы должны решить, какое ребро выбрать из множества минимальных ребер. В верхнем примере мы выбираем 1 из вершины 0, 2 из 1, 3 из 2 и 0 из 3, что приводит к образованию цикла в предполагаемом дереве MST. Каждое из ребер входит в некоторое дерево MST, но не все их них входят в каждое MST. Чтобы решить эту задачу, принимаем к исполнению правило разрыва связей, как показано в нижней части рисунка: выбираем минимальное ребро, ведущее в вершину с наименьшим индексом. Таким образом, из 1 мы выбираем 0, 0 из 1, 1 из 2 и 0 из 3, что в результате и дает дерево MST. Цикл прерван, поскольку вершина с максимальным индексом 3 не выбиралась ни из одного из ее соседей 2 или 1, а она может выбрать только одного из них (0).

Все эти алгоритмы суть специальные случаи общей парадигмы, которой все еще продолжают пользоваться исследователи, стремящиеся открыть новые алгоритмы построения MST. В частности, мы можем *в произвольном порядке* применять свойство сечения для выбора того или иного ребра в качестве ребра MST или свойство цикла для обоснования отказа включить ребро в MST и продолжать эту процедуру до тех пор, пока станет невозможным увеличение ни числа принятых, ни числа отвергнутых ребер. На этой стадии при любом делении множества вершин графа на два подмножества имеется ребро MST, соединяющее эти два подмножества (следовательно, применение свойства сечения не приводит к увеличению числа ребер дерева MST), и все циклы графа содержат, по меньшей мере, одно ребро, не входящее в MST (следовательно, применение свойства сечения не приводит к увеличению числа ребер дерева MST). Оба эти свойства в совокупности позволяют вычислить полное дерево MST.

Точнее, все три алгоритма, рассматриваемые нами во всех подробностях, могут быть объединены в один обобщенный алгоритм, выполнение которого мы начинаем с того, что выбираем лес поддеревьев MST, состоящих из одной вершины (и не содержащих никаких ребер), выполняем процедуру добавления в дерево MST минимального ребра, соединяющего два любых поддерева леса, и повторяем эту процедуру $V - 1$ раз до тех пор, пока не останется единственное дерево MST. В соответствии со свойством сечения, ни одно из ребер, которое порождает цикл, не нужно подвергать анализу для его включения в дерево MST, поскольку одно из ребер на том или ином предшествующем шаге было минимальным ребром, пересекающим некоторое сечение между поддеревьями MST, каждое из которых содержало свои вершины. По условиям алгоритма Прима, мы увеличиваем число ребер дерева по одному за один раз; по условиям алгоритмов Крускала и Борувки, мы объединяем деревья некоторого леса.

Согласно описанию, приведенному в этом разделе и в классической литературе, для выполнения указанных выше алгоритмов должны быть разработаны высокоуровневые абстрактные операции, такие как:

- Отыскание минимального ребра, соединяющего два поддерева.
- Определение, будет ли образован цикл при включении конкретного ребра.
- Удаление самого длинного ребра цикла.

Наша задача заключается в разработке алгоритма и структуры данных, которые позволили бы эффективно реализовать перечисленные операции. К счастью, эта задача предоставляет нам возможность эффективно использовать базовые алгоритмы и структуры данных, которые рассматривались ранее в этой книге.

С алгоритмами на деревьях MST связана долгая и интересная история, которая продолжает развиваться; мы будем к ней возвращаться по мере более близкого знакомства с ними. Наше растущее понимание различных методов реализации базовых абстрактных операций породила некоторую путаницу, окутывающую дату происхождения алгоритма. В самом деле, эти методы были впервые описаны в двадцатых годах прошлого столетия, раньше, чем появились компьютеры в том виде, в каком мы их знаем, и раньше, чем были получены фундаментальные алгоритмы сортировки и другие алгоритмы. Как нам теперь известно, выбор базовых алгоритмов и структур данных оказывает существенное влияние на производительность наших программ, даже когда мы получаем программную реализацию базовых схем вычислений. За последние годы исследования в области задач, для решения которых используются деревья MST, концентрируются на таких проблемах реализации, которые допускают использование классических схем. В целях сохранения последовательности и четкости изложения, мы будем называть базовые подходы употребляемыми здесь именам их исследователей, хотя абстрактные версии этих алгоритмов были исследованы намного раньше, а современные реализации используют алгоритмы и структуры данных, изобретенные намного позже того, когда эти методы были открыты.

До сих пор нерешенной остается задача поиска алгоритма построения деревьев MST за линейное время. Как мы сможем убедиться далее, многие реализации, которые мы будем изучать, выполняются за линейное время в широком диапазоне ситуаций, имеющих место на практике, однако в худшем случае эта зависимость перестает быть линейной. Разработка алгоритмов, которые гарантированно выполняются за линейное время на разреженных графах, все еще остается недостижимой целью для исследователей.

Помимо естественного поиска алгоритма решения этой фундаментальной задачи, изучение алгоритмов построения дерева MST подчеркивает важность правильного понимания базовых рабочих характеристик фундаментальных алгоритмов. По мере того, как программисты продолжают использовать алгоритмы и структуры данных все более высокого уровня абстракции, ситуация такого рода становится все более привычной. Полученные нами реализации АТД обладают различными рабочими характеристиками – по мере того, как мы используем АТД в качестве компонент алгоритмов решения задач еще более высокого уровня абстракции, возможности умножаются. В самом деле, мы часто используем алгоритмы, которые основаны на использовании деревьев MST и подобных абстракций (что становится возможным благодаря эффективным реализациям, которые будут изучаться далее в этой главе), в качестве средств решения другие задач на еще более высоком уровне абстракции.

Упражнения

▷ **20.26**. Присвойте метки, соответственно, от 0 до 5 следующим точкам на плоскости:

(1,3) (2,1) (6,5) (3,4) (3,7) (5,3).

Принимая длину ребра в качестве его веса, найдите дерево MST для графа, заданного множеством ребер

1-0 3-5 5-2 3-4 5-1 0-3 0-4 4-2 2-3.

20.27. Предположим, что граф образуется ребрами с различными весами. Должно ли его самое короткое ребро принадлежать дереву MST? Докажите его принадлежность дереву MST или дайте встречный пример.

20.28. Ответьте на вопрос упражнения 20.27 применительно к *самому длинному ребру* графа.

20.29. Приведите встречный пример, показывающий, почему не гарантирует нахождение дерева MST следующая стратегия: "начните с любой вершины и рассматривайте ее как дерево MST с одной вершиной, затем добавляем к ней $V-1$ вершину, при этом всегда должно выбираться следующее минимальное ребро, инцидентное вершине, которая была последней включена в дерево MST".

20.30. Предположим, что все ребра заданного графа обладают различными весами. Должно ли каждое минимальное в каждом цикле ребро принадлежать дереву MST? Докажите это утверждение или дайте встречный пример.

20.31. Пусть задано дерево MST графа G, предположим, что из графа G удалено некоторое ребро. Опишите, как найти дерево MST нового графа за время, пропорциональное числу ребер графа G.

20.32. Постройте дерево MST, которое получается после многократного применения свойства цикла к графу, изображенному на рис. 20.1, при этом ребра выбираются в заданном порядке.

20.33. Докажите, что многократное применение свойства цикла приводит к построению дерева MST.

20.34. Опишите, как адаптировать алгоритмы из данного раздела (при необходимости) к проблеме отыскания *минимального остовного леса* для взвешенного графа (объединение деревьев MST его связных компонентов).

20.3. Алгоритм Прима и поиск по приоритету

С точки зрения реализации алгоритм Прима, по-видимому, является простейшим из всех алгоритмов построения дерева MST, ему отдают предпочтение в случае насыщенных графов. Мы поддерживаем сечение графа, состоящее из *древесных* вершин (те, которые выбраны для представления дерева MST) и *недревесных* вершин (те, которые не попадают в дерево MST). Мы начинаем с того, что помещаем произвольную вершину в дерево MST, затем помещаем в MST минимальное пересекающее ребро (которое превращает недревесную вершину дерева MST в древесную) и повторяем подобную операцию $V-1$ раз, пока все вершины не окажутся в дереве.

Реализация алгоритма Прима "в лоб" следует непосредственно из этого описания. Чтобы найти очередное ребро для включения его в дерево MST мы должны исследовать все ребра, которые выходят из древесной вершины в недревесную вершину, затем выбрать

из них самое короткое и поместить его в дерево MST. Мы здесь не будем рассматривать соответствующую программную реализацию по причине ее исключительно высокой стоимости (см. упражнения 20.35–20.37). Применение простых структур данных, позволяющих устранить повторные вычисления, приводят упрощению алгоритма и повышению его быстродействия.

Добавление вершины в дерево MST представляет собой инкрементное изменение: чтобы реализовать алгоритм Прима, мы внимательно исследуем природу такого инкрементного изменения. При этом главным образом мы заинтересованы в том, чтобы найти кратчайшее расстояние каждой недревесной вершины до дерева. Когда мы присоединяем вершину v к дереву, единственное изменение, касающееся недревесной вершины w, состоит в том, что с добавлением вершины v в дерево w становится ближе к этому дереву, нежели раньше. Короче говоря, нам не надо проверять расстояние от вершины w до каждой вершины дерева – нам нужно знать минимальное расстояние в каждый момент и проверять, вызывает ли добавление вершины v в дерево необходимость изменить это минимальное расстояние.

Ориентированный чертеж возрастающего дерева MST показан справа от каждого чертежа графа. Ориентация есть побочный продукт изучаемого нами алгоритма: в общем случае само дерево MST мы рассматриваем как множество ребер, неупорядоченное и неориентированное.

Для реализации этой идеи нам потребуются такие структуры данных, которые предоставляли бы следующую информацию:

- Ребра дерева.
- Самое короткое дерево, соединяющее недревесные вершины с деревом.
- Длина этого ребра.

Простейшей реализацией каждой из этих структур данных будет вектор, индексированный именами вершин (мы можем использовать этот вектор для хранения древесных ребер, индексируя их именами вершин по мере их добавления в дерево). Программа 20.6 представляет собой реализацию алгоритма Прима для насыщенных графов. Она использует для этих структур данных, соответственно, векторы **mst**, **fr** и **wt**.

После включения нового ребра (и вершины) в дерево, мы должны выполнить еще две процедуры:

- Проверить, приблизит ли добавление нового ребра какую-либо из недревесных вершин к искомому дереву.
- Найти следующее ребро, подлежащее включению в искомое дерево.

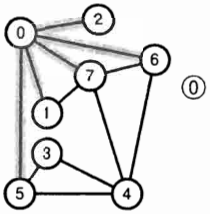
Реализация программы 20.6 решает обе эти задачи в течение одного просмотра недревесных вершин. Сначала она обновляет содержимое векторов **wt[w]** и **fr[w]**, если **v-w** приближает **w** к дереву, после чего обновляется текущий минимум, если **wt[w]** (длина **fr[w]**) показывает, что **w** ближе к дереву, чем любая другая недревесная вершина с меньшим индексом).

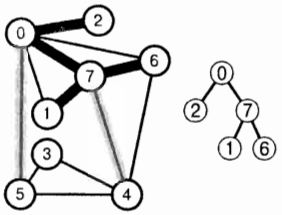
Свойство 20.6. *Используя алгоритм Прима, можно найти дерево MST насыщенного графа за линейное время.*

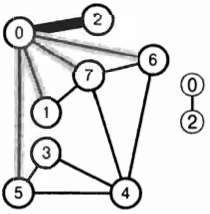
Доказательство: Анализ программы 20.6 показывает, что время ее выполнения пропорционально V^2 и поэтому линейно для случаев насыщенных графов. ■

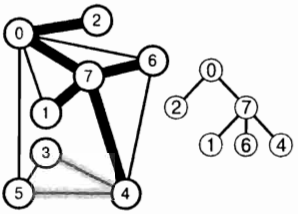
РИСУНОК 20.8. АЛГОРИТМ ПРИМА ПОСТРОЕНИЯ ДЕРЕВА MST

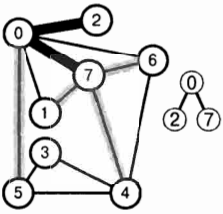
*Первое действие при вычислении дерева MST по алгоритму Прима предусматривает включение в это дерево вершины **0**. Затем мы находим все ребра, которые соединяют **0** с другими вершинами (еще не включенными в это дерево), и выбираем из них самое короткое (слева вверху). Ребра, соединяющие древесные вершины с недревесными, заштрихованы (бахрома), их список приводится под каждым чертежом графа. Чтобы не усложнять чертеж, мы перечисляем ребра, образующие бахрому, в порядке возрастания их длины, так что самое короткое ребро в этом списке следует первым. В различных реализациях алгоритма Прима используются различные структуры данных для поддержания этого списка и определения минимального ребра. Второе действие заключается в переносе самого короткого ребра **0-2** (вместе с вершиной, в которую оно нас приводит) из бахромы в дерево (вторая диаграмма сверху слева). В-третьих, мы переносим ребро **0-7** из бахромы в дерево и заменяем ребро **0-1** на **7-1**, а ребро **0-6** на **7-6** в бахроме (поскольку с включением вершины **7** в дерево **1** и **6** становятся ближе к дереву), и включаем ребро **7-4** в бахрому (поскольку добавление вершины **7** в дерево превращает **7-4** в ребро, которое соединяет древесную вершину с недревесной) (третья диаграмма сверху слева). Далее, мы переносим ребро **7-1** в дерево (диаграмма слева внизу). В завершение вычислений мы исключаем ребра **7-6**, **7-4**, **4-3** и **3-5** из очереди, обновляем бахрому после каждой вставки с тем, чтобы зафиксировать обнаруженные более короткие или новые пути (диаграммы справа сверху вниз).*

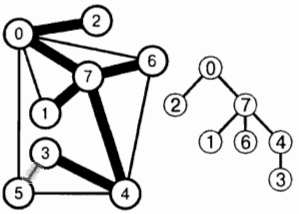


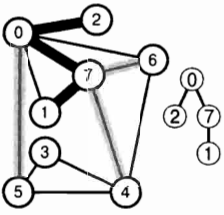


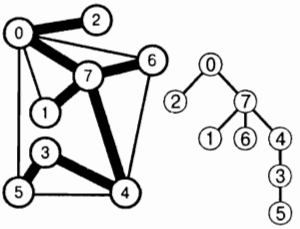












	0	1	2	3	4	5	6	7
mst	0	7-1	0-2	4-3	7-4	3-5	7-6	0-7
wt		.21	.29	.34	.46	.18	.25	.31

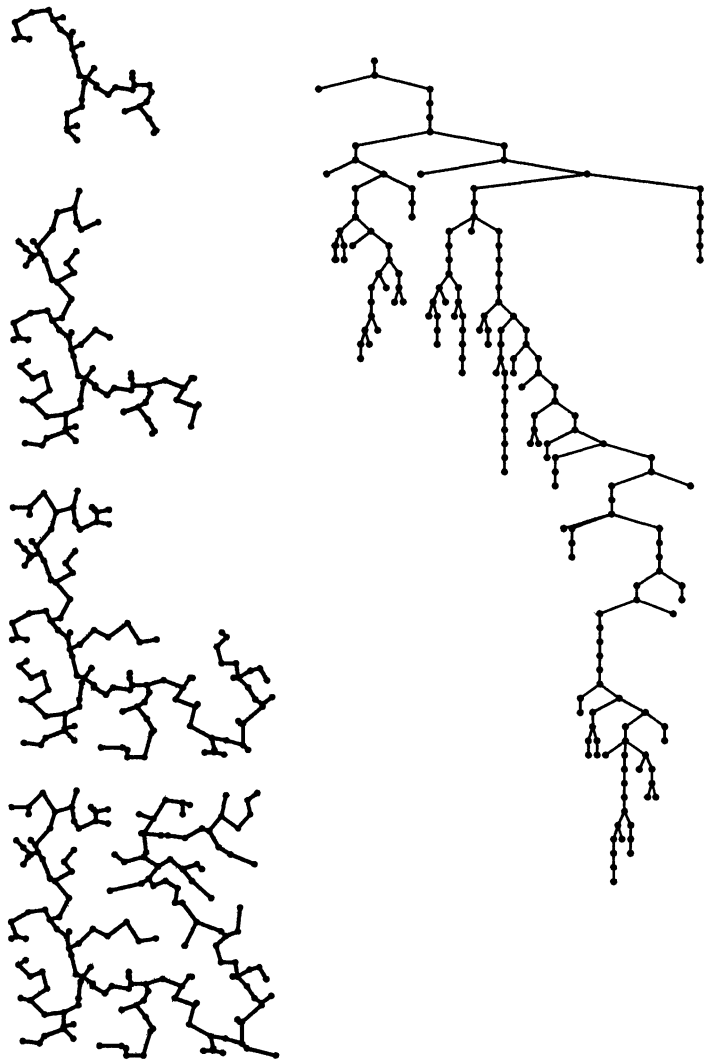


РИСУНОК 20.9. АЛГОРИТМ ПРИМА, ОБЕСПЕЧИВАЮЩИЙ ПОСТРОЕНИЕ ДЕРЕВА MST

Эта последовательность показывает, как растет дерево MST по мере того, как алгоритм Прима обнаруживает 1/4, 1/2, 3/4 всех ребер и все ребра дерева MST (сверху вниз). Ориентированное представление полного дерева MST показано справа.

На рис. 20.8 показан пример построения дерева MST с помощью алгоритма Прима; на рис. 20.9 показано, как развертывается дерево MST на примере более крупного графа.

В основу программы 20.6 положен тот факт, что мы можем совместить выполнение операций *поиска минимального ребра (find minimum)* и обновления в одном цикле, в котором исследуются все недревесные ребра. В насыщенных графах число ребер, которые нам, возможно, придется исследовать с целью обновления расстояния от недревесных вершин до дерева, пропорционально V, следовательно, просмотр всех недревесных ребер с тем, чтобы найти ту из них, которая ближе всех расположена к дереву, не влечет за собой чрезмерных дополнительных расходов. В то же время в случае разреженного графа мы можем рассчитывать на выполнение менее V шагов для выполнения каждой из этих операций. Основная трудность в реализации этой стратегии, которой мы далее воспользуемся, заключается в том, чтобы получить возможность работать с множеством ребер, которые представляют собой потенциальные кандидаты для включения в дерево MST, – с множеством ребер, которое далее мы будем называть *бахромой (fringe)*. Число ребер в бахроме обычно существенно меньше, чем число недревесных ребер, поэтому мы можем скорректировать описание алгоритма следующим образом. Начиная с петли исходной вершины из бахромы и пустого дерева MST, до тех пор, пока бахрома не опустеет, мы будем выполнять следующие операции:

> Переносим минимальное ребро (т.е. ребро минимальной длины или веса) из бахромы в дерево. Наносим визит в вершину, в которую оно ведет, и помещаем в бахрому любые ребра, которые ведут из этого дерева в одну из недревесных вершин, отбрасывая ребро большей длины, если два ребра в бахроме указывают на одну и ту же вершину.

Из этой формулировки ясно, что алгоритм Прима есть ни что иное, как обобщенный поиск на графе (см. раздел 18.8), в котором бахрома представлена в виде очереди с приоритетами, в основу которой заложена операция *удалить минимальное ребро (remove the minimum)* (см. главу 9). Мы будем называть обобщенный поиск на графе с очередью с приоритетами *поиском PFS* (priority-first search – поиск по приоритетам). Используя вес в качестве приоритетов, поиск по приоритетам реализует алгоритм Прима.

Программа 20.6. Алгоритм Прима, реализующий построение дерева MST

Реализация алгоритма Прима представляет собой метод, которому отдается предпочтение при работе с насыщенными графами и который может быть использовано для любого представления графа, поддерживающее функцию **edge**, выполняющую проверку существования заданного ребра. Внешний цикл обеспечивает приращение дерева MST посредством выбора минимального ребра, пересекающего сечение между вершинами, содержащимися в дереве MST, и вершинами, не входящими в это дерево. Цикл вершины **w** отыскивает минимальное ребро и в то же время (если **w** не содержится в MST) сохраняет неизменным условие, согласно которому ребро **fr[w]** является самым коротким (с весом **wt[w]**) ребром, ведущим из **w** в MST.

Результатом вычислений является вектор указателей на ребра. Первый указатель (**mst[0]**) не используется, остальные (от **mst[1]** до **mst[G.V()]** содержат дерево MST связной компоненты графа, которая содержит вершину **0**.

```
template <class Graph, class Edge> class MST
{ const Graph &G;
  vector<double> wt;
  vector<Edge *> fr, mst;
```

```
public:
  MST(const Graph &G) : G(G),
    mst(G.V()), wt(G.V(), G.V()), fr(G.V())
    { int min = -1;
      for (int v = 0; min != 0; v = min)
      {
        min = 0;
        for (int w = 1; w < G.V(); w++)
           if (mst[w] == 0)
           { double P; Edge* e = G.edge(v, w);
              if (e)
                if ((P = e->wt()) < wt[w])
                  { wt[w] = P; fr[w] = e; }
                if (wt[w] < wt[min]) min = w;
           }
        if (min) mst[min] = fr[min];
      }
    }
void show()
  { for (int v = 1; v < G.V(); v++)
      if (mst[v]) mst[v]->show();}
};
```

Эта формулировка учитывает важное замечание, которое мы сделали выше в разделе 18.7 в связи с реализацией поиска в ширину. Еще более простой общий подход заключается в том, чтобы просто хранить все ребра, инцидентные вершинам дерева, возлагая на механизм очередей с приоритетами обязанность отыскать наиболее короткое ребро и игнорировать более длинные ребра (см. упражнение 20.41). Как мы убедились в случае поиска в ширину, этот подход непривлекателен тем, что бахрома, как структура данных, без особой необходимости загромождается ребрами, которые никогда не попадут в дерево MST. Размеры бахромы могут возрастать пропорционально числу ребер E (со всеми затратами, вытекающими из необходимости содержать бахрому подобных размеров), в то время как подход с использованием поиска по приоритету дает гарантию того, что бахрома никогда не будет содержать более V вершин.

Как и в случае реализации алгоритма в общей форме, в нашем распоряжении имеется несколько подходов для обеспечения интерфейсов с АТД, реализующих очередь по приоритету. Один из подходов заключается в том, чтобы использовать очередь ребер, упорядоченную по их приоритетам, аналогично обобщенному поиску на графах, реализованному нами в виде программы 18.10. Программа 20.7 представляет собой реализацию, которая, по существу, эквивалентна программе 18.10, но в то же время она использует подход, в основу которого положены вершины, благодаря чему она может использовать индексированную очередь по приоритетам, в соответствии с изложенным в разделе 9.6. (Мы рассматриваем полную реализацию специального интерфейса с очередью с приоритетами, используемую программой 20.7, в программе 20.10, текст которой приводится в конце данной главы.) Будем называть *краевыми вершинами (fringe vertices)* подмножество недревесных вершин, которые соединены посредством ребер из бахромы с вершинами дерева, и будем использовать те же векторы **mst**, **fr** и **wt**, индексированные именами вершин, какие применялись в программе 20.6. Очередь с приоритетами содержит индекс каждой краевой вершины (этот элемент очереди обеспечивает доступ к самому короткому ребру, соединяющему краевую вершину с деревом), а также длину этого ребра во втором и третьем векторах.

Первое обращение к функции **pfs** (поиск по приоритету) в конструкторе программы 20.7 находит MST в связной компоненте, содержащей вершину **0**, а последующие обращения находят MST в других связных компонентах. Таким образом, этот класс фактически находит минимальные остовные леса в графах, не относящихся к числу связных (см. упражнение 20.34).

Свойство 20.7. *Использование реализации алгоритма Прима с поиском по приоритету, в котором для реализации очереди с приоритетами применяется полное бинарное дерево, позволяет вычислить дерево MST за время, пропорциональное* $E \lg V$.

Доказательство: Этот алгоритм напрямую реализует обобщенную идею алгоритма Прима (добавлять в дерево MST в качестве очередного минимальное ребро, которое соединяет некоторую вершину MST с вершиной, не входящей в MST). Каждая операция для очереди с приоритетами требует выполнения действий, число которых не превосходит значения $\lg V$. Каждая вершина выбирается при помощи операции *удалить минимальное ребро (remove the minimum)*; при этом в худшем случае каждое ребро может потребовать выполнения операции *изменить приоритет (change priority)*. ■

Поиск по приоритету представляет собой подходящее обобщение поиска ширину и поиска в глубину, ибо эти методы также могут быть получены за счет соответствующего выбора установок. Например, мы можем (в какой-то степени искусственно) использовать переменную **cnt** для присвоения уникального приоритета **cnt++** каждой вершине, когда помещаем ее в очередь по приоритетам. Если мы определим, что **P** есть **cnt**, мы получаем нумерацию узлов при обходе в прямом порядке и поиск в глубину, поскольку вновь встреченные узлы имеют наивысший приоритет. Если мы определим, что **P** есть **V-cnt**, мы получаем поиск в ширину, поскольку наивысший приоритет в этом случае имеют старые узлы. Присвоение таких значений приоритетов приводят к тому, что очереди с приоритетами ведут себя, соответственно, как стеки и как собственно очереди. Такая эквивалентность представляет чисто академический интерес, поскольку операции на очередях по приоритету не обязательны для поиска в глубину и для поиска в ширину. Кроме того, как уже отмечалось в разделе 18.8, формальное доказательство эквивалентности требует строгого соблюдения правила замены, чтобы получить ту же последовательность вершин как результат выполнения классических алгоритмов.

Как мы сможем убедиться далее, поиск по приоритетам охватывает поиск в глубину, поиск в ширину и алгоритм прима, обеспечивающий построение дерева MST, а также несколько других классических алгоритмов. Таким образом, время выполнения всех этих алгоритмов зависит от АТД очереди с приоритетами. В самом деле, мы приходим к общему результату, который охватывает не только две реализации алгоритма Прима, с которыми мы ознакомились в этом разделе, но также и широкий класс фундаментальных алгоритмов обработки графов.

Свойство 20.8. *Для всех графов и приоритетных функций можно вычислить остовное дерево при помощи поиска по приоритетам за линейное время плюс время, пропорциональное продолжительности выполнения* V *операций* **вставок** *(insert),* V *операций* **удалить минимальное ребро** *(delete the minimum) и* E *операций* **уменьшить ключ** *(decrease key) в очереди с приоритетами, размер которой не превышает* V.

Доказательство: Доказательство свойства 20.7 устанавливает этот результат в более общей форме. Мы должны проверить все ребра графа; отсюда следует условие "линейного времени". Рассматриваемый алгоритм никогда не увеличивает приоритет (он изменяет приоритет в сторону уменьшения). За счет более точного описания того, что нам нужно от АТД очереди с приоритетами (*уменьшить ключ*, а не обязательно *изменить приоритет*), мы усиливаем формулировку требований к рабочим характеристикам алгоритма. ■

В частности, использование реализации очереди с приоритетами на неупорядоченном массиве позволяет получить оптимальное решение для насыщенных графов, которые показывают ту же производительность в худшем случае, что и классическая реализация алгоритма Прима (программа 20.6). То есть, свойства 20.6 и 20.7 представляют собой частные случаи свойства 20.8; на протяжении этой книги мы рассмотрим другие многочисленные алгоритмы, которые существенно отличаются друг от друга только в части выбора функций назначения приоритетов и реализации очередей с приоритетами.

Свойство 20.7 представляет собой важный общий результат: устанавливаемая им временная граница есть верхняя граница для худшего случая, которая гарантирует для широкого класса задач обработки графов производительность, отличающуюся от оптимальной (линейное время) не более чем в $\lg V$ раз. Однако существуют две причины, в силу которых оно дает в некотором смысле пессимистическую оценку производительности на многих видах графов, с которыми нам приходится сталкиваться на практике. Во-первых, граница $\lg V$ для операций с очередями с приоритетами сохраняется только в тех случаях, когда количество вершин в бахроме пропорционально V, и даже в таком случае это всего лишь верхняя граница. В случае реального графа из практического приложения бахрома может быть небольшим множеством (см. рис. 20.10 и 20.11), а на выполнение реализаций некоторых операций на очереди с приоритетами будет затрачено существенно меньше, чем $\lg V$ действий. Хотя этот эффект и заметен, тем не менее он определяет только небольшую составляющую постоянного коэффициента, на который умножается время выполнения; например, доказательство того, что множество составных ребер не может содержать более $\sqrt{V}$ вершин улучшит граничное значение всего лишь в два раза. Что еще более важно, в общем случае мы выполняем намного меньше, чем E,

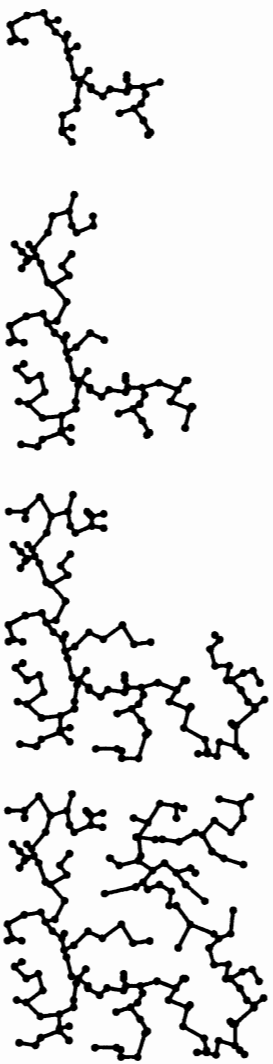


РИСУНОК 20.10. РЕАЛИЗАЦИЯ АЛГОРИТМА ПРИМА, ОБЕСПЕЧИВАЮЩЕГО ПОСТРОЕНИЕ ДЕРЕВА MST С ИСПОЛЬЗОВАНИЕМ ПОИСКА ПО ПРИОРИТЕТАМ.

Используя поиск по приоритетам, алгоритм Прима выполняет обработку вершин и ребер, наиболее близких к дереву MST (показаны серым цветом).

операций *уменьшить ключ*, поскольку мы выполняем эту операцию, только когда находим ребро, ведущее в узел, содержащийся в бахроме, которое короче известного на текущий момент ребра такого же типа. Такое событие происходит довольно редко: большая часть ребер не оказывает на очередь с приоритетами никакого влияния (см. упражнение 20.40). Целесообразно в большинстве случаев рассматривать поиск по приоритету как линейный по времени выполнения алгоритм при условии, что $V \lg V$ значительно больше E.

АДТ очереди с приоритетами и абстракции обобщенного поиска на графах позволяют проследить связи между различными алгоритмами. Поскольку эти абстракции (и программные механизмы, обеспечивающие поддержку их использования) были разработаны спустя много лет после открытия базовых методов, соответствие алгоритмов их классическим описаниям может интересовать разве что историков. Тем не менее, знания всех основных исторических фактов полезно, когда мы сталкиваемся с описаниями алгоритмов построения дерева MTS в публикациях по научным исследованиям или в других текстах, а понимание того, как эти немногочисленные простые абстракции связывают многих исследователей, разделенных во времени многими десятилетиями, служит убедительным доказательством их значимости и могущества. Учитывая вышесказанное, кратко рассмотрим здесь происхождение этих алгоритмов.

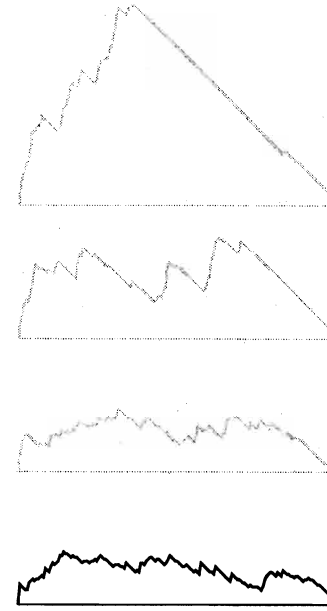


РИСУНОК 20.11. РАЗМЕР МНОЖЕСТВА СОСТАВНЫХ РЕБЕР В УСЛОВИЯХ РЕАЛИЗАЦИИ АЛГОРИТМА ПРИМА, ИСПОЛЬЗУЮЩЕГО ПОИСК ПО ПРИОРИТЕТАМ.

Чертеж в нижней части рисунка показывает размеры множества составных ребер по мере продвижения поиска по приоритетам для примера, приведенного на рис. 20.10. Для сравнения соответствующие чертежи для поиска в глубину, для рандомизированного поиска и для поиска в ширину из рис. 18.28, показаны вверху серым цветом.

Реализация дерева MTS для насыщенных графов, которая фактически эквивалентна программе 20.6, была впервые опубликована Примом (Prim) в 1961 г. и, независимо от него, Дейкстрой (Dijkstra) – несколько позже. Обычно она называется *алгоритмом Прима (Prim's algorithm)*, хотя формулировка Дейкстры носит несколько более обобщенный характер, что дает повод некоторым ученым рассматривать алгоритм вычисления дерева MTS как специальный случай алгоритма Дейкстры. Однако основная идея была высказана Ярником (Jarnik) в 1939 г., так что некоторые авторы называют этот метод *алгоритмом Ярника (Jarnik's algorithm)*, тем самым умаляя роль Прима (а также Дейкстры) до авторства разработки эффективной реализации рассматриваемого алгоритма для насыщенных графов. После того как АТД очереди с приоритетами в начале семидесятых годов получил широкое распространение, применение этого алгоритма для отыскания деревьев MTS на разреженных графах уже не представляло трудности; тот факт, что дерево MTS для разреженных графов стало возможным рассчитать за время, пропорциональное $E \lg V$, не связан с именем какого-либо исследователя. Как будет показано в разделе

20.6, с тех пор многие исследователи направили свои усилия на поиск эффективных реализаций как ключевого момента отыскания эффективных алгоритмов построения деревьев MTS для разреженных графов.

Упражнения

▷ **20.35**. Проведите анализ рабочих характеристик реализации алгоритма Прима "в лоб", о которой шла речь в начале данного раздела при обсуждении графа с V вершинами.

Указание: При решении этой задачи может пригодиться комбинаторная сумма: $\sum_{1 \le k < V} k(V-k) = (V+1)V(V-1)/6$.

○ **20.36**. Выполните упражнение 20.35 применительно к графам, у которых все вершины имеют одни и те же степени с фиксированным значением t.

○ **20.37**. Выполните упражнение 20.35 применительно к разреженным графам общего вида, содержащих V вершин и E ребер. Поскольку время выполнения зависит от весов ребер и степеней вершин, проведите анализ худшего случая. Приведите в качестве примера семейство графов, для которого ваши предположения относительно граничного значения для худшего случая подтверждаются.

20.38. Представьте в стиле рис. 20.8 результаты построения дерева MST с помощью алгоритма Прима для сети, определение которой дано в упражнении 20.26.

● **20.39**. Опишите семейство графов с V вершинами и E ребрами, для которого оценка времени выполнения алгоритма Прима, использующего поиск по приоритетам, подтверждается для худшего случая.

●● **20.40**. Разработайте походящий генератор случайных графов с V вершинами и E ребрами, такой, что время выполнения алгоритма Прима, использующего поиск по приоритетам (программа 20.7), будет нелинейным.

○ **20.41**. Внесите такие изменения в программу 20.7, чтобы она работала так же, как и программа 18.8, в части хранения в бахроме всех ребер, инцидентных вершинам дерева. Проведите эмпирические исследования с целью сравнить полученную вами реализацию с программой 20.7 на примере различных взвешенных графов (см. упражнения 20.9–20.14).

20.42. Постройте реализацию очереди с приоритетами, пользуясь интерфейсом, определенным в программе 9.12, в программе 20.7 (так что возможно использование любой реализации этого интерфейса).

20.43. Воспользуйтесь функцией **priority_queue** из библиотеки STL для реализации интерфейса очереди с приоритетами, который применяется в программе 20.7.

20.44. Предположим, что вы используете реализацию очереди с приоритетами, которая поддерживает сортированные списки. Каким будет время выполнения в худшем случае для графа с V вершинами и с E ребрами с точностью до постоянного множителя? В каких случаях этот метод будет полезен, если вообще будет когда-либо полезен? Обоснуйте ответ.

○ **20.45**. Ребро минимального остовного дерева, удаление которого из графа приводит к увеличению веса дерева MTS, называется *критическим ребром*. Покажите, как найти все критические ребра в графе за время, пропорциональное $E \lg V$.

20.46. Проведите эмпирические исследования с целью сравнить на различных взвешенных графах производительность программы 20.6 с производительностью программы 20.7, используя реализацию очереди с приоритетами в виде неупорядоченного массива (см. упражнения 20.10–20.14).

• **20.47**. Проведите эмпирические исследования на различных взвешенных графах с целью оценки эффекта от использования в программе 20.7 реализации очереди с приоритетами в виде турнира индексно-сортирующего дерева (см. упражнение 9.53) вместо программы 9.12 (см. упражнения 20.9–20.14).

20.48. Проведите эмпирические исследования на различных взвешенных графах с целью анализа весов деревьев (см. упражнение 20.23) как функции от V (см. упражнения 20.9–20.14).

20.49. Проведите эмпирические исследования на различных взвешенных графах с целью анализа максимального размера бахромы как функции от V (см. упражнения 20.9–20.14).

20.50. Проведите эмпирические исследования на различных взвешенных графах с целью анализа высоты дерева как функции от V (см. упражнения 20.9–20.14).

20.51. Проведите эмпирические исследования с целью анализа зависимости результатов выполнения упражнений 20.49 и 20.50 от выбора исходной вершины. Целесообразно ли использовать в качестве исходной точки случайную вершину?

20.52. Напишите клиентскую программу, которая выполняет графическую анимацию динамики алгоритма Прима. Ваша программа должна строить изображения, подобные представленным на рис. 20.10 (см. упражнения 17.56–17.60). Проверьте, как работает ваша программа, на случайных эвклидовых графах с близкими связями и на сетчатых графах (см. упражнения 20.17 и 20.19), используя столько точек, сколько можно обработать за приемлемое время.

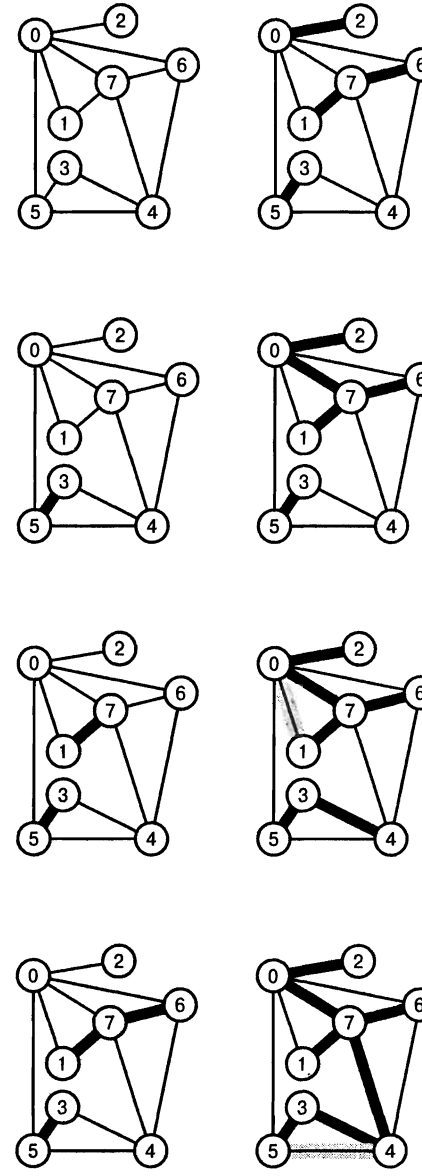
20.4. Алгоритм Крускала

Алгоритм Прима строит минимальное остовное дерево (MST) по одному ребру за раз, отыскивая на каждом шаге ребро, которое присоединяется к единственному дереву, растущему с каждым шагом. Алгоритм Крускала также строит дерево MST, увеличивая его на одно ребро на каждом шаге, однако в отличие от алгоритма Прима, он отыскивает ребро, которое соединяет два дерева в лесу, образованном растущими поддеревьями MST. Мы начнем построение с вырожденного леса, содержащего V деревьев, состоящих из одной вершины каждое, и выполняем операцию объединения двух деревьев (используя для этой цели по возможности наиболее короткие ребра), пока не останется единственное дерево, а именно, минимальное остовное дерево.

На рис. 20.12 представлен пример пошагового выполнения алгоритма Крускала; рис. 20.13 служит иллюстрацией динамических характеристик этого алгоритма на более крупном примере. Разобщенный лес поддеревьев MST постепенно превращается в единственное дерево. Ребра добавляются в дерево MST в порядке возрастания их длины, таким образом, лес содержит вершины, которые соединены друг с другом относительно короткими ребрами. В любой момент выполнения алгоритма каждая вершина расположена ближе к некоторой вершине своего поддерева, чем любая другая вершина, не входящая в это дерево.

РИСУНОК 20.12. АЛГОРИТМ КРУСКАЛА ПОСТРОЕНИЯ ДЕРЕВА MST

Пусть задан список ребер графа в произвольной форме (левый список ребер). Первый шаг алгоритма Крускала заключается в их сортировке по их весам (правый список ребер). Затем мы производим просмотр ребер этого списка в порядке возрастания их весов, добавляя в дерево MST ребра, которые не создают в нем циклов. Сначала мы добавляем ***5-3*** *(самое короткое ребро), далее ребро* ***7-6*** *(слева), затем* ***0-2*** *(справа вверху) и* ***0-7*** *(справа, вторая диаграмма сверху). Ребро* ***0-1*** *со следующим по величине весом создает цикл и не принимается во внимание. Ребра, которые мы не включаем в дерево MST, показаны в отсортированном списке серым цветом. Затем мы добавляем ребро* ***4-3*** *(справа, третья диаграмма сверху). Далее мы отвергаем ребро* ***5-4****, поскольку оно образует цикл, затем мы добавляем* ***7-4*** *(диаграмма справа внизу). Как только дерево MST будет готово, любое ребро с большим, чем у каждого ребра этого дерева весом, образует цикл и в силу этого обстоятельства игнорируется (алгоритм останавливается, когда в дерево MST будут включены $V - 1$ ребер). В отсортированном списке эти ребра помечены звездочками.*



0-6 .51	**5-3 .18**
0-1 .32	**7-1 .21**
0-2 .29	**7-6 .25**
4-3 .34	**0-2 .29**
5-3 .18	**7-0 .31**
7-4 .46	0-1 .32
5-4 .40	**4-3 .34**
0-5 .60	5-4 .40
6-4 .51	**7-4 .46**
7-0 .31	* 0-6 .51
7-6 .25	* 6-4 .51
7-1 .21	* 0-5 .60

Алгоритм Крускала прост в реализации, если для его построения используются базовые алгоритмические инструментальные средства, которые мы рассматривали в данной книге. В самом деле, мы можем использовать любую сортировку из описанных в части 3 для упорядочения ребер по весу и любой из алгоритмов решения задачи связности, рассмотренных в главе 1, для удаления тех из них, которые образуют циклы. Программа 20.8 представляет собой соответствующую реализацию функции построения дерева MST для АТД графа, которая в функциональном плане эквивалентна другим реализациям MST, рассмотренным в данной главе. Эта реализация не зависит от представления графа: она вызывает клиентскую программу **GRAPH** с целью получения вектора, в котором содержатся ребра графа, а затем на основе этого вектора строит дерево MST.

Обратите внимание на тот факт, что существуют два способа окончания работы алгоритма Крускала. Если мы найдем $V-1$ ребер, то мы уже построили остовное дерево и можем остановиться. Если мы проверим все вершины и не найдем при этом $V-1$ древесных ребер, это означает, что мы обнаружили, что граф не является связным, точно так же, как это делалось в главе 1.

Анализ времени выполнения алгоритма не представляет трудностей, ибо мы знаем время выполнения составляющих его операций АТД.

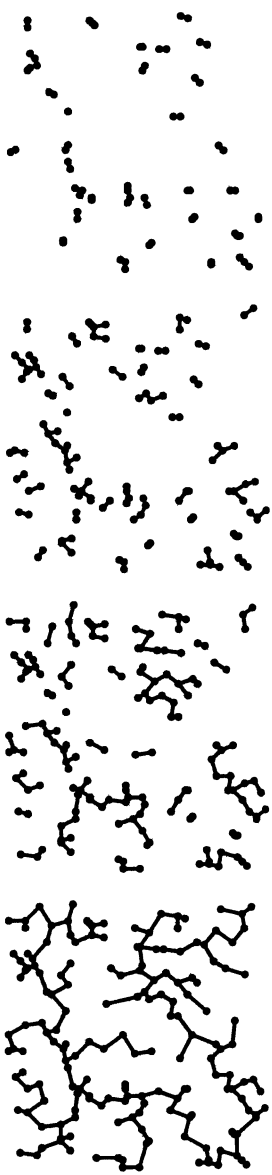


РИСУНОК 20.13. АЛГОРИТМ КРУСКАЛА, ОБЕСПЕЧИВАЮЩИЙ ПОСТРОЕНИЕ ДЕРЕВА MST

Эта последовательность показывает 1/4, 1/2, 3/4 всех ребер и все ребра дерева MST по мере его роста.

Программа 20.8. Алгоритм Крускала, обеспечивающий построение дерева MST

Эта реализация использует разработанные нами сортирующий АТД из главы 6 и АТД объединения-поиска из главы 4 для отыскания дерева MST, рассматривая ребра в порядке возрастания их весов и отбрасывая те ребра, которые образуют циклы, до тех пор, пока не будут обнаружены $V-1$ вершин, которые составляют остовное дерево.

Здесь не показан интерфейсный класс **EdgePtr**, который заключает в себе указатели на ребра, благодаря чему функция **sort** может сравнивать их, используя для этой цели перегрузку операции < в соответствии с изложенным в разделе 6.8, а также версия программы 20.2 с третьим аргументом спецификатора шаблона.

```
template <class Graph, class Edge, class EdgePtr>
class MST
{ const Graph &G;
  vector<EdgePtr> a, mst;
  UF uf;
public:
  MST(Graph &G) : G(G), uf(G.V()), mst(G.V())
  { int V = G.V(), E = G.E();
    a = edges<Graph, Edge, EdgePtr>(G);
    sort<EdgePtr>(a, 0, E-1);
```

```
    for (int i = 0, k = 1; i < E && k < V; i++)
      if (!uf.find(a[i]->v, a[i]->w))
        { uf.unite(a[i]->v, a[i]->w);
          mst[k++] = a[i]; }
      }
};
```

Свойство 20.9. *Алгоритм Крускала вычисляет дерево MST графа за время, пропорциональное* $E \lg E$.

Доказательство: Это свойство является следствием более общего факта, устанавливающего, что время выполнения программы 20.8 пропорционально затратам на сортировку E чисел плюс стоимость E операций *поиска (find)* и $V-1$ операций *объединения (union)*. Если мы используем стандартные реализации АТД, такие как сортировка слиянием и взвешенный алгоритм поиска-объединения с делением пополам, то основные затраты приходятся на сортировку. ■

Мы проведем сравнение производительности алгоритмов Крускала и Прима в разделе 20.6. А пока обратите внимание на то обстоятельство, что время выполнения, пропорциональное $E \lg E$,не обязательно хуже, чем $E \lg V$, поскольку E, самое большее, может достигать значения V^2, следовательно, $\lg E$ не превосходит $2 \lg V$. Различие в производительности на конкретных графах обусловливаются особенностями реализации и зависят от того, приближается ли фактическое время выполнения к этим граничным значениям, соответствующим худшему случаю.

На практике мы можем воспользоваться быстрой сортировкой или быстрой системной сортировкой (в основу которой, скорее всего, будет положена быстрая сортировка). И хотя следствием такого подхода может быть непривлекательная (по крайней мере, в теории) квадратичная зависимость времени выполнения сортировки в худшем случае, тем не менее, она, по-видимому, может дать максимально возможное быстродействие рассматриваемого алгоритма. В самом деле, с другой стороны, мы можем воспользоваться поразрядной сортировкой, чтобы выполнить упорядочение ребер за линейное время (при определенных ограничениях на веса ребер), благодаря чему затраты на выполнение E операций *find* будут составлять основную часть всех затрат, и внести изменения в формулировку свойства 20.9, в результате чего это свойство будет утверждать, что при сохранении прежних требований к весам ребер время выполнения алгоритма Крускала не выходит за пределы постоянного коэффициента в выражении $E \lg^* E$ (см. главу 2). Напомним, что функция $\lg^* E$ есть число итераций двоичной логарифмической функции, прежде чем результат примет значение меньше единицы; ее значение меньше 5, если E меньше 2^{65536}. Другими словами, такая корректировка делает алгоритм Крускала в действительности *линейным* в большинстве практических ситуаций.

Обычно стоимость построения дерева MST с помощью алгоритма Крускала даже меньше, чем стоимость обработки всех ребер, поскольку его построение завершается задолго до того, как будет рассмотрена существенная часть всех ребер (длинного) графа. Мы должны учитывать этот факт в своих попытках существенно уменьшить время в условиях многих практических ситуаций за счет того, что в процессе сортировки игнорируются все ребра, длина которых превышает длину самого длинного ребра MST. Один из самых простых способов достижения этой цели предусматривает использование очереди

с приоритетами, реализация которой применяет операцию *construct* (построить), выполняющуюся за линейное время, и операцию *remove the minimum* (удалить минимальное ребро) за логарифмическое время.

Например, мы можем достичь таких рабочих характеристик в рамках стандартной реализации полного бинарного дерева, используя для этой цели восходящую конструкцию (см. раздел 9.4). В частности, мы вносим в программу 20.8 следующие изменения: сначала мы вызываем процедуру **sort** с целью вызова функции **pq.construct()** с тем, чтобы построить полное бинарное дерево за время, пропорциональное E. Затем мы вносим изменения во внутренний цикл, суть которых состоит в отборе из очереди с приоритетами самых коротких ребер, т.е. **e = pq.delmin()**, и замене всех обращений к **a[i]** на обращения к **e**.

Свойство 20.10. *Версия алгоритма Крускала, в основу которой положена очередь с приоритетами, вычисляет дерево MST графа за время, пропорциональное* $E + X \lg V$, *где* X *есть число ребер графа, не превосходящих по длине самое длинное ребро в дереве MST.*

Доказательство: Обратимся к предыдущему обсуждению, которое показывает, что затраты на вычисление рассматриваемого алгоритма суть затраты на построение очереди размера E плюс стоимость выполнения X операций *удалить ребро минимальной длины* (delete the minimum), X операций *поиска (find)* и $V-1$ операций *объединения (union)*. Обратите внимание, что основная доля затрат приходится на построение очереди с приоритетами (и рассматриваемый алгоритм линеен во времени), пока X больше $E / \lg V$. ■

Мы можем воспользоваться той же идеей для получения аналогичных выгод от реализации рассматриваемого алгоритма на основе быстрой сортировки. Рассмотрим, что случится, если мы воспользуемся прямой рекурсивной быстрой сортировкой, по условиям которой мы проводим разбиение на элементе i, затем проводим рекурсивную сортировку подфайла, расположенного слева от i, и подфайла справа от i. Мы учитываем тот факт, что в силу особенностей построения алгоритма, первые i элементов расположены в порядке, установленном сортировкой после завершения первого рекурсивного вызова (см. программу 9.2). Этот очевидный факт позволяет немедленно получить быстродействующую реализацию алгоритма Крускала: если мы поместим проверку, порождает ли ребро **a[i]** цикл между рекурсивными вызовами, то получаем алгоритм, который, в соответствие с конструкцией, выполняет проверку первых i ребер в порядке сортировки по завершении первого рекурсивного вызова! Если мы включим проверку на повторное выполнение, после того как мы уже выявили $V-1$ ребер дерева MST, то мы имеем в своем распоряжении алгоритм, который сортирует столько ребер, сколько их необходимо для вычисления дерева MST, плюс несколько дополнительных стадий разбиения, использующих элементы с большими значениями (см. упражнение 20.57). Как и в случае простых реализаций сортировки, этот алгоритм может обеспечить квадратичную зависимость времени выполнения в худшем случае, однако мы можем рассчитывать на вероятностную гарантию того, что время его выполнения в худшем случае не подойдет близко к указанному пределу. Кроме того, подобно простым реализациям сортировки, эта программа, по всей видимости, обеспечивает большее быстродействие, чем реализация на базе полного бинарного дерева, что обусловливается ее более коротким внутренним циклом.

Если же граф не является связным, то версия алгоритма Крускала, основанная на частичной сортировке, не дает никаких преимуществ, поскольку в этом случае необходимо проводить анализ всех ребер графа. Даже в случае связного графа самое длинное ребро может оказаться в дереве MST, так что любая реализация метода Крускала должна выполнять анализ всех ребер. Например, граф может состоять из плотных кластеров вершин, при этом все они соединены между собой короткими ребрами и только одна не входящая в кластер вершина соединена с какой-либо вершиной длинным ребром. Несмотря на возможность таких нетипичных вариантов, подход с применением частичной сортировки, по-видимому, достоин нашего внимания, поскольку он обещает существенный выигрыш в производительности и не требует больших дополнительных затрат, а то и не требует их вовсе.

Здесь уместна краткая историческая справка, в некотором смысле она довольно поучительна. Крускал предложил свой алгоритм в 1956 г., однако, опять-таки, соответствующие реализации АТД не подвергались подробному изучению в течение многих лет. В силу этого обстоятельства, рабочие характеристики реализаций, такие как версия программы 20.8 с очередями с приоритетами, не получили надлежащей оценки вплоть до семидесятых годов прошлого столетия. Другим интересным историческим фактом является то, что в статье Крускала упоминалась версия алгоритма Прима (см. упражнение 20.59), а Борувка ссылался в своей статье на оба эти подхода. Эффективная реализация метода Крускала на разреженных графах предшествовала реализации метода Прима для разреженных графов в силу того, что абстрактными типами данных поиска-объединения (и сортировки) стали пользоваться раньше, чем абстрактными типами данных очереди с приоритетами. По существу, прогресс в современном состоянии алгоритма Крускала, равно как и в реализации алгоритма Прима, обусловлен главным образом улучшением характеристик производительности АТД. С другой стороны, применимость абстракции поиска-объединения к алгоритму Крускала и применимость абстракции очереди с приоритетами к алгоритму Прима стали для многих исследователей основной мотивацией для поиска более совершенных реализаций упомянутых выше АТД.

Упражнения

▷ **20.53**. Покажите в стиле рис. 20.12 результат вычисления дерева MST по алгоритму Крускала для сети, определение которой дано в упражнении 20.26.

○ **20.54**. Проведите эмпирические исследования на различных видах взвешенных графов с целью определения длины самого длинного ребра дерева MST и числа ребер графа, длина которых не превосходит длины этого ребра (см. упражнения 20.9–20.14).

● **20.55**. Разработайте реализацию АТД поиска-объединения, которое выполняет операцию *find* за постоянное время и операцию *union* за время, пропорциональное $\lg V$.

20.56. Проведите эмпирическое тестирование на различных видах взвешенных графов с целью сравнить полученную вами при решении упражнения 20.55 реализацию АТД с взвешенной операцией поиска-объединения с делением пополам (программа 1.4), когда алгоритм Крускала представляет собой клиентскую программу (см. упражнения 20.9–20.14). Выведите затраты на сортировку ребер отдельно с таким расчетом, чтобы можно было изучать их влияние на общие затраты и на часть расходов, связанных с АТД поиска-объединения.

20.57. Разработайте реализацию на основе идеи, описанной в тексте, в рамках которой мы интегрируем алгоритм Крускала с быстрой сортировкой с тем, чтобы проверить каждое ребро на принадлежность дереву MST, если нам известно, что все ребра с малым весом подверглись проверке.

○ **20.58**. Внесите в алгоритм Крускала такие изменения, чтобы сделать возможной реализацию двух функции АТД, которые заполняют вектор, индексированный именами вершин, поставляемый клиентской программой. Этот вектор осуществляет разбиение вершин на k кластеров, обладающих тем свойством, что ни одно ребро с длиной, большей d, не соединяет две вершины из различных кластеров. Первая функция принимает k в качестве аргумента и возвращает d; вторая функция принимает d как аргумент и возвращает k. Проверьте свою программу на случайных эвклидовых графах с близкими связями и на сетчатых графах (см. упражнения 20.17 и 20.19) различных размеров, варьируя значения k и d.

20.59. Разработайте реализацию алгоритма Прима, в основу которой положена предварительная сортировка ребер.

● **20.60**. Напишите клиентскую программу, которая выполняет графическую анимацию динамики алгоритма Крускала (см. упражнение 20.52). Проверьте полученную программу на случайных эвклидовых графах с близкими связями и на сетчатых графах (см. упражнения 20.17 и 20.19), используя при этом столько точек, сколько можно обработать за приемлемый промежуток времени.

20.5. Алгоритм Борувки

Следующий алгоритм вычисления дерева MTS, который мы будем рассматривать, также принадлежит к числу самых старых. Подобно алгоритму Крускала, мы строим дерево MTS, добавляя ребра в развернутый лес поддеревьев MTS, но делаем это поэтапно, добавляя на каждом этапе несколько ребер в дерево MTS. На каждом этапе мы отыскиваем наиболее короткое ребро, которое соединяет каждое поддерево MTS с некоторым другим поддеревом, затем включаем каждое такое ребро в дерево MTS.

И вновь разработанный нами в главе 1 АТД поиска-объединения позволяет получить эффективную реализацию. Для рассматриваемой задачи целесообразно расширить интерфейс этого АТД, обеспечив доступность операции *find* для клиентских программ. Мы будем пользоваться этой функцией для того, чтобы присвоить индекс каждому поддереву с таким расчетом, чтобы можно было быстро определить, к какому поддереву принадлежит заданная вершина. Обладая такой возможностью, мы способны эффективно реализовать каждую операцию, необходимую для выполнения алгоритма Борувки.

Прежде всего, мы построим вектор, индексированный именами вершин, который для каждого поддерева MTS определяет ближайшего соседа. Затем на каждом ребре графа мы выполняем следующие операции:

- Если соединяются две вершины одного и того же дерева, результат операции отбрасывается.
- В противном случае выполните проверку расстояний между вершинами двух ближайших соседних деревьев, которые соединяет это ребро, и обновите их, если в этом есть необходимость.

После такого просмотра всех ребер графа, вектор ближайших соседних вершин содержит информацию, которая нам нужна для соединения поддеревьев. Для каждого индекса вершины мы выполняем операцию *union* с целью ее соединения с ближайшей соседней вершиной. На следующей стадии мы отбрасываем все более длинные ребра, которые соединяют другие пары вершин в уже соединенных поддеревьях MTS. Рисунки 20.14 и 20.15 служат иллюстрацией работы выбранного нами алгоритма.

Программа 20.9 представляет собой прямую реализацию алгоритма Борувки. Следующие три главных фактора делают эту реализацию эффективной:

- Стоимость каждой операции *find* фактически остается постоянной.
- Каждый этап уменьшает число поддеревьев MTS в лесе, по меньшей мере, в два раза.
- На каждом этапе отбрасывается значительное количество ребер.

Трудно дать точную количественную оценку всех этих факторов, в то же время нетрудно установить следующие границы.

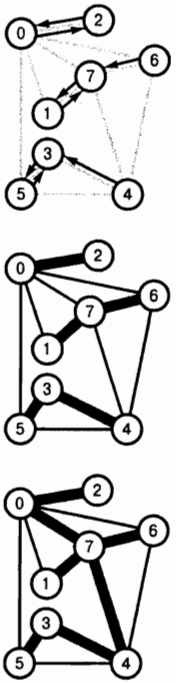


РИСУНОК 20.14. АЛГОРИТМ БОРУВКИ ПОСТРОЕНИЯ ДЕРЕВА MTS

*На диаграмме вверху показаны ориентированные ребра, проведенные из каждой вершины в ближайшие к ним соседние вершины. Из этой диаграммы следует, что ребра **0-2**, **1-7** и **3-5** являются самыми короткими ребрами, инцидентные обоим их вершинам, **6-7** есть кратчайшее ребро для вершины **6**, а **4-3** – кратчайшее ребро для вершины **4**. Все эти ребра содержатся в дереве MTS и образуют лес поддеревьев MTS (диаграмма в центре), который вычисляется на первом этапе выполнения алгоритма Борувки. На втором этапе этот алгоритм завершает вычисление поддеревьев MTS (диаграмма внизу) добавлением ребра **0-7**, которое является кратчайшим ребром, инцидентным каждой вершине тех поддеревьев, которые он соединяет между собой, и ребра **4-7**, которое является кратчайшим ребром, инцидентным каждой вершине нижнего поддерева.*

Программа 20.9. Алгоритм Борувки построения дерева MTS

Эта реализация алгоритма Борувки построения дерева MTS использует версию АТД объединения-поиска из главы 4 (в интерфейс добавляется функция **find** с одним аргументом) с целью привязки индексов к поддеревьям MTS по мере их построения. На каждом этапе проверяются все оставшиеся ребра; те из них, которые соединяют отдельные поддеревья, сохраняются до следующего этапа. Массив **a** содержит ребра, которые не были отвергнуты и которые еще не включены в поддеревья MTS. Индекс **N** используется для хранения ребер, отложенных до следующего этапа (приводимый ниже программный код переустанавливает значение **E** вместо **N** в конце каждого этапа), а индекс **h** используется для доступа к следующему проверяемому ребру. Ближайший сосед каждой компоненты хранится в массиве **b** с номерами компоненты **find** в качестве индексов. В конце каждого этапа каждая компонента соединяется с ближайшей соседней, а ребра ближайшей соседней вершины добавляются в дерево MTS.

```
template <class Graph, class Edge> class MST
{ const Graph &G;
   vector<Edge *> a, b, mst;
   UF uf;
public:
   MST(const Graph &G): G(G), uf(G.V()), mst(G.V()+1)
   { a = edges<Graph, Edge>(G);
      int N, k = 1;
      for (int E = a.size(); E != 0; E = N)
      { int h, i, j;
         b.assign(G.V(), 0);
         for (h = 0, N = 0; h < E; h++)
         { Edge *e = a[h];
            i = uf.find(e->v()), j = uf.find(e->w());
            if (i == j) continue;
            if (!b[i] || e->wt() < b[i]->wt()) b[i] = e;
            if (!b[j] || e->wt() < b[j]->wt()) b[j] = e;
            a[N++] = e;
         }
         for (h = 0; h < G.V(); h++)
          if (b[h])
            if (!uf.find(i = b[h]->v(), j = b[h]->w()))
               { uf.unite(i, j); mst[k++] = b[h]; }
      }
   }
};
```

Свойство 20.11. *Время прогона алгоритма Борувки с целью вычисления дерева MTS заданного графа есть $O(E \lg V \lg^* E)$.*

Доказательство: Поскольку число деревьев в лесе уменьшается наполовину на каждом этапе, число этапов не превышает значения $\lg V$. Время выполнения каждого этапа, самое большее, пропорционально затратам на выполнение E операций *find*, что меньше $E \lg^* E$, или линейно с точки зрения практических приложений. ■

Время прогона, оценка которого дается свойством 20.11, представляет собой консервативную верхнюю границу, поскольку оно не учитывает существенное уменьшение числа ребер на каждом этапе. На выполнение операций *find* затрачивается постоянное время на ранних проходах, а на последующих проходах число рассматриваемых ребер существенно уменьшается. В самом деле, для многих графов число ребер уменьшается по экспоненте от числа вершин, а общее время прогона оказывается пропорциональным E. Например, как показано на рис. 20.16, рассматриваемый алго-

	0	1	2	3	4	5	6	7
Начальный этап	0	1	2	3	4	5	6	7
Этап 1	0	1	0	3	3	3	1	1
Этап 2	1	1	1	1	1	1	1	1

РИСУНОК 20.15. ПРИМЕНЕНИЕ МАССИВА ПОИСКА-ОБЪЕДИНЕНИЯ В АЛГОРИТМЕ БОРУВКИ

На данной диаграмме показано содержимое массива поиска-объединения, соответствующего примеру, представленному на рис. 20.14. Первоначально каждый элемент содержит свой собственный индекс, указывающий вершину в лесе изолированных вершин. В результате выполнения первого этапа мы получаем три компоненты, представленные вершинами 0, 1 и 3 (в условиях этого крохотного примера деревья, построенные при помощи операций поиска и объединения, лежат в одной плоскости). По окончании второго этапа мы получаем единственную компоненту, которую представляет вершина 1.

ритм отыскивает дерево MTS приводимого нами в качестве примера более крупного графа всего лишь за четыре этапа.

Можно сдвинуть коэффициент $\lg^* E$ в область более низких теоретических границ времени прогона алгоритма Борувки и сделать его пропорциональным $E \lg V$ за счет представления поддеревьев MTS в виде двухсвязных списков вместо использования операций *union* и *find*. Однако это усовершенствование достаточно трудно для реализации и возможное повышение производительности не настолько заметно, чтобы можно было рекомендовать его для применения в практических приложениях (см. упражнения 20.66 и 20.67).

Как было отмечено выше, алгоритм Борувки – это самый старый алгоритм из числа тех, которые мы рассматриваем: его идея впервые была выдвинута в 1926 г. применительно к приложениям по распределению энергии. Этот метод был открыт вторично Соллиным (Sollin) в 1961 г.; несколько позже он привлек к себе внимание как основа для алгоритмов вычисления деревьев MTS с эффективной асимптотической производительностью и как основа для параллельного построения деревьев MTS.

Упражнения

▷ **20.61**. Покажите в стиле рис. 20.14 результат вычисления дерева MTS сети, определение которой дано в упражнении 20.26, с помощью алгоритма Борувки.

○ **20.62**. Почему программа 20.9 выполняет проверку *find* перед тем, как выполнять операцию *union*? *Указание*: Рассмотрите ребра одинаковой длины.

○ **20.63**. Объясните, почему значение **b(h)** может быть равным нулю (программа 20.9) в проверке, которая защищает операцию *union*.

● **20.64**. Дайте описание семейства графов с V вершинами E ребрами, для которых число ребер, не отвергнутых в процессе выполнения всех этапов алгоритма Борувки, настолько велико, что возникают условия, характерные для худшего случая.

20.65. Разработайте реализацию алгоритма Борувки, основанную на предварительной сортировке ребер.

○ **20.66**. Разработайте реализацию алгоритма Борувки, который использует для представления поддеревьев MST двухсвязные кольцевые списки, благодаря чему можно выполнять слияние и переименование поддеревьев за время, пропорциональное E, на каждом этапе (благодаря чему отношения эквивалентности в АТД не нужны).

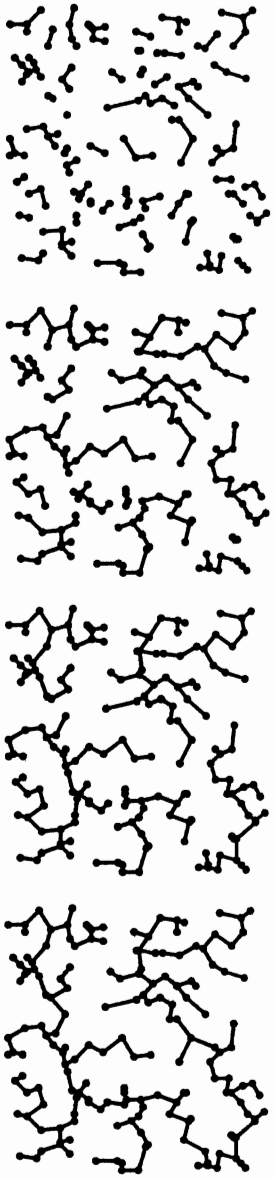


РИСУНОК 20.16. АЛГОРИТМ БОРУВКИ ПОСТРОЕНИЯ ДЕРЕВА MST

Для построения дерева MST в данном примере достаточно всего лишь четырех этапов (снизу вверх).

○ **20.67**. Проведите эмпирические исследования для сравнения реализации алгоритма Борувки из упражнения 20.66 с реализацией, приведенной в тексте (программа 20.9), на различных типах взвешенных графов (см. упражнения 20.9–20.14).

● **20.68**. Проведите эмпирические исследования для составления таблицы количества этапов и числа ребер, обрабатываемых на каждом этапе в аглоритме Борувки для различных типов взвешенных графов (см. упражнения 20.9–20.14).

20.69. Разработайте реализацию алгоритма Борувки, обеспечивающую построение нового графа (образующую лес, в котором каждая вершина представляет одно дерево) на каждом этапе.

● **20.70**. Напишите клиентскую программу, которая выполняет графическую анимацию динамики алгоритма Борувки (см. упражнения 20.52 и 20.60). Проверьте полученную программу на случайных эвклидовых графах с близкими связями и на сетчатых графах (см. упражнения 20.17 и 20.19), используя при этом столько точек, сколько можно обработать за приемлемый промежуток времени.

20.6. Сравнения и усовершенствования

Значения времени прогона рассмотренных выше базовых алгоритмов построения дерева MST сведены в таблицу 20.1, а в таблице 20.2 можно найти результаты эмпирических исследований по сравнению этих алгоритмов. На основании данных этих таблиц можно заключить, что реализация алгоритма Прима на графах, представленных в виде матрицы смежности лучше всего подходит для разреженных графов, что все другие методы отличаются по производительность от производительности наилучшего возможного случая лишь с точностью до небольшого постоянного множителя (затраты времени на извлечение ребер) на графах средней насыщенности, и что метод Крускала сводит решение этой задачи к сортировке на разреженных графах.

Короче говоря, мы можем считать проблему "решенной" для практических целей. Для большинства графов затраты на поиск дерева MST лишь немного выше, чем затраты на извлечение ребер графа. Это правило не распространяется на крупные, исключительно разреженные графы, тем не менее, достигнутое увеличение производительности даже на таких графах может при благоприятном стечении обстоятельств превзойти примерно в 10 раз лучшие из других известных алгоритмов. Результаты, приведенные в таблице 20.2, зависят от модели, использованной для генерирования графов, однако они в определенной степени характерны и для многих других моделей графов (см., например, упражнение 20.80). Тем не менее, эти теоретические результаты не отрицают существования алгоритмов с гарантированным линейным временем выполнения на всех графах; здесь же мы проведем обзор широкомасштабных исследований, проводимых с целью совершенствования реализации этих методов.

Прежде всего, проведенные широкомасштабные исследования позволили разработать более совершенные реализации очереди с приоритетами. Структура данных *полного бинарного дерева Фибоначчи (Fibonacci heap)*, представляющего собой расширение биномиальной очереди, достигает теоретически оптимальной производительности за счет постоянного времени выполнения операции *decrease key* (уменьшить ключ) и логарифмического времени выполнения операции *remove the minimum* (удалить минимальное ребро), что, в соответствие со свойством 20.8, приводит к тому, что время выполнения алгоритма Прима

становится пропорциональным сумме $E + V \lg V$. Полные бинарные деревья Фибоначчи являются более сложными структурами, нежели биномиальные очереди; они несколько неудобны при использовании на практике, но в то же время некоторые из простых реализации очередей с приоритетами обладают сходными характеристиками производительности (*см. раздел ссылок*).

Таблица 20.1. Затраты на алгоритмы построения дерева MST

Данная таблица подводит итоги затрат (худший случай времени прогона) на выполнение алгоритмов построения дерева MST для алгоритмов, изучение которых проводилось в данной главе. Приводимые здесь формулы основаны на предположении, что такое дерево MST существует и существует X ребер, длина которых не превосходит размеров самого длинного ребра в дереве MST (см. свойство 20.10). Границы худшего случая могут оказаться чрезмерно консервативными, чтобы оказаться полезными для прогнозирования производительности графов на практике. Рассматриваемые алгоритмы выполняются за время, близкое к линейному, причем в широком диапазоне реальных ситуаций.

Алгоритм	Затраты в худшем случае	Комментарии
Прима (стандартный)	V^2	Оптимальный для насыщенных графов.
Прима (поиск по приоритету, частично упорядоченное полное бинарное дерево)	$E \lg V$	Консервативная верхняя граница.
Прима (поиск по приоритету, d-арное частично упорядоченное полное дерево)	$E \log_d V$	Линейное время выполнения на всех графах, кроме чрезмерно разреженных.
Крускала	$E \lg E$	Основная часть затрат уходит на сортировку.
Крускала (частичная сортировка)	$E + X \lg V$	Затраты зависят от размера самого длинного ребра.
Борувки	$E \lg E$	Консервативная верхняя граница.

Один из эффективных подходов связан с применением сортировки с поразрядным обменом в реализации очереди с приоритетами. По производительности такие методы обычно эквивалентны методу Крускала или даже методам, предполагающим применение поразрядной сортировки в качестве метода, использующего частичную сортировку, который мы обсуждали в разделе 20.4.

Другой давно известный простой подход, предложенный Д. Джонсоном (D. Jonhson) в 1977 г., является наиболее эффективным методом: очередь с приоритетами для алгоритма Прима реализуется с помощью d-арных частично упорядоченных полных деревьев, а не с помощью стандартных полных бинарных деревьев (см. рис. 20.17). Программа 20.10 представляет собой полную реализацию интерфейса очереди по приоритету, который мы использовали ранее и в основу которого положен этот метод. Для такой реализации очереди с приоритетами на выполнение операции *decrease key* требуется менее $\log_d V$ шагов, а для выполнения операции *remove the minimum* требуется время, пропорциональное $d \log_d V$. По свойству 20.8 подобное поведение приводит к тому, что время выполнения алгоритма Прима будет пропорционально $Vd \log_d V + E \log_d V$, которое линейно для графов, не относящихся к категории разреженных.

Свойство 20.12. *Пусть задан граф с V вершинами и E ребрами и пусть d обозначает насыщенность E / V. Если $d < 2$, то время выполнения алгоритма Прима пропорционально $V \lg V$. В противном случае мы можем уменьшить время выполнения в худшем случае в $\lg(E/V)$ раз, воспользовавшись в качестве очереди с приоритетами $\lceil E/V \rceil$-арным частично упорядоченным полным деревом.*

Доказательство: В продолжение рассуждений из предыдущего параграфа отметим, что число шагов есть $Vd \log_d V + E \log_d V$, таким образом, время выполнения в лучшем случае пропорционально $E \log_d V = (E \lg V)/\lg d$. ■

Если E пропорционально $V^{1+\varepsilon}$, по свойству 20.12 время выполнения в худшем случае пропорционально E/ε, и значение линейно при любом значении константы ε. Например, если число ребер пропорционально $V^{3/2}$, то затраты не превышают $2E$; если число ребер пропорционально $V^{4/3}$, то затраты не превышают $3E$, и если число ребер пропорционально $V^{5/4}$, то затраты не превышают $4E$. Для графа с одним миллионом вершин затраты меньше $6E$, если насыщенность не превышает 10.

Таблица 20.2. Эмпирические исследования алгоритмов, вычисляющих дерево MST

Эта таблица показывает относительное время выполнения различных алгоритмов вычисления MST на случайных взвешенных графах разной насыщенности. При малых значениях насыщенности лучшие результаты дает алгоритм Крускала, поскольку он позволяет применять быструю сортировку. При больших значениях насыщенности лучшей остается классическая реализация алгоритма Прима, поскольку в этом случае отпадает необходимость в непроизводительных затратах на предварительную обработку списков. Для графов с промежуточной насыщенностью реализация алгоритма Прима с поиском по приоритету выполняется за время, не превышающее времени, необходимого для проверки каждого ребра графа, умноженного на небольшое постоянное значение.

E	*V*	C	H	J	P	K	K*	*ε/E*	B	*ε/E*
Насыщенность 2										
20000	10000	2	22	27		9	11	1.00	14	3.3
50000	25000	8	69	84		24	31	1.00	38	3.3
100000	50000	15	169	203		49	66	1.00	89	3.8
200000	100000	30	389	478		108	142	1.00	189	3.6
Насыщенность 20										
20000	1000	2	5	4	20	6	5	.20	9	4.2
50000	2500	12	12	13	130	16	15	.28	25	4.6
100000	5000	14	27	28		34	31	.30	55	4.6
200000	10000	29	61	61		73	68	.35	123	5.0
Насыщенность 100										
100000	1000	14	17	17	24	30	19	.06	51	4.6
250000	2500	36	44	44	130	81	53	.05	143	5.2
500000	5000	73	93	93		181	113	.06	312	5.5
1000000	10000	151	204	198		377	218	.06	658	5.6
Насыщенность V/2.5										
400000	1000	61	60	59	20	137	78	.02	188	4.5
2500000	2500	597	409	400	128	1056	687	.01	1472	5.5

Обозначения:

C	Извлекается всего ребер.
H	Алгоритм Прима (списки смежных вершин/индексированное частично упорядоченное полное бинарное дерево).
J	Версия Джонсона алгоритма Прима (очередь с приоритетами в виде частично упорядоченного полного d-арного дерева.
P	Алгоритм Прима (представление графа в виде матрицы смежности).
K	Алгоритм Крускала.
K*	Версия алгоритма Крускала с частичной сортировкой.
B	Алгоритм Борувки.
ε	Исследованные ребра (операции union).

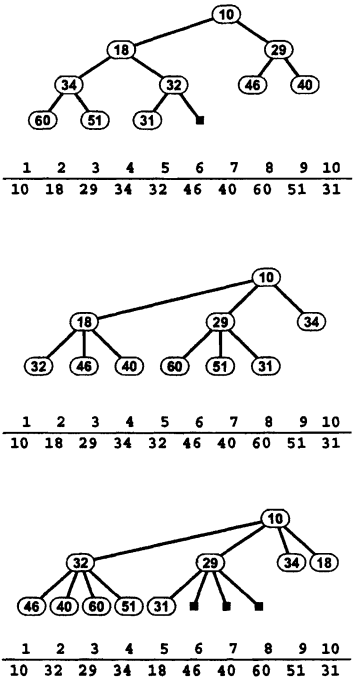


РИС. 20.17. 2-, 3- И 4-АРНЫЕ ЧАСТИЧНО УПОРЯДОЧЕННЫЕ ПОЛНЫЕ ДЕРЕВЬЯ

Когда мы сохраняем стандартное бинарное полное частично упорядоченное дерево в виде массива (диаграмма вверху), мы используем косвенные связи для перехода из некоторого узла i *вниз по дереву в его дочерние* $2i$ *и* $2i+1$ *узлы и вверх по дереву в его предок* $i/2$. *В 3-нарном полном частично упорядоченном дереве (диаграмма в центре) соответствующими связями узла* i *являются связи с дочерними узлами* $3i-1$, $3i$ *и* $3i+1$ *и с их родителем* $\lfloor (i+1)/3 \rfloor$. *Наконец, в 4-нарном полном частично упорядоченном дереве (диаграмма внизу) косвенными связями узла* i *являются связи с дочерними узлами* $4i-2$, $4i-1$, $4i$ *и* $4i+1$ *и с их родителем* $\lfloor (i+2)/4 \rfloor$. *Увеличение коэффициента ветвления в реализации полного частично упорядоченного дерева может оказаться полезным в приложениях, подобных алгоритму Прима, в условиях которых требуется выполнение большого числа операций* ***decrease key***.

Соблазн таким способом минимизировать границы времени выполнения алгоритмов для худшего случая облегчается пониманием того факта, что части $Vd \log_d V$ затрат избежать не удается (для операции *remove the minimum* мы должны исследовать d потомков в частично упорядоченном полном d-арном дереве по мере того, как мы продолжаем проверку ребер, перемещаясь вниз по дереву), однако части $E \lg V$, по-видимому, можно избежать (ибо большая часть ребер не требует обновления очереди с приоритетами, как было показано при обсуждении свойства 20.8).

Для типовых графов, подобных тем, результаты исследования которых отражены в таблице 20.2, уменьшение параметра d не оказывает влияния на время выполнения, а использование больших значений d может слегка замедлить реализацию. Тем не менее, небольшая защита, предусмотренная для производительности в худшем случае, делает целесообразной реализацию этого метода в силу ее простоты. В принципе, мы можем настроить реализацию таким образом, чтобы она выбирала оптимальное значение d для некоторых типов графов (выбирайте наибольшее значение, которое не замедляет выполнение алгоритма), но небольшое фиксированное значение (например, 3, 4 или 5) вполне подойдет, исключение могут представлять конкретные крупные классы графов с нестандартными характеристиками.

Использование частично упорядоченных полных d-арных деревьев нецелесообразно для разреженных графов, поскольку d должно быть больше или равным 2, а из этого условия следует, что мы не можем получить асимптотическое время выполнения, меньше чем $V \lg V$. Если насыщенность принимает небольшое постоянное значение, то линейный по времени выполнения алгоритм построения дерева MST будет выполняться за время, пропорциональное V.

Цель разработки практических алгоритмов для вычисления дерева MST для разреженных графов, обеспечивающих построение дерева MST за линейное время, труднодостижима. Интенсивным исследованиям подверглись различные варианты алгоритма Борувки как базиса алгоритмов вычисления дерева MST на сильно разреженных графах за почти линейное время (*см. раздел ссылок*). Такие исследования позволяют надеяться на то, что нам, наконец, удастся получить линейный во времени алгоритм, пригодный для практических целей, и они даже показали существование рандомизированного линейного по времени алгоритма. И хотя в общем случае эти алгоритмы достаточно сложны, упрощенные версии некоторых из них могут зарекомендовать себя на практике как весьма полезные. В то же время, мы можем использовать рассмотренные здесь базовые алгоритмы для вычисления дерева MST за линейное время в большинстве практических ситуаций, хотя, возможно, потребуется заплатить дополнительную цену в виде множителя $\lg V$ для разреженных графов.

Программа 20.10. Реализация очереди с приоритетами в виде многопозиционного частично упорядоченного полного дерева

Этот класс использует частично упорядоченное полное дерево с многократным ветвлением в узлах для реализации непрямого интерфейса для очереди с приоритетами, которая используется в данной книге. В его основе лежат изменения, внесенные в программу 9.12, в рамках которых конструктор принимает ссылки на вектор приоритетов, чтобы стала возможной реализация функций **getmin** и **lower** вместо **delmax** и **change** и обобщение функций **fixUp** и **fixDown**, благодаря которым они могут поддерживать d-арное сортирующее дерево (так, что операция *remove the minimum* выполняется за время, пропорциональное $d \log_d V$, в то же время операция *decrease the key* требует для своего выполнения $\log_d V$ шагов).

```
template <class keyType> class PQi
{ int d, N;
  vector<int> pq, qp;
  const vector<keyType> &a;
  void exch(int i, int j)
   { int t = pq[i]; pq[i] = pq[j]; pq[j] = t;
     qp[pq[i]] = i; qp[pq[j]] = j; }
     void fixUp(int k)
      { while (k > 1 && a[pq[(k+d-2)/d]] > a[pq[k]])
        { exch(k, (k+d-2)/d); k = (k+d-2)/d; } }
     void fixDown(int k, int N)
      { int j;
        while ((j = d*(k-1)+2) <= N)
         {
          for (int i = j+1; i < j+d && i <= N; i++)
            if (a[pq[j]] > a[pq[i]]) j = i;
          if (!(a[pq[k]] > a[pq[j]])) break;
          exch(k, j); k = j;
         }
      }
```

```
public:
   PQi(int N, const vector<keyType> &a, int d = 3) :
    a(a), pq(N+1, 0), qp(N+1, 0), N(0), d(d) { }
   int empty() const { return N == 0; }
   void insert(int v)
    { pq[++N] = v; qp[v] = N; fixUp(N); }
   int getmin()
    { exch(1, N); fixDown(1, N-1); return pq[N--]; }
   void lower(int k)
    { fixUp(qp[k]); }
};
```

Упражнения

○ **20.71**. [В. Высоцкий] Разработайте реализацию алгоритма, обсуждавшегося в разделе 20.2, который строит дерево MST путем добавления ребер по одному за раз и удаления самых длинных ребер из получающихся при этом циклов (см. упражнение 20.33). Воспользуйтесь представлением леса поддеревьев MST в виде родительских связей. *Указание*: При обходе путей в деревьях поменяйте направления указателей на обратные.

20.72. Проведите эмпирические тестирования с целью сравнения времени выполнения полученной вами реализации в упражнении 20.71 и времени выполнения алгоритма Крускала на взвешенных графах различного вида (см. упражнения 20.9–20.14). Проверьте, оказывает ли влияние на получаемые результаты рандомизация порядка, в котором проводится анализ ребер.

● **20.73**. Опишите, как вы будете искать дерево MST графа настолько большого, что в основной памяти одновременно могут находиться всего лишь V ребер.

○ **20.74**. Разработайте реализацию очереди с приоритетами, в которой операции *remove the minimum* и *find the minimum* выполняются за постоянное время и в которой время выполнения операции *decrease key* пропорционально логарифму размера очереди с приоритетами. Сравните полученную реализацию с 4-арными сортирующими деревьями, когда вы используете алгоритм Прима для отыскания дерева MST на разреженных графах, на различных видах взвешенных графов (см. упражнения 20.9–20.14).

20.75. Проведите эмпирические исследования с целью сравнения производительности различных реализаций при использовании алгоритма Прима на различных видах взвешенных графов (см. упражнения 20.9–20.14). Рассмотрите результаты применения d-арных сортирующих деревьев для различных значений d, биномиальных очередей, функции **priority_queue** из библиотеки STL, сбалансированных деревьев и любых других структур данных, которые, по вашему мнению, могут оказаться эффективными.

20.75. Разработайте реализацию, которая расширяет алгоритм Борувки за счет построения обобщенной очереди, содержащей лес поддеревьев MST. (Применение программы 20.9 соответствует использованию очереди FIFO.) Проведите эксперименты с другими реализациями алгоритма с обобщенными очередями на различных видах взвешенных графах (см. упражнения 20.9–20.14).

● **20.77**. Разработайте генератор случайных связных кубических графов (каждая вершина которого имеет степень 3), ребра которых снабжены случайными весами. Выполните для этого случая тонкую настройку рассмотренных нами алгоритмов вычисления деревьев MST, а затем определите, какой из них обладает наибольшим быстродействием.

○ **20.78.** Для $V = 106$ начертите кривую отношения верхней границы стоимости алгоритма Прима с d-арным сортирующим деревом к E как функцию от насыщенности d, для d из диапазона от 1 до 100.

○ **20.79.** Из таблицы 20.2 следует, что стандартная реализация алгоритма Крускала обладает значительно большим быстродействием, чем реализация с частичным упорядочением на графах с малой насыщенностью. Дайте объяснение этому явлению.

● **20.80.** Проведите эмпирические исследования в стиле таблицы 20.2 для случайных полных графов, наделенных весами, подпадающими под распределение Гаусса (см. упражнение 20.18).

20.7. Эвклидово дерево MST

Предположим, что даны N точек на плоскости, и мы хотим найти кратчайший набор линий, соединяющих все эти точки. Эта геометрическая задача называется задачей *поиска эвклидова дерева MST (Euclidian MST)* (см. рис. 20.18). Один из подходов к ее решению предусматривает построение полного графа с N вершинами и $N(N-1)/2$ ребрами – одно ребро соединяет каждую пару вершин, а вес этого ребра равен расстоянию между соответствующими точками. Затем мы можем при помощи алгоритма Прима найти дерево MST за время, пропорциональное N^2.

В общем случае для этого решения характерно низкое быстродействие. Эвклидова задача несколько отличается от задач на графах, которые мы рассматривали выше, поскольку все ребра определены неявно. Объем входных данных пропорционален N, так что решение, которое мы наметили для этой задачи в первом приближении, есть *квадратичный* алгоритм. Исследования показывают, что можно несколько улучшить этот показатель. Из рассматриваемой геометрической структуры следует, что большая часть ребер в полном графе не будет использоваться при решении задачи, и нет необходимости включать в граф большую часть этих ребер, прежде чем мы построим минимальное остовное дерево (дерево MST).

Свойство 20.13. *Мы можем отыскать эвклидово дерево MST для N точек за время, пропорциональное $N \log N$.*

Этот факт представляет собой прямое следствие двух важных фактов, касающихся точек на плоскости, которые мы будем рассматривать в части 7. Во-первых, граф, известный как *триангуляция Делони (Delauney triangulation)*, содержит дерево MST по определению. Во-вторых, триангуляция Делони есть планарный граф, число ребер которого пропорционально N. ■

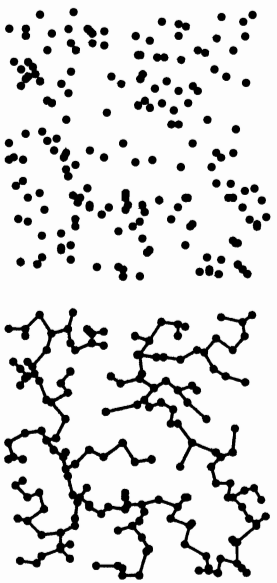


РИСУНОК 20.18. ЭВКЛИДОВО ДЕРЕВО MST

Пусть задано множество N точек на плоскости (вверху), тогда эвклидово дерево MST есть кратчайшее множество линий, которые их соединяют (внизу). Эта задача выходит далеко за рамки задач обработки графов, поскольку нам придется использовать глобальную информацию геометрического характера о точках на плоскости, дабы избежать необходимости обработки всех N^2 неявных ребер, соединяющих эти точки.

В принципе, мы можем вычислить триангуляцию Делони за время, пропорциональное $N \log N$, затем либо прогнать алгоритм Крускала, лиюо метод поиска по приоритету с целью вычисления эвклидового дерева MST за время, пропорциональное $N \log N$. Однако написание программы, вычисляющей триангуляцию Делони – задача не из легких даже для опытного программиста, так что в силу сказанного выше подобный подход может оказаться слишком убийственным для решения таких задач на практике.

Другие подходы вытекают из геометрических алгоритмов, которые будут рассматриваться в части 7. Для случайно распределенных точек мы можем поделить плоскость на квадраты таким образом, чтобы каждый квадрат содержал примерно $\lg N/2$ точек, что мы и делали в программе 3.20, осуществляющей поиск ближайшей точки. Далее, даже если мы включаем в граф только ребра, соединяющие каждую точку с точками из соседних квадратов, то, по-видимому (но отнюдь не обязательно), все эти ребра окажутся в минимальном остовном дереве; в этом случае мы можем воспользоваться алгоритмом Крускала или реализацией алгоритма Прима с поиском по приоритету, чтобы не снижать производительность на завершающей стадии вычислений. Примеры, который мы использовали на рис. 20.10, рис. 20.13 и рис. 20.16 и других подобных рисунках, были построены с использованием именно этого способа (рис. 20.19). Мы можем разработать новую версию алгоритма Прима на базе алгоритмов поиска ближайшего соседа, чтобы не обрабатывать отдаленные вершины.

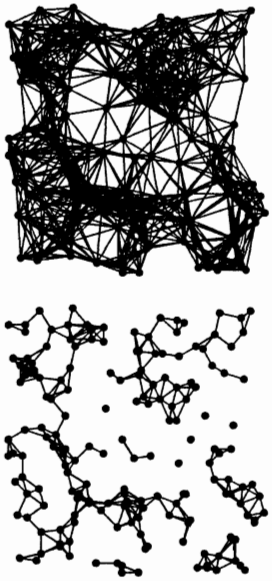


РИСУНОК 20.19. ЭВКЛИДОВЫ ГРАФЫ БЛИЖАЙШИХ СОСЕДЕЙ

Один из способов вычисления эвклидового дерева MST состоит в построении графа, в котором каждое ребро соединяет каждую пару точек, расположенных друг от друга на расстоянии, не превышающем d, подобно графу, изображенному на рис. 20.8 и на ряде других. Тем не менее, этот метод дает слишком много ребер, если расстояние d достаточно велико (диаграмма вверху), при этом нет гарантии того, что существуют все ребра, соединяющие все точки, если d меньше длины самого длинного ребра в дереве MST (диаграмма внизу).

Имея в своем распоряжении богатый выбор подходов к решению этой задачи и возможности линейных алгоритмов для решения в общем виде задачи отыскания дерева MST, важно отметить, что в рассматриваемом случае существует простая нижняя граница наших возможностей.

Свойство 20.14. *Вычислить эвклидово дерево MST для N точек не проще, чем сортировка N чисел.*

Доказательство: Пусть задан список чисел, подлежащих сортировке, преобразуем этот список в список точек, в котором в качестве координаты x берется соответствующее число из списка чисел, а координата y принимается равной 0. Требуется найти дерево MST для построенного списка точек. Далее (как и в случае алгоритма Крускала) поместите точки в АТД графа и выполните поиск в глубину с целью построения остовного дерева, начиная с точки с минимальной координатой x. Это основное дерево представлено в виде связного списка упорядоченных чисел; таким образом, мы решили задачу сортировки чисел. ■

Точные интерпретации этой нижней границы достаточно сложны, поскольку базовые операции, используемые для решения этих двух задач (сравнение координат в задаче сортировки и сравнение расстояний в задаче построения дерева MST), различны и ввиду того, что существует возможность использования методов, таких как поразрядная сортировка и метод перспективных сеток. Тем не менее, мы можем интерпретировать эту границу следующим образом: по мере выполнения сортировки мы должны следить за тем, чтобы алгоритм вычисления эвклидова дерева MST, который использует $N \lg N$ операций сравнения, был оптимальным, пока мы не используем числовые свойства координат; в этом случае мы можем надеяться, что он будет выполняться за линейное время (см. раздел ссылок).

Интересно поразмышлять над отношениями, связывающими графы и геометрические алгоритмы, которая обусловлена решением задачи вычисления эвклидова дерева MST. Многие практические задачи, с которыми нам, возможно, придется столкнуться, могут быть сформулированы либо как геометрические задачи, либо как задачи на графах. Если преобладающим свойством является физическое расположение объектов, то можно обращаться к помощи геометрических алгоритмов, описанных в части 7; однако если фундаментальное значение приобретают взаимосвязи между объектами, алгоритмы на графах, рассмотренные в данном разделе, скорее всего, окажутся более предпочтительными.

Эвклидово дерево MST, по-видимому, приходится на интерфейс между двумя этими подходами (входные данные обладают геометрическими свойствами, а элементы выходных данных взаимосвязаны), и в силу этого обстоятельства разработка простых методов вычисления эвклидова дерева MST остается труднодостижимой целью. В главе 21 мы столкнемся с подобного рода задачей, которая также приходится на этот интерфейс, но там эвклидов подход реализуется через алгоритмы, обладающие куда большим быстродействием, чем соответствующие задачи на графах.

Упражнения

▷ **20.81**. Дайте встречный пример, показывающий, почему не работает следующий метод поиска эвклидова дерева MST: "Сортировать точки по их координатам x, затем найти минимальные основные деревья первой половины и второй половины, затем найти кратчайшие ребра, соединяющие их".

○ **20.82**. Разработайте быструю версию алгоритма Прима для вычисления эвклидового дерева MST для множества точек, равномерно распределенных на плоскости, в основу которой положен принцип игнорирования отдаленных точек до тех пор, пока к ним не приблизится само дерево.

●● **20.83**. Разработайте алгоритм, который при заданном множестве N точек на плоскости находит множество ребер, мощность которого пропорциональна N и в котором обязательно содержится достаточно просто вычисляемое дерево MST, так что можно отдать предпочтение компактной и эффективной реализации алгоритма.

○ **20.84**. Пусть дано случайное множество, состоящее из N (равномерно распределенных) точек в единичном квадрате. Эмпирическим путем определите значение d с точностью до двух десятичных разрядов, такое, что множество ребер, образованных всеми парами точек в пределах расстояния, не превышающего d, с вероятностью 99% содержит дерево MST.

○ **20.85**. Выполните упражнение 20.84 для точек, в которых каждая координата получена путем выборки из распределения Гаусса со средним значением 0.5 и со среднеквадратическим отклонением 0.1.

● **20.86**. Опишите, как вы смогли бы увеличить производительность алгоритмов Крускала и Борувки для случая разреженных эвклидовых графов.

Глава 21

Кратчайшие пути

Каждый путь во взвешенном орграфе ассоциируется с *весом пути* (*path weight*), величиной, представляющей собой сумму весов ребер, составляющих этот путь. Это обстоятельство позволяет нам сформулировать такую задачу как "найти путь между двумя заданными вершинами, имеющий минимальный вес". Задачи о кратчайших путях и являются темой данной главы. Постановки такого рода не только интуитивно подходят для описания многих проблемно-ориентированных приложений, они также вводят нас в тот могучий мир, где мы будем искать эффективные алгоритмы решения задач в общем виде, которые применимы к широкому кругу реальных приложений.

Некоторые из рассматриваемых в данной главе алгоритмов непосредственно связаны с алгоритмами, изучавшимися в главах 17–20. Введенная нами система понятий поиска на графе здесь применима непосредственно, и некоторые из специфических механизмов, использованных в главах 17 и 19 для выражения отношения связности в графах и орграфах, создают основу для решения задач о кратчайших путях.

Для краткости взвешенные орграфы мы будем называть *сетями (networks)*. На рис. 21.1 показан пример сети и ее стандартные представления. Ранее в разделе 20.1 мы уже разработали интерфейс АТД с матрицей смежности и реализацией класса списков смежности — в вызове конструктора следует только передать **true** в качестве второго аргумента так, чтобы класс содержал одно представление каждого ребра, так же как это делалось в главе 19 при получении представлений орграфов из представлений неориентированных графов из главы 17 (см. программы 20.1–20.4).

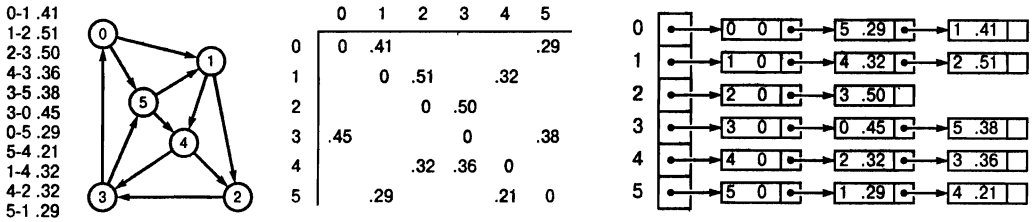


РИСУНОК 21.1. ПРИМЕР СЕТИ И ЕЕ ПРЕДСТАВЛЕНИЯ

Эта сеть (взвешенный орграф) представлена четырьмя различными способами: в виде списка ребер, в графическом виде, с помощью матрицы смежности и в виде списка смежных вершин. Как и для алгоритмов MST, в клетках матрицы и в узлах списка мы показываем веса, но в программах используем указатели на ребра. Хотя на рисунке мы часто изображаем длины ребер пропорциональными их весам (как это делалось для алгоритмов MST), мы не настаиваем на этом правиле, поскольку большинство алгоритмов поиска кратчайших путей могут иметь произвольные неотрицательные веса (отрицательные веса приводят к дополнительным сложностям). Матрица смежности не является симметричной, а списки смежности содержат один узел для каждого ребра (как в невзвешенном орграфе). Несуществующие ребра представляются пустыми клетками в матрице (незаполненные места на рисунке) и вообще отсутствуют в списках (ребер). Петли длины 0 введены потому, что при этом упрощается реализация алгоритмов кратчайших путей. Для экономии места они не представлены в списке ребер слева и нужны для того, чтобы показать типичную последовательность действий, когда мы добавляем их по соглашению, при создании матрицы смежности или представления в виде списков смежности.

Как подробно пояснялось в главе 20, мы используем указатели на абстрактные ребра взвешенного орграфа, чтобы расширить применимость полученных реализаций. При таком подходе для орграфов проявляются некоторые заслуживающие внимания отличия по сравнению с тем, что мы имели для неориентированных графов в разделе 20.1. Во-первых, поскольку имеется только одно представление каждого ребра, нам не нужно вызывать функцию **from()** в классе ребра (см. программу 20.1) при использовании итератора: в орграфе значение **e->from(v)** истинно для каждого указателя ребра **e**, возвращаемого итератором для **v**. Во-вторых, как было показано в главе 19, часто при обработке орграфа полезно иметь возможность работать с его обратным графом, но нам потребуется подход, отличный от того, который был принят в программе 19.1, поскольку та реализация создает ребра, образующие обратный граф, тогда как мы предполагаем, что клиенты АТД графа, предоставляющие указатели на ребра, не должны создавать петель (см. упражнение 21.3).

В приложениях или системах могут потребоваться любые типы графов; на этот случай имеется упражнение, суть которого заключается в разработке такого АТД сети, из которого можно создать производные АТД для невзвешенных неориентированных графов из глав 17 и 18, невзвешенных орграфов из главы 19 или взвешенных неориентированных графов из главы 20 (см. упражнение 21.10).

Когда мы имеем дело с сетями, в общем случае удобно оставлять петли во всех представлениях. Такое соглашение придает алгоритмам гибкость, позволяя использовать порог максимального значения веса, чтобы указать, что из вершины не может исходить ребро, направленное в нее же. В наших примерах мы используем петли веса **0**, хотя петли с положительным весом, конечно, имеют смысл во многих приложениях. Многие приложения также требуют наличия параллельных ребер, возможно, с отличающимися весами.

Как упоминалось в разделе 20.1, различные варианты для игнорирования или создания комбинации подобных ребер присущи и ряду других приложений. Для упрощения в этой главе ни один из наших примеров не использует параллельных ребер, и мы не допускаем параллельных ребер в представлении с помощью матрицы смежности; мы также не проверяем наличия параллельных ребер, и не удаляем их из списков смежности.

Все свойства связности ориентированного графа, которые мы рассматривали в главе 19, остаются справедливыми и для сетей. В той главе нашей целью было выяснить, *возможно* ли достичь одной вершины, отправляясь из другой; в этой главе, принимая во внимание веса, мы будем стремиться найти *наилучший* путь, ведущий из одной вершины в другую.

Определение 21.1. Кратчайшим путем *между двумя вершинами s и t в сети называется такой направленный простой путь из s в t, что никакой другой путь не имеет более низкого веса.*

Это определение лаконично, однако за его краткостью скрываются все достоинства. Во-первых, если t не достижима из s, то никакого пути не существует вообще, а, следовательно, нет и кратчайшего пути. Для удобства рассматриваемые алгоритмы часто трактуют этот случай как эквивалент ситуации, в которой между s и t существует путь с бесконечным весом. Во-вторых, как и в случае алгоритмов MST (Minimal Spanning Tree, минимальное остовное дерево; в литературе также встречается эквивалентное сокращение – SST, Shortest Spanning Tree – *прим. перев.*), мы используем сети, где в примерах веса ребер пропорциональны длинам ребер, однако наше определение не содержит этого требования, поэтому в алгоритмах (кроме одного в разделе 21.5) подобное допущение не делается. Действительно, алгоритмы для наиболее коротких путей предстают в своем лучшем виде, когда они находят алогичные кратчайшие пути, как, например, путь между двумя вершинами, который проходит через несколько других вершин, но имеет общий вес меньше веса ребра, непосредственно соединяющего эти вершины. В-третьих, может существовать несколько путей с одним и тем же весом из одной вершины в другую, и мы обычно удовлетворяемся тем, что выделяем один из них. На рис. 21.2 показан пример с проставленными весами, который иллюстрирует эти замечания.

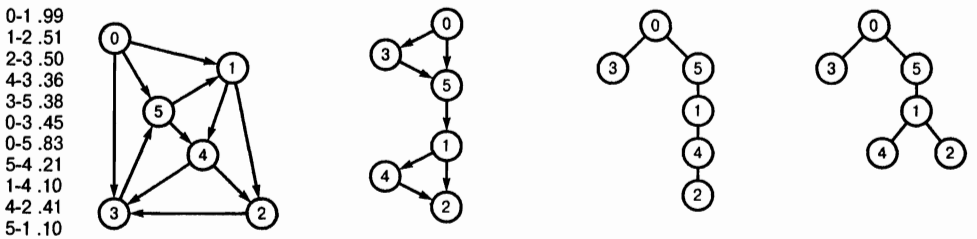


РИСУНОК 21.2. ДЕРЕВЬЯ КРАТЧАЙШИХ ПУТЕЙ

*Дерево кратчайших путей (SPT, Shortest-Path Tree) определяет наиболее короткие пути из корня в другие вершины (см. определение 21.2). В общем случае, различные пути могут иметь одну и ту же длину, так что может существовать несколько SPT, определяющих кратчайшие пути из данной вершины. В сети данного примера, показанной слева, все кратчайшие пути из **0** есть подграфы графа DAG, показанного справа от сети. Дерево с корнем в **0** является подграфом этого DAG тогда и только тогда, когда оно является SPT для **0**. Два дерева справа представляют собой подобные деревья.*

Ограничение на простые пути, присутствующее в определении, несущественно для сетей, содержащих ребра, которые имеют неотрицательный вес, поскольку в подобной сети любой цикл в пути можно удалить, преобразуя путь к такому, который не является более длинным (и даже короче, если цикл не содержит ребер с нулевым весом). Однако в случае, когда мы рассматриваем сети с ребрами, которые могли бы иметь отрицательный вес, необходимость ограничиться простыми путями просматривается без труда: в противном случае понятие кратчайшего пути бессмысленно в случае наличия цикла в сети с отрицательным весом. Например, предположим, что ребро **3-5** в сети на рис. 21.1 имеет вес **-0.38**, а ребро **5-1** — вес **-0.31**. Тогда вес цикла **1-4-3-5-1** должен составить **0.32 + 0.36 − 0.38 − 0.31 = -0.01** и мы могли бы кружить по этому циклу, порождая произвольно короткие пути. Обратите внимание, что *необязательно*, как это имеет место в данном примере, чтобы все ребра в цикле отрицательного веса имели отрицательные веса; значение имеет лишь *сумма* весов ребер. Для краткости ориентированные циклы с общим отрицательным весом мы будем называть *отрицательными циклами*.

Предположим, что в вышеприведенном определении некоторая вершина на пути из s в t принадлежит также некоторому отрицательному циклу. В этом случае существование (непростого) кратчайшего пути из s в t должно вызвать противоречие, поскольку мы могли бы использовать цикл для создания пути, который имел бы вес ниже, чем любое заданное значение. Чтобы устранить такое противоречие, в данном определении мы ограничиваемся простыми путями так, чтобы понятие кратчайшего пути можно было строго определить для любой сети. Тем не менее, до раздела 21.7 мы не рассматриваем отрицательных циклов в сетях, поскольку, как там будет показано, они представляют собой поистине принципиальное препятствие при решении проблем кратчайших путей.

Чтобы найти кратчайшие пути во взвешенном неориентированном графе, мы строим сеть с теми же вершинами и с двумя ребрами (по одному в каждом направлении), которые соответствуют каждому ребру в исходном графе. Существует взаимно однозначное соответствие между простыми путями в сети и простыми путями в графе, а стоимости путей одни и те же; таким образом, проблемы кратчайших путей для них эквивалентны. В самом деле, когда мы строим стандартные списки смежности или представление взвешенного неориентированного графа в виде матрицы смежности, мы строим точно такую же сеть (см., например, рис. 20.3). Такая конструкция не будет продуктивной, если веса могут быть отрицательными, поскольку при этом получаются отрицательные циклы в сети, а мы не знаем, как решать проблемы кратчайших путей в сетях с отрицательными циклами (см. раздел 21.7). Другими словами, алгоритмы для сетей, которые мы излагаем в этой главе, работают также и для взвешенных неориентированных графов.

В определенных приложениях удобно вместо ребер присваивать веса вершинам, либо дополнительно рассматривать веса на ребрах; мы также могли бы рассмотреть более сложные задачи, где имеют значение как количество ребер в пути, так и общий вес пути. Мы можем ставить подобные задачи, формулируя их в терминах сетей со взвешенными ребрами (см., например, упражнение 21.4) или несколько расширив базовые алгоритмы (см., например, упражнение 21.52).

Поскольку различия ясны из контекста, мы не вводим специальную терминологию, позволяющую отличать кратчайшие пути во взвешенных графах от кратчайших путей в графах, которые не имеют весов (где вес пути есть просто число составляющих его ребер — см. раздел 17.7). Поскольку частные задачи, которые могут быть поставлены на

неориентированных или невзвешенных графах, легко решаются с помощью тех же алгоритмов, которые обрабатывают сети, то к сетям (со взвешенными ребрами) применяется обычная терминология, как это и делается в данной главе.

Мы сосредоточимся на тех же базовых задачах, которые в разделе 18.7 были определены для неориентированных и невзвешенных графов. Мы вновь формулируем их здесь, обращая внимание на то, что определение 21.1 неявно обобщает их, с учетом весов в сетях.

Кратчайший путь источник-сток. Для заданной начальной вершины *s* и конечной вершины *t* найти кратчайший путь в графе из *s* в *t*. Мы будем говорить о начальной вершине, как об *источнике (source)*, а о конечной вершине — как о *стоке (sink)*, за исключением тех ситуаций, где такое использование упомянутых терминов конфликтует с определением источников (вершин без входящих ребер) и стоков (вершин без исходящих ребер) в орграфе.

Кратчайшие пути из единственного источника. Для заданной начальной вершины *s* найти кратчайшие пути из *s* во все остальные вершины графа.

Кратчайшие пути между всеми парами вершин. Найти кратчайшие пути, соединяющие каждую пару вершин в графе. Для краткости, при ссылке на это множество из V^2 путей мы иногда будем использовать термин *все кратчайшие пути.*

Если имеется несколько кратчайших путей, соединяющих любую заданную пару вершин, мы довольствуемся любым из них. Поскольку пути имеют различное количество ребер, наши реализации предоставляют функции-элементы, которые позволяют клиентам различать пути за время, пропорциональное длинам путей. Любой кратчайший путь также неявно содержит и собственную длину, но наши реализации выдают длины явно. В итоге, чтобы быть точным, когда в только что данных постановках задач мы говорим "найти кратчайший путь", мы имеем в виду "вычислить длину кратчайшего пути и способ обхода заданного пути за время, пропорциональное его длине".

На рис. 21.3 показаны кратчайшие пути для сети, изображенной на рис. 21.1. В сетях с V вершинами для решения задачи с единственным источником нужно определить V путей, а для решения задачи всех пар потребуется отыскать V^2 путей.

```
      0          .41  0-1        .82  0-5-4-2    .86  0-5-4-3     .50  0-5-4      .29  0-5
1.13  1-4-3-0         1          .51  1-2        .68  1-4-3       .32  1-4       1.06  1-4-3-5
 .95  2-3-0      1.17  2-3-5-1         2        .50  2-3        1.09  2-3-5-4    .88  2-3-5
 .45  3-0         .67  3-5-1      .91  3-5-4-2    0  3           .59  3-5-4      .38  3-5
 .81  4-3-0      1.03  4-3-5-1    .32  4-2        .36  4-3          0  4          .74  4-3-5
1.02  5-4-3-0     .29  5-1        .53  5-4-2      .57  5-4-3       .21  5-4          0  5
```

РИСУНОК 21.3. ВСЕ КРАТЧАЙШИЕ ПУТИ

Эта таблица дает все кратчайшие пути в сети рис. 21.1 вместе с их длинами. Данная сеть является сильно связной, так что в ней существуют пути, соединяющие каждую пару вершин.

Цель алгоритма поиска кратчайшего пути из источника в сток заключается в вычислении одного из элементов (входов) этой таблицы; цель алгоритма поиска кратчайших путей из единственного источника сводится к вычислению одной из строк в этой таблице; наконец, цель алгоритма поиска кратчайших путей между всеми парами предполагает вычисление всей таблица. В общем случае, мы используем более компактные представления, которые содержат по существу ту же информацию и разрешают клиентам обойти любой путь за время, пропорциональное числу его ребер (см. рис. 21.8).

В наших реализациях мы используем более компактное представление, чем эти списки путей; мы ранее уже отмечали это в разделе 18.7, и, кроме того, обсудим его подробно в разделе 21.1.

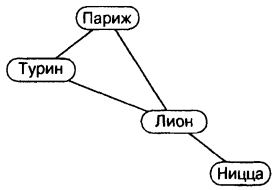
В реализациях на C++ мы строим алгоритмические решения этих задач в виде реализаций АТД, что дает возможность создавать эффективные клиентские программы, которые могут решать разнообразные практические задачи обработки графов. Например, как будет показано в разделе 21.3, мы реализуем классы для определения кратчайших путей для всех пар с помощью конструкторов внутри классов, которые поддерживают запросы на поиск кратчайшего пути за постоянное (линейное) время. Мы также построим классы для решения задачи с единственным источником так, что клиенты, которым необходимо вычислить кратчайшие пути из заданной вершины (или небольшого их множества) могут избежать расходов на вычисления кратчайших путей для других вершин. Внимательное исследование таких результатов и соответствующее использование рассматриваемых нами алгоритмов поможет осознать разницу между эффективным решением практической задачи и решением, которое настолько дорого, что никакой клиент не мог бы позволить себе воспользоваться им.

Задачи о кратчайших путях в различных видах возникают во широком спектре приложений. Многие приложения интуитивно приводят непосредственно к их геометрической интерпретации, тогда как другие вовлекают структуры произвольного вида. Так же, как и в случае минимальных остовных деревьев (MST), которые рассматривались в главе 20, мы иногда будем обращаться к геометрической интерпретации, дабы облегчить понимание алгоритмов решения этих задач, не упуская при этом из виду того факта, что наши алгоритмы действуют должным образом и в более общих структурах. В разделе 21.5 мы рассмотрим специализированные алгоритмы для эвклидовых сетей. Чрезвычайно важно, что в разделах 21.6 и 21.7 будет показано, что базовые алгоритмы эффективны для многочисленных приложений, в которых при помощи сетей представляется абстрактная модель вычислений.

Дорожные карты. Таблицы, которые дают расстояния между всеми парами главных городов, – это замечательная особенность многих дорожных карт. Мы предполагаем, что издатель карты позаботился о том, чтобы расстояния, несомненно, были кратчайшими, но наше допущение не обязательно всегда верно (см., например, упражнение 21.11). В общем случае, такие же таблицы существуют и для неориентированных графов, которые мы будем рассматривать как сети с ребрами в обоих направлениях, соответствующими каждой дороге. Вполне можно подумать об их применении к улицам с односторонним движением — для карт города, а также в отношении ряда аналогичных приложений. Как было показано в разделе 21.3, нетрудно также обеспечить и другую полезную информацию, как, например, таблицу, которая сообщает о том, как выполнить (т.е. пройти) кратчайшие пути (см. рис. 21.4). В современных приложениях встроенные системы автомобилей и других транспортных систем предоставляют возможности подобного рода. Карты являются эвклидовыми графами, и в разделе 21.4 рассматриваются алгоритмы поиска кратчайших путей, которые при поиске учитывают позицию вершины.

Маршруты авиалиний. Карты маршрутов и расписания для авиалиний или других транспортных систем могут быть представлены как сети, для которых различные задачи о кратчайших путях имеют первостепенное значение. Например, может потребоваться минимизировать время, которое займет перелет между двумя городами, или же свести к

минимуму стоимость путешествия. Стоимости, или расходы, в таких сетях могли бы включать функции времени, денег или других интегральных ресурсов. Например, перелеты между двумя городами обычно занимают больше времени в одном направлении, нежели в другом, по причине преобладающих ветров. Авиапассажиры также знают, что стоимость перелета не обязательно является простой функцией расстояния между городами — ситуации, где это будет стоить дешевле, например, использование окольного маршрута (или пересадки) вместо прямого перелета, встречаются довольно-таки часто. Такие усложненные модели могут быть обработаны базовыми алгоритмами поиска кратчайших путей, которые мы рассматриваем в этой главе; эти алгоритмы разработаны в предположении, что все стоимости будут положительными.



	Лион	Ницца	Париж	Турин
Лион		200	400	420
Ницца	200		600	620
Париж	400	600		120
Турин	420	620	120	

	Лион	Ницца	Париж	Турин
Лион		Ницца	Париж	Турин
Ницца	Лион		Лион	Лион
Париж	Лион	Лион		Турин
Турин	Лион	Лион	Париж	

РИСУНОК 21.4. РАССТОЯНИЯ И ПУТИ
Дорожные карты обычно содержат таблицы расстояний наподобие приведенной в центре рисунка для небольшого подмножества французских городов. Города соединяются через шоссе, как показано в верхней части рисунка. Тем не менее, на картах редко встречается таблица, подобная той, что внизу, хотя она была бы также полезной, поскольку отмечает пункты следования, лежащие на кратчайшем пути. Например, чтобы решить, как добраться из Парижа в Ниццу, можно было бы обратиться к такой таблице, из которой видно, что начинать следует с поездки в Лион.

Фундаментальные вычисления кратчайших путей, предлагаемые этими приложениями, — это лишь то, что лежит на поверхности области применимости алгоритмов кратчайших путей. В разделе 21.6 мы рассматриваем задачи из областей применения, которые кажутся не имеющими отношения к данной тематике, в контексте обсуждения вопросов *сведения*, т.е. формального механизма для демонстрации связей между задачами. Мы решаем задачи для этих приложений путем превращения их в абстрактные задачи поиска кратчайших путей, которые не имеют интуитивной геометрической связи с только что описанными задачами. Действительно, некоторые приложения приводят к рассмотрению задач о кратчайших путях в сетях с отрицательными весами. Такие задачи могут оказаться намного более трудноразрешимыми, чем те задачи, где отрицательные веса не допускаются. Задачи о кратчайших путях для таких приложений не только прокладывают мост между элементарными алгоритмами и алгоритмически неразрешимыми задачами, но также подводят нас к мощным и общим механизмам принятия решений.

Как и для алгоритмов MST из главы 20, мы часто смешиваем понятия веса, стоимости и расстояния. Как и там, мы обычно используем естественную привлекательность геометрической интуиции, даже рассматривая более общие постановки задач с произвольными весами ребер; так, мы говорим о "длине" пути и ребра вместо того, чтобы сказать "вес", и говорим, что один путь "короче" другого вместо того, чтобы сказать, что он "имеет меньший вес". Мы также можем сказать, что **v** находится "ближе" к **s**, чем **w**, вместо того, чтобы говорить, что "ориентированный путь наименьшего веса из **s** в **v** имеет вес меньше, чем вес ориентированного пути наименьшего веса из **s** в **w**", и т.д. Это проявляется и в стандартном использовании термина "кратчайшие пути" и выглядит естественным,

даже когда веса не связаны с расстояниями (см. рис. 21.2). Тем не менее, в разделе 21.6, при распространении наших алгоритмов на объекты с отрицательными весами, от подобной практики пришлось отказаться.

Эта глава организована следующим образом. После посвящения в фундаментальные принципы в разделе 21.1, в разделах 21.2 и 21.3 мы исследуем базовые алгоритмы для задач поиска кратчайших путях из единственного источника и между всеми парами. Затем в разделе 21.4 мы рассматриваем ациклические сети (или, сокращенно, взвешенные графы DAG) и в разделе 21.5 — методы использования геометрических свойств для решения задачи источник-сток в эвклидовых графах. Затем в разделах 21.6 и 21.7 мы переключаемся в другом направлении, чтобы рассмотреть более общие задачи, где будут исследоваться алгоритмы поиска кратчайших путей, по мере возможности обращаясь к сетям с отрицательными весами как и к высокоуровневым средствам принятия решений.

Упражнения

▷ **21.1**. Пометьте следующие точки на плоскости от **0** до **5**, соответственно:

(1, 3) (2, 1) (6, 5) (3, 4) (3, 7) (5, 3).

Принимая длины ребер в качестве весов, рассмотрите сеть, определяемую ребрами

1-0 3-5 5-2 3-4 5-1 0-3 0-4 4-2 2-3.

Нарисуйте сеть и дайте структуру списков смежности, которая формируется программой 20.5.

21.2. Покажите в стиле рис. 21.3 все кратчайшие пути в сети, определяемой в упражнении 21.1.

○ **21.3**. Разработайте реализацию класса сети, который представляет обратный взвешенный орграф, определяемый его ребрами. Включите в реализацию класса конструктор "обратного копирования", который принимает граф в качестве аргумента и использует все ребра этого графа для построения его обратного представления.

○ **21.4**. Покажите, что для вычисления кратчайших путей в сетях с неотрицательными весами, как на вершинах, так и на ребрах (где вес пути определен как сумма весов вершин и ребер в пути), достаточно построения АТД сети, который имеет веса только на ребрах.

21.5. Найдите какую-либо доступную большую сеть, содержащую расстояния или стоимости. Возможно, это будет географическая база данных с записями для дорог, соединяющих города, либо графики движения самолетов или поездов.

21.6. На базе программы 17.12 напишите генератор разреженных случайных сетей. Чтобы присвоить ребрам веса, определите АТД ребер случайного веса и напишите две реализации: одна — для генерации равномерно распределенных весов, а другая — для генерации весов в соответствии с распределением Гаусса. Напишите клиентские программы, порождающие разреженные случайные сети для обоих распределений весов с так подобранным набором значений V и E, полученных из различных распределений весов ребер, чтобы можно было использовать их для выполнения эмпирических тестов на графах.

○ **21.7**. Разработайте генератор насыщенных (плотных) случайных сетей, основанный на программе 17.13, и генератор весов ребер по методике, описанной в упражнении 21.6. Напишите клиентские программы, порождающие случайные сети для обоих распределений веса с так подобранным набором значений V и E, чтобы можно было использовать их для выполнения эмпирических тестов на графах, получаемых из упомянутых моделей.

21.8. Реализуйте независимую от представления сети клиентскую функцию, которая строит сеть, получая из стандартного ввода ребра с весами (пары целых чисел из диапазона от **0** до $V-1$ с весами между **0** и **1**).

● **21.9**. Напишите программу, которая генерирует V случайных точек на плоскости, затем строит сеть с ребрами (в обоих направлениях), соединяющими все пары точек, расположенных на расстоянии, которое не превышает заданное d (см. упражнение 17.74), и устанавливает вес каждого ребра равным расстоянию между двумя точками, соединенными этим ребром. Определите, как установить d, чтобы ожидаемое число ребер было равно E.

○ **21.10**. Разработайте базовый класс и производные классы, реализующие АТД для графов, которые могут быть неориентированными или ориентированными, взвешенными или невзвешенными и насыщенными или разреженными.

▷ **21.11**. Назначение следующей таблицы из опубликованной дорожной карты — дать длину кратчайших маршрутов, соединяющих города. Она содержит ошибку. Откорректируйте таблицу. Добавьте также таблицу в стиле рис. 21.4, которая показывает, как проследовать по кратчайшему маршруту.

	Провиденс	**Вестерли**	**Нью-Лондон**	**Норвич**
Провиденс	±	53	54	48
Вестерли	53	±	18	101
Нью-Лондон	54	18	±	12
Норвич	48	101	12	±

21.1 Основные принципы

Наши алгоритмы поиска кратчайших путей базируются на простой операции, известной как *ослабление*, или *релаксация (relaxation)*. В начале выполнения алгоритма поиска кратчайших путей известны только ребра сети и их веса. По мере продолжения мы собираем сведения о кратчайших путях, которые соединяют различные пары вершин. Наши алгоритмы пошагово обновляют эти сведения и делают новые предположения о кратчайших путях, основываясь на информации, полученной к настоящему моменту. На каждом шаге мы проверяем, можно ли найти путь, более короткий, чем некоторый известный путь. Термин *ослабление (релаксация)* обычно используется, чтобы описать этот шаг, который *ослабляет* ограничения вдоль кратчайшего пути. Мы можем представить себе резиновую ленту, плотно натянутую на пути, соединяющем две вершины: успешная операция ослабления разрешает нам ослабить натяжение этой резиновой ленты вдоль более короткого пути.

Наши алгоритмы основываются на многократном применении одного из двух типов операций ослабления:

- *Ослабление ребра.* Проверка, дает ли продвижение вдоль данного ребра новый кратчайший путь к вершине назначения.
- *Ослабление пути.* Проверка, дает ли прохождение через данную вершину новый кратчайший путь, соединяющий две других заданных вершины.

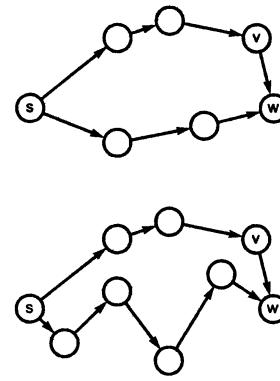


РИСУНОК 21.5. ОСЛАБЛЕНИЕ РЕБРА
Эти диаграммы иллюстрируют операцию ослабления, на которой основываются наши алгоритмы поиска кратчайших путей для единственного источника. Мы прослеживаем известные кратчайшие пути из источника s *в каждую вершину и задаемся вопросом, лежит ли ребро* v-w *на более коротком пути в* w. *На верхнем примере это не выполняется; это значит, что мы должны его (ребро) отвергнуть. На нижнем примере это выполняется; значит, мы должны обновить наши структуры данных, чтобы указать, что лучший известный путь достижения* w *из* s *проходит через* v, *поэтому и принимается* v-w.

Ослабление ребра есть частный случай ослабления пути; тем не менее, мы рассматриваем оба случая как отдельные операции, поскольку используем их порознь (в одном случае это алгоритмы для единственного источника, во втором — алгоритмы для всех пар). В обоих случаях главное требование, которое мы предъявляем к структурам данных, используемым для представления текущего состояния знаний о кратчайших путях сети, состоит в том, что мы должны иметь возможность обновить их, чтобы можно было легко отобразить изменения, связанные с операцией ослабления.

Прежде всего, рассмотрим ослабление ребра, которое иллюстрируется на рис. 21.5. Все рассматриваемые алгоритмы поиска кратчайших путей для единственного источника базируются на таком шаге: приводит ли нас данное ребро к рассмотрению более короткого пути из источника к адресату?

Структуры данных, необходимые для поддержания этих действий, просты. Во-первых, наша основная задача состоит в том, чтобы вычислить длины кратчайших путей из источника в каждую из прочих вершин. Условимся сохранять длины известных кратчайших путей из источника в каждую из вершин в индексированном вершинами векторе **wt**. Во-вторых, для записи самого пути продвижения от вершины к вершине, мы будем придерживаться тех же соглашений, которые учитывались в других алгоритмах поиска на графе в главах 18–20: мы используем индексированный вершиной вектор **spt** для записи последнего ребра на кратчайшем пути из источника в данную индексированную вершину. Эти ребра составляют дерево.

В случае применения таких структур данных реализация ослабления ребра является простой задачей. При поиске кратчайших путей для единственного источника мы используем следующий код, чтобы произвести ослабление вдоль ребра **e**, направленного от **v** к **w**:

```
if (wt[w] > wt[v] + e->wt())
     { wt[w] = wt[v] + e->wt(); spt[w] = e; }
```

Этот фрагмент кода и прост, и выразителен; вместо определения ослабления как высокоуровневой абстрактной операции, мы включаем в наши реализации приведенный выше код без каких-либо изменений.

Определение 21.2. *Пусть задана сеть и некоторая вершина* ***s****,* **деревом кратчайших путей** (**SPT**, Shortest-Paths Tree) *для* ***s*** *будет подсеть, содержащая* ***s*** *и все вершины, достижимые из* ***s****, образующая направленное дерево с корнем в* ***s*** *такое, что каждый путь в дереве является кратчайшим путем в сети.*

Возможно существование нескольких путей одной и той же длины, соединяющих заданную пару узлов, так что SPT не обязательно уникальны. Вообще, как видно из рис. 21.2, если мы выбираем кратчайшие пути из вершины *s* в каждую вершину, достижимую из *s* в сети, и из подсети, порожденной ребрами этих путей, мы можем получить DAG. Различные кратчайшие пути, соединяющие пары узлов, могут рассматриваться как подпути некоторого более длинного пути, содержащего оба узла. Благодаря такой особенности, мы обычно ограничиваемся вычислением любого SPT для данного орграфа и начальной вершины.

Наши алгоритмы в общем случае инициализируют элементы вектора **wt** некоторым сигнальным значением. Упомянутая величина должна быть достаточно малой, чтобы ее добавление при проверке ослабления не вызывало переполнения, и достаточно большой, чтобы никакой простой путь не имел большего веса. Например, если веса ребер находятся между **0** и **1**, в качестве сигнального значения можно выбрать **V**. Имейте в виду, что следует соблюдать особую осторожность при проверке наших допущений, когда сигнальные значения используются в сетях, которые в принципе могут иметь отрицательные веса. Например, если обе вершины имеют сигнальные значения, то код ослабления не выполнит никаких действий, если **e.wt** не отрицательно (что, вероятно, будет иметь место в большинстве реализаций), однако изменит **wt[w]** и **spt[w]**, если этот вес отрицателен.

Наш код всегда использует вершину *назначения (адресат)* в качестве индекса для сохранения ребер SPT (**spt[w]->w() == w**). Для краткости и согласованности с главами 17–19 мы применяем обозначение **st[w]** для ссылки на вершину **spt[w]->v()** (в тексте и особенно на рисунках), дабы подчеркнуть, что вектор **spt** является действительно представлением родительской связи в дереве кратчайших путей (см. рис. 21.6). Вычислить кратчайший путь из *s* в *t* можно, продвигаясь по дереву от *t* к *s*; когда мы делаем это, то проходим по ребрам в направлении, противоположном их направлению в сети, и заходим в вершины пути в обратном порядке (**t**, **st[t]**, **st[st[t]]** и т.д.).

Один из способов получения ребер из SPT на пути в порядке следования от источника к стоку предполагает использование стека. Например, следующий код выводит путь из источника в заданную вершину **w**:

```
stack <EDGE *> P; EDGE *e = spt[w];
while (e) { P.push(e); e = spt[e->v()]); }
if (P.empty()) cout << P.top()->v();
while (!P.empty())
   { cout << "-" << P.top()->w(); P.pop(); }
```

В реализации класса мы могли бы использовать аналогичный код для того, чтобы клиент смог получить вектор, содержащий ребра пути.

Если необходимо просто распечатать или иным способом обработать ребра пути, прохождение всего пути в обратном порядке для достижения первого ребра может оказаться нежелательным. Один из подходов, позволяющих обойти упомянутое затруднение, заключается в том, чтобы работать с обратной сетью, как показано на рис. 21.6.

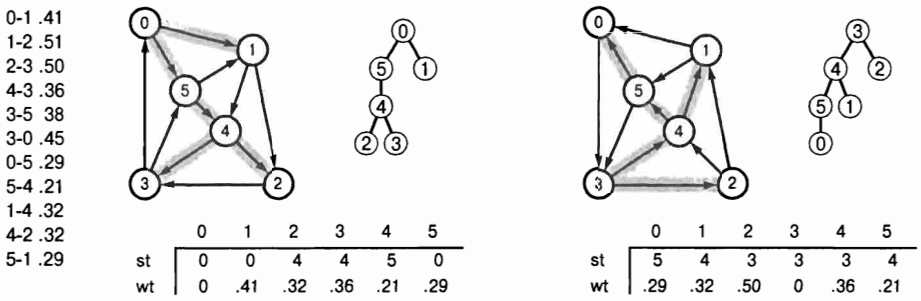


РИСУНОК 21.6. ДЕРЕВЬЯ КРАТЧАЙШИХ ПУТЕЙ

Кратчайшие пути из **0** *к другим узлам в этой сети — это* **0-1**, **0-5-4-2**, **0-5-4-3**, **0-5-4** *и* **0-5**, *соответственно. Эти пути определяют остовное дерево, которое дано в трех представлениях (серые ребра в изображении сети, ориентированное дерево и родительские связи с весами) в центре. Связи в представлении родительских связей (то, что обычно вычисляется), ориентированы в противоположном направлении по сравнению со связями в орграфе, так что мы иногда работаем с обратным орграфом. Остовное дерево, определяемое кратчайшими путями из* **3** *в каждый из других узлов в обратном представлении, показано справа. Представление этого дерева в виде родительских связей дает кратчайшие пути из каждого из других узлов в* **2** *в исходном графе. Например, мы можем найти кратчайший путь* **0-5-4-3** *из* **0** *в* **3**, *следуя по связям* **st[0] = 5**, **st[5] = 4** *и* **st[4] = 3**.

Мы используем обратный порядок и ослабление ребра для задач определения кратчайших путей из единственного источника, поскольку SPT дает компактное представление кратчайших путей из источника во все другие вершины в векторе с точно V входами.

Теперь мы рассмотрим ослабление пути, которое является основой некоторых наших алгоритмов для поиска всех пар: дает ли прохождение через данную вершину более короткий путь, соединяющий две других заданных вершины? Например, предположим, что имеются три вершины *s*, *x* и *t*, и мы хотим знать, что лучше, пройти из *s* в *x*, а затем из *x* в *t*, или сразу пройти из *s* в *t*, не заходя в *x*. Для прямолинейных соединений на эвклидовом пространстве неравенство треугольника говорит нам, что маршрут через *x* не может быть более коротким, чем прямой маршрут из *s* в *t*, но для путей на сети это вполне возможно (см. рис. 21.7). Чтобы определить, как именно обстоят дела, нам нужно знать длины путей из *s* в *x*, из *x* в *t* и остальных, ведущих из *s* в *t* (но не включающих *x*). Затем мы просто проверяем, меньше ли сумма первых двух, чем третий; если это так, то мы должны соответствующим образом обновить наши данные.

Ослабление пути подходит для решения задачи всех пар, где мы сохраняем длины кратчайших путей между всеми парами тех вершин, которые встречались до сих пор. В частности, в коде вычисления кратчайших путей для всех пар этого вида мы поддерживаем вектор векторов **d** такой, что **d[s][t]** есть длина кратчайшего пути из **s** в **t**. Кроме того, мы также поддерживаем вектор векторов **p** такой, что **p[s][t]** есть следующая вершина на кратчайшем пути из **s** в **t**. Первый из них носит название матрицы *расстояний*, а второй — матрицы *путей*. На рис. 21.8 показаны две упомянутых матрицы для сети нашего примера. Матрица расстояний является главной целью вычислений, а матрица путей используется из-за того, что она явно более компактна, хотя и несет ту же информацию, что и полный список путей, который показан на рис. 21.3.

В терминах этих структур данных ослабление пути выражается следующим кодом:

```
if (d[s][t] > d[s][x] + d[x][t])
  { d[s][t] = d[s][x] + d[x][t] ;
    p[s][t] = p[s][x]; }
```

Подобно ослаблению ребра, этот код читается как другая формулировка данного нами неформального описания, так что в реализациях он используется непосредственно. Выражаясь более формально, ослабление пути отражает следующее.

Свойство 21.1. *Если вершина x лежит на кратчайшем пути из s в t, то этот путь складывается из кратчайшего пути из s в x, за которым следует кратчайший путь из x в t.*

Доказательство: Воспользуемся доказательством от противного. Иначе для построения более короткого пути из *s* в *t* мы могли бы воспользоваться любым более коротким путем из *s* в *x* или из *x* в *t*. ■

Мы сталкивались с операцией ослабления пути во время обсуждения алгоритмов транзитивного замыкания в разделе 19.3. Если веса ребер и путей либо равны **1**, либо бесконечны (т.е. вес пути есть **1** только в том случае, если все ребра пути имеют вес **1**), то ослабление пути суть операция, которая использовалась в алгоритме Уоршалла (если существуют пути из *s* в *x* и из *x* в *t*, то существует путь из *s* в *t*). Если мы определяем вес пути как число ребер на этом пути, то алгоритм Уоршалла обобщает алгоритм Флойда для нахождения всех кратчайших путей в невзвешенном орграфе; как будет показано в разделе 21.3, в дальнейшем это обобщается и применительно к сетям.

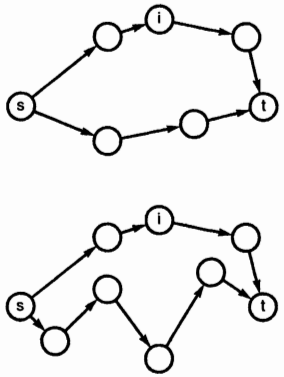


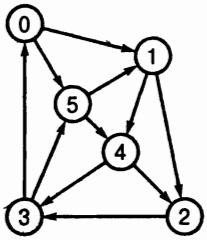
РИСУНОК 21.7. ОСЛАБЛЕНИЕ ПУТИ

*Эти схемы иллюстрируют операцию ослабления, на которой основаны наши алгоритмы кратчайших путей для всех пар. Мы отслеживаем лучший известный путь между всеми парами вершин и задаемся вопросом, является ли вершина **i** такой, что путь, проходящий через нее, явно лучше известного кратчайшего пути из **s** в **t**. На верхнем примере это не так; на примере внизу ответ будет положительным. Каждый раз, когда мы встречаемся с некоторой вершиной **i**, причем такой, что длина известного кратчайшего пути из **s** в **i** плюс длина известного кратчайшего пути из **i** в **t** меньше длины известного кратчайшего пути из **s** в **t**, мы обновляем наши структуры данных, чтобы отметить, что в данный момент нам известен более короткий путь из **s** в **t** (в направлении от источника к **i**).*

Важно обратить внимание, что с точки зрения математика все эти алгоритмы могут быть приведены к общему алгебраическому виду; это унифицирует их описание и помогает в их понимании. С точки зрения программиста важно обратить внимание, что мы можем реализовать каждый из этих алгоритмов, используя абстрактную операцию + (для вычисления веса пути на основе весов ребер) и абстрактную операцию < (для вычисления минимального значения на множестве весов пути), причем обе операции определяются исключительно в контексте операции ослабления (см. упражнения 19.55 и 19.56).

Из свойства 21.1 следует, что кратчайший путь из *s* в *t* содержит кратчайшие пути из *s* в каждую другую вершину вдоль пути в *t*. Большинство алгоритмов поиска кратчайших путей также вычисляют кратчайшие пути из *s* в каждую вершину, которая находится ближе к *s*, чем к *t* (независимо от того, лежит ли вершина на пути из *s* в *t*), хотя это и не обязательно (см. упражнение 21.8).

0-1 .41
1-2 .51
2-3 .50
4-3 .36
3-5 .38
3-0 .45
0-5 .29
5-4 .21
1-4 .32
4-2 .32
5-1 .29



	0	1	2	3	4	5
0	0	.41	.82	.86	.50	.29
1	1.13	0	.51	.68	.32	1.06
2	.95	1.17	0	.50	1.09	.88
3	.45	.67	.91	0	.59	.38
4	.81	1.03	.32	.36	0	.74
5	1.02	.29	.53	.57	.21	0

	0	1	2	3	4	5
0	0	1	5	5	5	5
1	4	1	2	4	4	4
2	3	3	2	3	3	3
3	0	5	5	3	5	5
4	3	3	2	3	4	3
5	4	1	4	4	4	5

РИСУНОК 21.8. ВСЕ КРАТЧАЙШИЕ ПУТИ

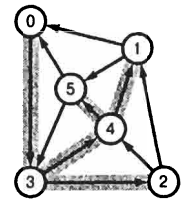
Две матрицы справа суть компактные представления всех кратчайших путей для типовой сети слева. Они содержат ту же информацию, что и список на рис. 21.3. Матрица расстояний слева содержит длину кратчайшего пути: элемент на пересечении строки s и столбца t есть длина кратчайшего пути из s в t. Матрица путей справа содержит информацию, необходимую для того, чтобы пройти по пути: элемент на пересечении строки s и столбца t есть следующая вершина на пути из s в t.

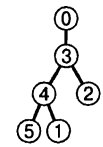
Решение задачи о кратчайших путях источник-сток с помощью такого алгоритма, когда t является наиболее удаленной от s вершиной, эквивалентно решению задачи о кратчайших путях из единственного источника для s. И наоборот, мы могли бы воспользоваться решением задачи о кратчайших путях из единственного источника для s в качестве метода нахождения вершины, наиболее удаленной от s.

Матрица путей, которая используется в наших реализациях для задачи нахождения всех пар, является также представлением деревьев кратчайших путей для каждой вершины. Мы определили **p[s][t]** как вершину, которая следует за s на кратчайшем пути из s в t. Таким образом, это то же, что и вершина, которая в обратной сети предшествует s на кратчайшем пути из t в s. Другими словами, столбец t в матрице путей сети суть индексированный вершиной вектор, который в обратном представлении задает SPT для вершины t. И наоборот, можно построить матрицу путей для сети за счет заполнения каждого столбца индексированным вершиной вектором SPT для соответствующей вершины в обратном представлении. Это соответствие проиллюстрировано на рис. 21.9.

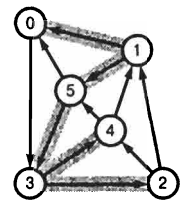
В конечном итоге ослабление дает нам базовые абстрактные операции, которые необходимы для построения алгоритмов поиска кратчайших путей. Основная сложность связана с выбором, сохранять начальное или конечное ребро в кратчайшем пути. Например, алгоритмы для единственного источника более естественно выражаются при сохранении конечного ребра в пути так, что для реконструкции пути потребуется только один индексированный вершинами вектор, поскольку все пути ведут обратно к источнику. Этот выбор не представляет принципиальной трудности, поскольку можно либо использовать обратный граф как обычно, либо создать функции-элементы, которые скрывают это от клиентов. Например, можно было бы определить функцию-элемент в интерфейсной части, которая бы возвращала ребра кратчайшего пути в векторе (см. упражнения 21.15 и 21.16).

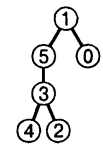
1-0 .41
2-1 .51
3-2 .50
3-4 .36
5-3 .38
0-3 .45
5-0 .29
4-5 .21
4-1 .32
2-4 .32
1-5 .29



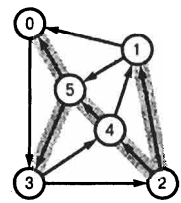


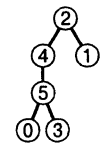
	0	1	2	3	4	5
st	0	4	3	0	3	4
wt	0	1.13	.95	.45	.81	1.02



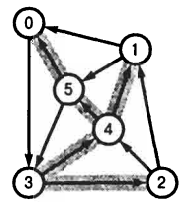


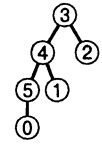
	0	1	2	3	4	5
st	1	1	3	5	3	1
wt	.41	0	1.17	.67	1.03	.29



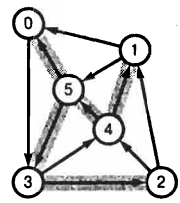


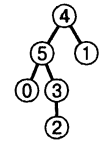
	0	1	2	3	4	5
st	5	2	2	5	2	4
wt	.82	.51	0	.91	.32	.53



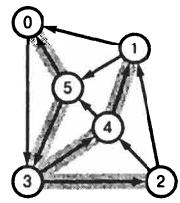


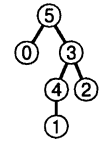
	0	1	2	3	4	5
st	5	4	3	3	3	4
wt	.86	.68	.50	0	.36	.57





	0	1	2	3	4	5
st	5	4	3	5	4	4
wt	.50	.32	1.09	.59	0	.21





	0	1	2	3	4	5
st	5	4	3	5	3	5
wt	.29	1.06	.88	.38	.74	0

РИСУНОК 21.9.
ВСЕ КРАТЧАЙШИЕ ПУТИ В СЕТИ

*Эти диаграммы изображают SPT для каждой вершины в обратном представлении сети из рис. 21.8 (из **0** в **5**, сверху вниз) как поддеревья сети (слева), ориентированные деревья (в центре) и в виде представления родительских связей, включающего индексированное вершинами множество для длин путей (справа). Объединение этих массивов, образующее матрицы путей и расстояний (где каждый массив становится столбцом), дает решение задачи о кратчайших путях для всех пар, которая показана на рис. 21.8.*

Соответственно, для простоты все наши реализации в этой главе включают функцию-элемент **dist**, которая возвращает длину кратчайшего пути и *либо* функцию-элемент **path**, возвращающую начальное ребро в кратчайшем пути, *либо* функцию-элемент **pathR**, которая возвращает конечное ребро в кратчайщем пути. Например, наши программы для единственного источника, которые используют ослабление ребра, обычно реализуют эти функции:

```
Edge *pathR(iirt w) const { return spt[w];}
double dist(int v) { return wt[v]; }
```

Аналогично, и программы для всех путей, которые используют ослабление пути, обычно реализуют эти функции:

```
Edge *path(int s, int t) { return p[s][t]; }
double dist(int s, int t) { return d[s][t]; }
```

В некоторых ситуациях стоило бы построить интерфейсы, основанные на одном или другом либо на обоих упомянутых вариантах, а затем выбрать вручную один из них, который является наиболее естественным для конкретного алгоритма.

Упражнения

▷ **21.12**. Нарисуйте SPT из вершины **0** для сети, определенной в упражнении 21.1, и для ее обратного представления. Дайте представление для обоих деревьев в виде родительских связей.

21.13. Рассмотрите ребра в сети, определенной в упражнении 21.1, как *неориентированные* ребра, такие, что каждому ребру соответствуют ребра равного веса в обоих направлениях в сети. Решите упражнение 21.12 для соответствующей сети.

▷ **21.14**. Измените направление ребра **0-3** на рис. 21.2. Нарисуйте два различных SPT с корнем в **3** для измененной сети.

21.15. Напишите функцию, использующую функцию-элемент **pathR** из реализации задачи для единственного источника, которая помещает в вектор STE указатели на ребра пути из источника **v** в данную вершину **w**.

21.16. Напишите функцию, использующую функцию-элемент **path** из реализации задачи для всех путей, которая помещает в вектор STE указатели на ребра пути из данной вершины **v** в другую заданную вершину **w**.

21.17. Напишите программу, которая использует разработанную функцию из упражнения 21.16, выводящую все пути в стиле рис. 21.3.

21.18. Приведите пример, который показывает, как узнать, какой путь из **s** в **t** является кратчайшим, если неизвестна длина более короткого пути из **s** в **x** для некоторого **x**.

21.2 Алгоритм Дейкстры

В разделе 20.3 мы обсуждали алгоритм Прима для нахождения минимального остовного дерева (MST) взвешенного неориентированного графа: мы строили его, добавляя на каждом шаге одно ребро, всегда выбирая следующее кратчайшее ребро, которое соединяет вершину, включенную в MST, с вершиной еще не входящей в MST. Для вычисления SPT можно воспользоваться приблизительно такой же схемой. Мы начинаем путем размещения источника в SPT, затем строим SPT, добавляя на каждом шаге одно ребро,

всегда выбирая такое из следующих ребер, которое дает кратчайший путь из источника в вершину, не включенную в SPT. Другими словами, мы добавляем вершины к SPT в порядке их расстояния (на дереве SPT) от стартовой вершины. Этот метод известен как *алгоритм Дейкстры (Dijkstra)*.

Как обычно, мы должны делать различие между алгоритмом на уровне абстракции в этом неформальном описании и множеством конкретных реализаций (как, например, в программе 21.1). Эти различия связаны, главным образом, со способом представления графа и реализацией очереди с приоритетами, даже если такие различия не всегда известны из литературных источников. После установки того факта, что алгоритм Дейкстры корректно выполняет вычисление кратчайших путей для единственного источника, мы рассмотрим другие реализации алгоритма и обсудим их взаимоотношения с программой 21.1.

Свойство 21.2 *Алгоритм Дейкстры решает задачу нахождения кратчайших путей для единственного источника на сетях, имеющих неотрицательные веса.*

Доказательство: Для заданной вершины-источника s необходимо установить, что пути в дереве из корня s в каждую вершину x в дереве, вычисленные по алгоритму Дейкстры, соответствуют кратчайшим путям в графе из s в x. Приведенное утверждение доказывается методом индукции. Исходя из предположения, что только что вычисленное поддерево обладает этим свойством, нам нужно просто показать, что добавление новой вершины x приводит к добавлению кратчайшего пути в эту вершину. Однако все другие пути, ведущие в x, должны начинаться путем в дереве и завершаться ребром, исходящим из вершины, не лежащей на дереве. Конструктивно все такие пути являются более длинными, чем данный путь из s в a, что, собственно, и требовалось доказать.

Таким же образом можно показать, что алгоритм Дейкстры решает задачу поиска кратчайших путей в постановке источник-сток, если начинать в источнике и останавливаться, когда сток ставится в очередь с приоритетами. ■

Доказательство потеряло бы силу, если бы веса ребер могли принимать отрицательные значения, поскольку оно предполагает, что по мере добавления ребер к пути длина самого пути не уменьшается. В сети с отрицательными весами ребер это допущение недействительно, поскольку *любое* встретившееся ребро, ведущее в некоторую вершину дерева и имеющее достаточно большой отрицательный вес, могло бы дать более короткий путь, ведущий в эту вершину, нежели путь в дереве. Мы обсуждаем это явление в разделе 21.7 (см. рис. 21.28).

Рисунок 21.10 иллюстрирует эволюцию SPT для типового графа при вычислении по алгоритму Дейкстры, а на рис. 21.11 показан рисунок большего ориентированного дерева SPT. Хотя алгоритм Дейкстры отличается от алгоритма Прима для MST только способом выбора приоритетов, деревья SPT обладают характерными особенностями, отличающими их от MST. У них корень расположен в начальной вершине, и все ребра направлены в сторону от корня, тогда как MST не имеют корня и являются неориентированными. Мы иногда представляем MST как направленные деревья с корнем, например, когда мы используем алгоритм Прима, но такие структуры еще имеют характерные отличия от SPT (сравните представление ориентированного дерева на рис. 20.9 с представлением на рис. 21.11). Действительно, природа SPT отчасти зависит от выбора начальной вершины, как это видно из рис. 21.12.

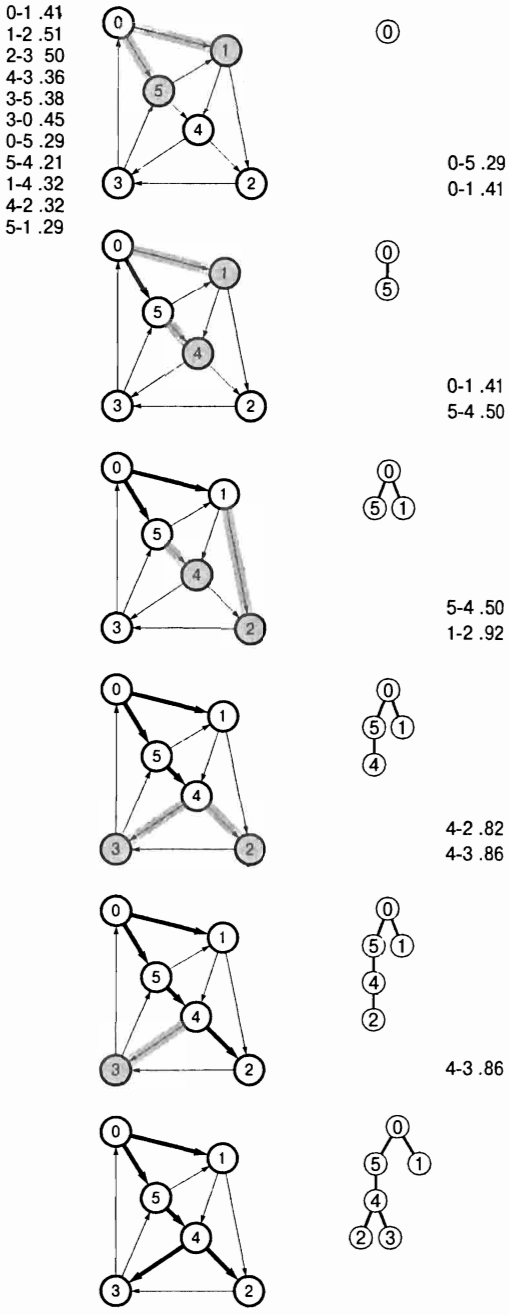


РИСУНОК 21.10. АЛГОРИТМ ДЕЙКСТРЫ

Эта последовательность показывает построение для типовой сети при помощи алгоритма Дейкстры остовного дерева кратчайших путей с корнем в вершине **0**. *Толстыми черными линиями на схемах сетей показаны ребра дерева, а толстыми серыми — ребра бахромы. Представления в виде ориентированного дерева, в процессе его роста показаны в центре, а список ребер бахромы приведен справа.*

На первом шаге мы добавляем **0** *к дереву, а ребра, выходящие из него,* **0-1** *и* **0-5**, *— к бахроме (рисунок вверху). На втором шаге мы перемещаем самое короткое из этих ребер,* **0-5**, *из бахромы в дерево и проверяем ребра, отходящие от него: ребро* **5-4** *добавляется к бахроме, а ребро* **5-1** *удаляется, поскольку оно не является частью пути из* **0** *в* **1**, *более короткого, чем известный путь* **0-1** *(второй рисунок сверху). Приоритет ребра* 5-4 *в бахроме определяется длиной пути из* **0**, *который оно представляет,* **0-5-4**. *На третьем шаге мы перемещаем* **0-1** *из бахромы в дерево, добавляем к бахроме* **1-2** *и удаляем из нее* 1-4 *(третий сверху рисунок). На четвертом мы перемещаем* **5-4** *из бахромы в дерево, добавляем* **4-3** *к бахроме, и заменяем* **1-2** *в* **4-2**, *поскольку путь* **0-5-4-2** *короче, чем* **0-1-2** *(четвертый сверху). Мы держим в бахроме не более одного ребра, ведущего к каждой вершине, выбирая из них то, которое лежит на кратчайшем пути из* **0**. *Вычисления завершаются перемещением* **4-2** *и затем* **4-3** *из бахромы в дерево (рисунок внизу).*

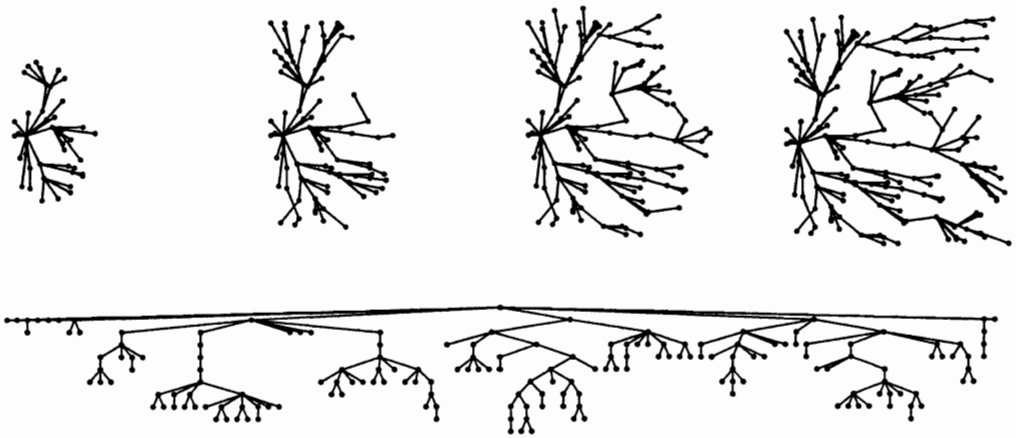


РИСУНОК 21.11. ОСТОВНОЕ ДЕРЕВО КРАТЧАЙШИХ ПУТЕЙ

Этот рисунок иллюстрирует последовательность применения алгоритма Дейкстры для решения задачи о кратчайших путях для единственного источника в случайном эвклидовом орграфе с близкими связями (с ориентированными в обоих направлениях ребрами, соответствующими каждой начерченной линии) — в том же стиле, что и рис. 18.13, 18.24 и 20.9. Дерево поиска имеет характерные черты BFS, поскольку вершины стремятся соединиться друг к другом короткими путями, но оно несколько более вытянутое и менее широкое, поскольку использование расстояний приводит к несколько более длинным путям, нежели использование длин путей.

Исходная реализация Дейкстры, которая подходит для насыщенных графов, в точности соответствует алгоритму Прима для MST. А именно, мы просто изменяем назначение приоритета **P** в программе 20.6 с

```
P = e->wt()
```

(вес ребра) на

```
P = wt[v] + e->wt()
```

(расстояние от источника до ребра назначения). Это изменение приводит к классической реализации алгоритма Дейкстры: на каждом шаге происходит приращение SPT на одно ребро, и каждый раз обновляются расстояния в дереве для всех вершин, смежных с вершиной-адресатом этого ребра. В то же время проверяются все вершины, не включенные в дерево, с целью нахождения такого ребра, которое будет перемещено в дерево, чья вершина-адресат не принадлежит дереву минимальных расстояний от источника.

Свойство 21.3. *С помощью алгоритма Дейкстры можно найти любое SPT в насыщенной сети за линейное время.*

Доказательство: Как и в случае алгоритма Прима для MST, после просмотра кода программы 20.6 становится очевидным, что время выполнения пропорционально V^2, а это линейно для насыщенных графов. ■

Для разреженных графов можно сделать улучшение, рассматривая алгоритм Дейкстры в качестве обобщенного метода поиска на графе, который отличается от поиска в глубину (DFS), от поиска в ширину (BFS) и от алгоритма Прима для MST только правилом добавления ребра к дереву.

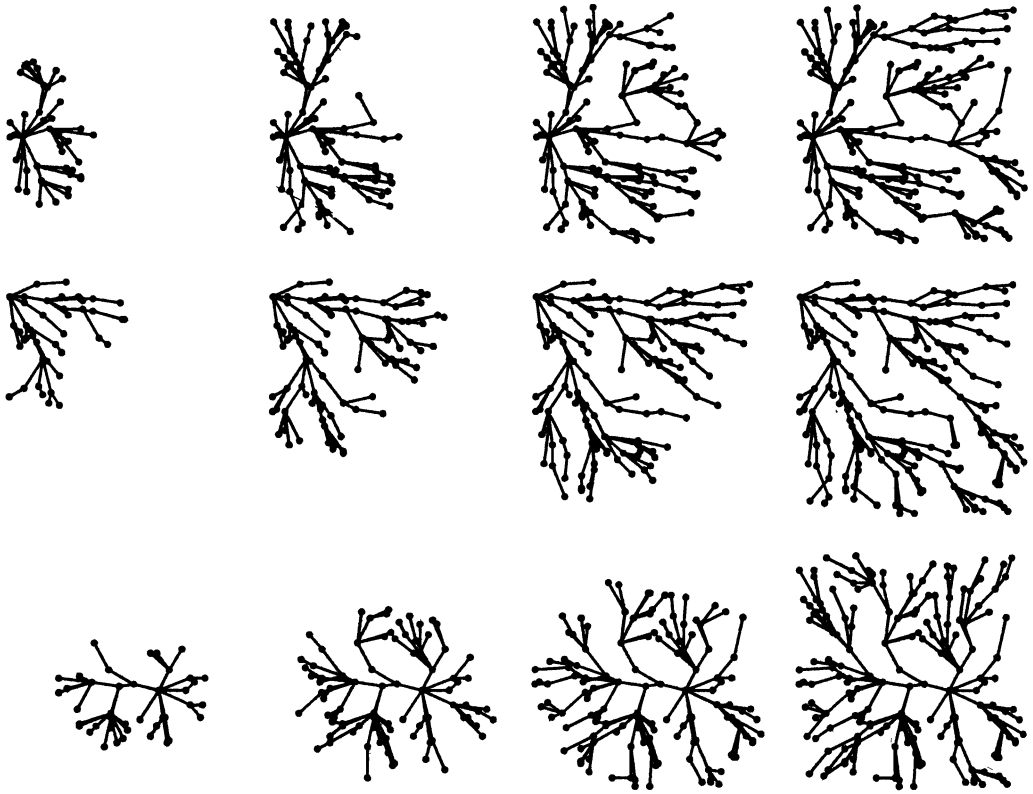


РИСУНОК 21.12. ПРИМЕРЫ SPT

Эти три примера демонстрируют рост SPT для трех различных исходных положений: левое ребро (наверху), верхний левый угол (в центре), и центр (внизу).

Как и в главе 20, ребра, которые соединяют вершины дерева с вершинами, не включенными в дерево, содержатся в обобщенной очереди, называемой *бахромой (fringe)*. Для реализации этой обобщенной очереди используются приоритеты, а также предусматривается настройка приоритетов таким образом, чтобы объединить алгоритмы DFS, BFS и Прима в одной реализации (см. раздел 20.3). Эта схема поиска по приоритетам (PFS, Priority-First Search) содержит в себе также алгоритм Дейкстры. То есть, изменение оператора присваивания **P** в программе 20.7 на

```
P = wt[v] + e->wt()
```

(расстояние от источника до вершины, в которую входит ребро) дает реализацию алгоритма Дейкстры, которая предназначена для разреженных графов.

Программа 21.1. Алгоритм Дейкстры (приоритетный поиск)

Этот класс реализует АТД для кратчайших путей из единственного источника с линейным временем предварительной обработки, приватными данными, требующими пространства памяти, пропорционального *V*, и функциями-элементами с линейным временем выполнения, которые возвращают длину кратчайшего пути и конечную вершину пути из источника в любую заданную вершину. Конструктор реализует алгоритм Дейкстры,

используя для вычисления SPT очередь с приоритетами вершин (в порядке возрастания их расстояния от источника). Интерфейс очереди с приоритетами совпадает с использованным в программе 20.7 и реализован в программе 20.10.

Конструктор предназначен также для обобщенного поиска на графе и реализует другие алгоритмы PFS с другими назначениями приоритетов для **P** *(см. рассуждения по тексту раздела)*. Оператор, присваивающий весам вершин дерева значения **0**, необходим для общей реализации PFS, но не для алгоритма Дейкстры, поскольку приоритеты вершин, добавляемых к SPT, являются неубывающими.

```
template <class Graph, class Edge> class SPT
{ const Graph &G;
  vector<double> wt;
  vector<Edge *> spt;
public:
  SPT(const Graph &G, int s) : G(G),
    spt(G.V()), wt(G.V(), G.V())
  { PQi<double> pQ(G.V(), wt);
    for (int v = 0; v < G.V(); v++) pQ.insert(v);
    wt[s] = 0.0; pQ.lower(s);
    while (!pQ.empty())
    { int v = pQ.getmin(); // wt[v] = 0.0;
      if (v != s && spt[v] == 0) return;
      typename Graph::adjIterator A(G, v);
      for (Edge* e = A.beg(); !A.end(); e = A.nxt())
        { int w = e->w();
          double P = wt[v] + e->wt();
          if (P < wt[w])
            { wt[w] = P; pQ.lower(w); spt[w] = e; }
        }
    }
  }
  Edge *pathR(int v) const { return spt[v]; }
  double dist(int v) const { return wt[v]; }
};
```

Программа 21.1 является альтернативной реализацией PFS для разреженных графов; она несколько проще, чем программа 20.7, и непосредственно соответствует неформальному описанию алгоритма Дейкстры, данному в начале этого раздела. Она отличается от программы 20.7 тем, что в ней инициируется очередь с приоритетами, содержащая все вершины сети, и она поддерживает эту очередь с использованием сигнальных значений для вершин, которые не лежат ни на дереве, ни на бахроме (невидимые вершины с сигнальными значениями). Для сравнения, программа 20.7 содержит в очереди с приоритетами только вершины, связанные с деревом единственным ребром. Хранение всех вершин в очереди упрощает код, но может привести к небольшой потере эффективности для некоторых графов (см. упражнение 21.31).

Общие результаты, рассмотренные в главе 20 и касающиеся производительности поиска по приоритету (PFS), дают нам определенные сведения об эффективности этих реализаций алгоритма Дейкстры для разреженных графов (программа 21.1 и программа 20.7, с соответствующими изменениями). Для справки мы повторно приводим здесь эти результаты. Поскольку доказательства не зависят от функции вычисления приоритета, они повторяются без изменений. Это результаты для наихудшего случая, и они относятся к обеим программам, хотя программа 20.7 может оказаться более эффективной для многих классов графов, поскольку она поддерживает бахрому меньших размеров.

Свойство 21.4. *Для всех сетей и всех приоритетных функций мы можем вычислить остовное дерево с помощью PFS за время, пропорциональное времени, необходимому для V* **вставок (insert)**, *V* **удалений минимума (delete the minimum)** *и применения E операций* **уменьшения ключа (decrease key)** *в очереди с приоритетами размера не более V.*

Доказательство: Этот факт следует непосредственно из реализаций программ 20.7 и 21.1, основанных на очереди с приоритетами. Он дает завышенную верхнюю границу, поскольку размер очереди с приоритетами часто намного меньше *V*, особенно в программе 20.7. ■

Свойство 21.5. *При реализации алгоритма Дейкстры с PFS, использующего полное бинарное дерево для представления очереди с приоритетами, мы можем вычислить любое SPT за время, пропорциональное* $E \lg V$.

Доказательство: Этот результат является прямым следствием свойства 21.4. ■

Свойство 21.6. *Для заданного графа с V вершинами и E ребрами обозначим через d плотность E/V. Если* $d < 2$, *то время выполнения алгоритма Дейкстры пропорционально* $V \lg V$. *В противном случае мы можем улучшить время выполнения не менее чем на коэффициент* $\lg(E/V)$, *т.е.* $O(E \lg_d V)$ *(которое линейно, если E составляет, по крайней мере,* $V^{1+\varepsilon}$), *при использовании для очереди с приоритетами полного* $\lceil E/V \rceil$*-арного дерева.*

Доказательство: Этот результат непосредственно отражает свойство 20.12 и реализацию очереди с приоритетами при помощи нескольких полных бинарных деревьев, которая будет обсуждаться несколько ниже. ■

Таблица 21.1 Алгоритмы поиска по приоритету

Все эти четыре классических алгоритма обработки графов могут быть реализованы при помощи PFS, т.е. обобщенного, основанного на очередях с приоритетами поиска на графе, который строит остовные деревья, перемещая в дерево одно ребро за один шаг. Детали динамики поиска зависят от представления графа, реализации очереди с приоритетами, а также реализации PFS, однако деревья поиска в общем характеризуют различные алгоритмы, как показано на рисунках, на которые приводятся ссылки в четвертом столбце.

Алгоритм	Приоритет	Результат	Рисунок
DFS	Обратное предварительное упорядочивание	Дерево рекурсии	18.13
BFS	Предварительное упорядочивание	SPT (ребра)	18.24
Прима	Вес ребра	MST	20.8
Дейкстры	Вес пути	SPT	21.9

Таблица 21.1 подводит итог известных сведений о четырех основных рассмотренных нами алгоритмах PFS. Они отличаются только используемой функцией вычисления приоритета, и эта разница приводит к остовным деревьям, которые полностью отличаются одно от другого по внешнему виду (как и должно быть). Например, на рисунках, к которым отсылает нас таблица (и для многих других графов), дерево DFS является высоким и тонким, дерево BFS – коротким и толстым; SPT похоже на дерево BFS, но не такое

короткое и толстое, а MST – вообще не короткое и не толстое, равно как и не высокое и не тонкое.

Мы уже рассматривали четыре различных реализации PFS. Первая — это классическая реализация насыщенного графа, которая содержит в себе алгоритм Дейкстры и алгоритм Прима поиска MST (программа 20.6). Три других реализации, отличающиеся содержимым очереди с приоритетами, суть реализации разреженных графов:

- Ребра бахромы (программа 18.10).
- Вершины бахромы (программа 20.7).
- Все вершины (программа 21.1).

Первая реализация имеет, прежде всего, познавательную ценность, тогда как вторая является наиболее совершенной, а третья, возможно, – наиболее простой. В этих рамках уже содержится описание 16 различных реализаций классических алгоритмов поиска на графе — когда мы рассматриваем различные реализации очередей с приоритетами, происходит дальнейшее нарастание возможных вариантов. Это быстрое увеличение сетей, алгоритмов и реализаций подчеркивает полезность общих формулировок производительности в свойствах от 21.4 до 21.6, что также приводится в таблице 21.2.

Таблица 21.2 Стоимость реализаций алгоритма Дейкстры

Эта таблица подводит итог стоимости (наихудшее значение продолжительности исполнения) различных реализаций алгоритма Дейкстры. При соответствующей реализации очередей с приоритетами алгоритм выполняется за линейное время (время, пропорциональное V^2 для насыщенных сетей, и значению E – для разреженных сетей), за исключением очень разреженных сетей.

Алгоритм	Оценка худшего случая	Комментарий
Классический	V^2	Оптимален для насыщенных графов
PFS, заполненное бинарное дерево	$E \lg V$	Самая простая реализация
PFS, полное бинарное дерево для представления бахромы	$E \lg V$	Консервативная верхняя оценка
PFS, полное d-арное дерево разреженный	$E \lg_d V$	Линейный, если граф не слишком

Как правило, фактическое время нахождения кратчайших путей для алгоритмов MST, скорее всего, ниже оценок времени для худшего случая, прежде всего, потому, что большинство ребер не требуют операции *уменьшения ключей*. На практике, за исключением наиболее разреженных графов, мы считаем, что время выполнения не является линейным.

Название *алгоритма Дейкстры* обычно используется для обозначения как абстрактного метода построения SPT путем добавления вершин в порядке их расстояния от источника, так и его реализации как алгоритма со временем, пропорциональным V^2, для построения матрицы смежности, поскольку Дейкстра представил и то, и другое в своей статье, опубликованной в 1959 г. (а также показал, что тем же методом можно вычислить и MST). Увеличение производительности на разреженных графах связано с более поздними усовершенствованиями в технологии АТД и реализациях очереди с приоритетами, которые не являются характерными для задач поиска кратчайших путей. Более высокая про-

изводительность алгоритма Дейкстры – это одно из наиболее важных свойств этой технологии (*см. раздел ссылок*). Как и в случае с MST, для указания на специфические комбинации мы применяем такие термины, как, например, "реализация алгоритма Дейкстры с PFS с использованием полного *d*-арного дерева".

В разделе 18.8 было показано, что в невзвешенных неориентированных графах использование нумерации приоритетов в прямом порядке обхода вершин графа приводит к тому, что очередь с приоритетами действует как очередь FIFO и приводит к BFS. Алгоритм Дейкстры дает нам другую реализацию BFS: когда все веса ребер равны 1, а обход вершин производится в порядке номеров ребер на кратчайшем пути к стартовой вершине. В этом случае очередь с приоритетами не действует в точности как очередь FIFO, поскольку элементы с равными приоритетами не обязательно изымаются из очереди в том порядке, в котором они в нее помещались.

Каждая из этих реализаций помещает в индексированный вершинами вектор **spt** ребра дерева кратчайших путей SPT (Shortest Path Tree) из вершины **0**, а в индексированный вершиной вектор **wt** — длины кратчайших путей в каждую вершину в SPT; кроме того, каждая реализация имеет функции-элементы, обеспечивающие доступ к этим данным со стороны клиентов. Как обычно, имеется возможность построить различные функции обработки графов и классы, основаууst на этих первичных данных (см. упражнения 21.21–21.28).

Упражнения

▷ **21.19**. Покажите в стиле рис. 21.10 результат использования алгоритма Дейкстры для вычисления SPT сети, определяемой в упражнении 21.1 с начальной вершиной **0**.

○ **21.20**. Как бы вы искали *второй* кратчайший путь в сети из *s* в *t*?

21.21. Напишите клиентскую функцию, которая использует объект SPT для нахождения наиболее удаленной вершины от заданной вершины *s* (вершина, для которой кратчайший путь из *s* является наиболее длинным).

21.22. Напишите клиентскую функцию, которая использует объект SPT для вычисления среднего значения длины кратчайших путей из заданной вершины в каждую из вершин, достижимых из нее.

21.23. Разработайте класс, основанный на программе 21.1, с функцией-элементом **path**, которая возвращает **vector**, определенный в библиотеке STL и содержащий указатели на ребра кратчайшего пути, соединяющего *s* и *t* в направлении от *s* к *t*.

▷ **21.24**. Напишите клиентскую функцию, которая использует ваш класс из упражнения 21.23 для распечатки кратчайших путей из заданной вершины во все остальные вершины заданной сети.

21.25. Напишите клиентскую функцию, которая использует объект SPT, чтобы найти все вершины, лежащие в пределах заданного расстояния *d* от заданной вершины в заданной сети. Время выполнения функции должно быть пропорционально размеру подграфа, порождаемого этими и инцидентными им вершинами.

21.26. Разработайте алгоритм для нахождения ребер, устранение которых вызывает максимальное возрастание длины кратчайшего пути из одной заданной вершины в другую в заданной сети.

• **21.27**. Реализуйте класс, который использует объекты SPT для выполнения *анализа чувствительности* ребер сети по отношению к заданной паре вершин *s* и *t*. Вычислите такую матрицу размером *V* на *V*, чтобы для каждого *u* и *v* элемент на пересечении строки *u* и столбца *v* был равен 1, если в сети существует ребро *u*-*v*, вес которого может быть увеличен без увеличения длины кратчайшего пути из *s* в *t* и равен 0 в противном случае.

○ **21.28**. Реализуйте класс, который использует объекты SPT, чтобы отыскать кратчайший путь, соединяющий один заданный набор вершин с другим заданным набором вершин в заданной сети.

21.29. Воспользуйтесь своим решением из упражнения 21.28 для реализации клиентской функции, которая находит кратчайший путь от левого ребра к правому ребру в случайной сети (см. упражнение 20.17).

21.30. Покажите, что MST неориентированного графа эквивалентно *критическому ресурсу* SPT этого графа: для каждой пары вершин *v* и *w* оно дает путь, соединяющий их, в котором наиболее длинное ребро обладает минимально возможной длиной.

21.31. Выполните эмпирические исследования эффективности двух версий алгоритма Дейкстры для разреженных графов, которые описаны в этом разделе (программа 21.1 и программа 20.7, с соответствующим определением приоритета), для различных сетей (см. упражнения 21.4–21.8). Воспользуйтесь реализацией очереди с приоритетами с помощью стандартного полного бинарного дерева.

21.32. Выполните эмпирические исследования с целью получения наилучшего значения *d*, используя реализацию очереди с приоритетами в виде *d*-арного полного дерева (см. программу 20.10) для каждой из трех рассмотренных реализаций алгоритма PFS (программы 18.10, 20.7 и 21.1), примененных на различных сетях (см. упражнения 21.4–21.8).

• **21.33**. Выполните эмпирические исследования для определения эффекта от реализации очереди с приоритетами в виде турнира индексно-сортирующего дерева (см. упражнение 9.53) в программе 21.1 для различных сетей (см. упражнения 21.4–21.8).

○ **21.34**. Выполните эмпирические исследования с целью анализа высоты и средней длины пути в SPT для различных сетей (см. упражнения 21.4–21.8).

21.35. Разработайте класс для задачи поиска кратчайших путей типа источник-сток, который основывается на коде, подобном программе 21.1, но который инициализирует очередь с приоритетами и источником, и стоком. При таком подходе SPT растет из каждой вершины; ваша основная задача — принять решение, что делать, когда два SPT соприкасаются.

• **21.36**. Опишите семейство графов с *V* вершинами и *E* ребрами, для которых достигается худший случай времени выполнения алгоритма Дейкстры.

•• **21.37**. Разработайте приемлемый генератор случайных графов с *V* вершинами и *E* ребрами, для которых время счета PFS-реализации алгоритма Дейкстры с использованием кучи является суперлинейным.

• **21.38**. Напишите клиентскую программу графической анимацией динамики алгоритма Дейкстры. Ваша программа должна создавать изображения, подобные рис. 21.11 (см. упражнения 17.56–17.60). Протестируйте программу на случайных эвклидовых сетях (см. упражнение 21.9).

21.3 Кратчайшие пути между всеми парами

В этом разделе мы рассмотрим два метода решения задачи поиска кратчайших путей для всех пар. Алгоритмы, которые планируется реализовать, непосредственно обобщают два базовых алгоритма, которые были рассмотрены в разделе 19.3 для задачи транзитивного замыкания. Первый метод заключается в выполнении алгоритма Дейкстры из каждой вершины для получения кратчайших путей из этой вершины во все остальные. Если мы реализуем очередь с приоритетами при помощи полнгого бинарного дерева, то при таком подходе время выполнения в худшем случае будет пропорционально $VE \lg V$, однако за счет использования d-арного полного дерева мы можем улучшить эту границу, доведя ее для многих типов сетей до VE. Второй метод, который позволяет напрямую решить данную задачу за время, пропорциональное V^3, является расширением алгоритма Уоршалла, известным как *алгоритм Флойда (Floyd's algorithm)*.

Программа 21.2. АТД поиска кратчайших путей для всех пар

Все наши решения задачи о кратчайших путях для всех пар представляются в виде классов с конструктором и двумя функциями реализации запросов: **dist**, возвращающая длину кратчайшего пути из первого аргумента во второй, и одна из двух возможных функций вычисления пути: либо **path**, возвращающая указатель на первое ребро в кратчайшем пути, либо **pathR**, возвращающая указатель на конечное ребро в кратчайшем пути. Если такого пути не существует, то функция **path** возвращает **0**, а **dist** не определена.

Мы используем функции **path** или **pathR** в зависимости от того, что удобнее для рассматриваемого алгоритма; на практике нам следовало бы поместить одну из них или другую (или обе) в интерфейс, а в реализациях использовать различные функции преобразования, как обсуждалось в разделе 21.1 и в упражнениях в конце этого раздела.

```
template <class Graph, class Edge> class SPall
{
  public:
    SPall(const Graph &);
    Edge *path(int, int) const;
    Edge *pathR(int, int) const;
    double dist(int, int) const;
};
```

Оба класса реализуют интерфейс АТД *абстрактных кратчайших путей* для нахождения кратчайших расстояний и путей. Этот интерфейс, который показан в программе 21.2, является обобщением интерфейса *абстрактного транзитивного замыкания* взвешенного орграфа для выяснения связности в орграфах, который изучался в главе 19. В обеих реализациях класса конструктор решает задачу поиска кратчайших путей для всех пар и сохраняет результат в приватных элементах данных. Кроме того, поддерживаются функции реализации запросов, которые возвращают длину кратчайшего пути из одной заданной вершины в другую, а также первое или последнее ребра в пути. Основное назначение такого АТД состоит в его практическом использовании для реализации алгоритмов поиска кратчайших путей для всех пар.

Программа 21.3. Вычисление диаметра сети

Эта клиентская функция иллюстрирует использование интерфейса из программы 21.2. Она находит наиболее длинный из кратчайших путей в данной сети, распечатывает путь и возвращает его вес (т.е. диаметр сети).

```
template <class Graph, class Edge>
double diameter(Graph &G)
{ int vmax = 0, wmax = 0;
  allSP<Graph, Edge> all(G);
  for (int v = 0; v < G.V(); v++)
    for (int w = 0; w < G.V(); w++)
      if (all.path(v, w))
        if (all.dist(v, w) > all.dist(vmax, wmax))
          { vmax = v; wmax = w; }
  int v = vmax; cout << v;
  while (v != wmax)
    { v = all.path(v, wmax)->w(); cout << "-" << v; }
  return all.dist(vmax, wmax);
}
```

Программа 21.3 представляет собой типовую клиентскую программу, которая при поиске *взвешенного диаметра (weighted diameter)* сети использует интерфейс АТД для поиска всех кратчайших путей. Она проверяет все пары вершин, чтобы найти такую пару, для которой длина кратчайшего пути максимальна; затем она обходит путь, ребро за ребром. На рис. 21.13 показан путь, вычисленный этой программой для нашего примера эвклидовой сети.

Цель алгоритмов этого раздела состоит в обеспечении линейного времени выполнения функций запроса. Как правило, мы ожидаем, что будет огромное количество таких запросов, поэтому мы готовы к существенным затратам ресурсов на представление приватных элементов данных и предварительную обработку в конструкторе, что будет содействовать быстрым ответам на запросы. Оба рассматриваемых нами алгоритма требуют области памяти для приватных элементов данных, пропорциональной V^2.

Основной недостаток этого общего подхода состоит в том, что для огромной сети мы не можем иметь достаточного объема доступной памяти (или не можем предоставить необходимое время на предварительную обработку). В принципе, наш интерфейс предоставляет свободу выбора между временными затратами на предварительную обработку и затратами памяти при обработке запроса. Если мы ожидаем только несколько запросов, предварительную обработку можно не делать, а просто выполнять для каждого запроса единственный алгоритм; тем не менее, существуют ситуации, требующие более совершенных алгоритмов (см. упражнения 21.48–21.50). Эта задача обобщает ту, которой мы посвятили большую часть главы 19 и связана с поддержкой быстрых запросов достижимости при ограниченном пространстве памяти.

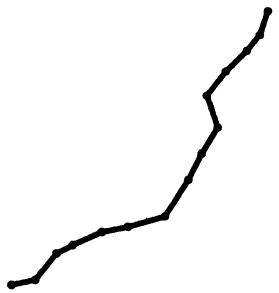


РИСУНОК 21.13. ДИАМЕТР СЕТИ
Наибольший элемент в матрице всех кратчайших путей сети представляет собой диаметр сети. Диаметр сети – это длина наиболее длинного из кратчайших путей, показанных здесь для типовой эвклидовой сети.

Первая рассматриваемая реализация функций АТД поиска кратчайших путей для всех пар решает эту задачу за счет использования алгоритма Дейкстры для решения в отношении каждой вершины задачи с единственным источником. В C++ можно написать этот метод непосредственно, как показано в программе 21.4: для решения задачи с единственным источником для каждой вершины строится один вектор объектов SPT. Этот прием обобщает основанный на BFS метод для невзвешенных неориентированных графов, который рассматривался в разделе 17.7. Это также похоже на использование DFS в программе 19.4, где вычисляется транзитивное замыкание невзвешенного орграфа с началом в каждой вершине.

Программа 21.4. Алгоритм Дейкстры для поиска всех кратчайших путей

Этот класс использует алгоритм Дейкстры построения SPT для каждой вершины так, чтобы можно было вычислить **pathR** и запросить **dist** для любой пары вершин.

```
#include "SPT.cc"
template <class Graph, class Edge> class allSP
{ const Graph &G;
  vector< SPT<Graph, Edge> *> A;
public:
  allSP(const Graph &G) : G(G), A(G.V())
    { for (int s = 0; s < G.V(); s++)
        A[s] = new SPT<Graph, Edge>(G, s); }
  Edge *pathR(int s, int t) const
    { return A[s]->pathR(t); }
  double dist(int s, int t) const
    { return A[s]->dist(t); }
};
```

Свойство 21.7. *С помощью алгоритма Дейкстры можно найти все кратчайшие пути в сети с неотрицательными весами за время, пропорциональное* $VE \log_d V$, *где* $d = 2$, *если* $E < 2V$, *и* $d = E/V$ *в противном случае.*

Доказательство: Непосредственно следует из свойства 21.6. ■

Как и в задачах о кратчайших путях для единственного источника с MST, эта граница завышена; время счета VE оказывается таким же, как и для типовых графов.

Чтобы сравнить эту реализацию с другими, полезно изучить матрицы, скрытые в структуре вектора векторов приватных элементов данных. Векторы **wt** формируют точно такую же матрицу расстояний, как та, которую мы рассматривали в разделе 21.1: элемент на пересечении строки s и столбца t есть длина кратчайшего пути из s в t. Как показано на рис. 21.8 и 21.9, векторы **spt** перенесены из матрицы путей: элемент на пересечении строки s и столбца t представляет собой последний элемент в кратчайшем пути из s в t.

Для насыщенных графов можно было бы использовать представление в виде матрицы смежности и избежать вычисления обратного графа за счет неявного транспонирования матрицы (обмен индексами строк и столбцов), как в программе 19.7. Разработка реализации по этому плану представляет собой интересное упражнение по программированию и приводит к компактной реализации (см. упражнение 21.43); тем не менее, другой подход, который рассматривается немного ниже, допускает даже более компактную реализацию.

Метод выбора для решения задачи поиска кратчайших путей для всех пар в насыщенных графах, который был разработан Р. Флойдом (R. Floyd), в точности совпадает с методом Уоршалла за исключением того, что вместо использования операции логического *или* для отслеживания существования путей, он проверяет расстояния для каждого ребра, чтобы определить, является ли это ребро частью нового, более короткого пути. Действительно, как уже отмечалось, алгоритмы Флойда и Уоршалла идентичны при соответствующей абстрактной постановке (см. разделы 19.3 и 21.1).

Программа 21.5. Алгоритм Флойда для поиска всех кратчайших путей

Эта реализация интерфейса из программы 21.2 использует алгоритм Флойда, представляющий собой обобщение алгоритма Уоршалла (см. программу 19.3), который отыскивает кратчайшие пути между каждой парой точек вместо того, чтобы только проверять их существование.

После инициализации матриц расстояний и путей ребрами графа мы выполняем последовательность операций ослабления для вычисления кратчайших путей. Алгоритм прост в реализации, но проверка того, что он вычисляет кратчайшие пути более сложна *(см. рассуждения по тексту раздела)*.

```
template <class Graph, class Edge> class allSP
{ const Graph &G;
   vector <vector <Edge *> > p;
   vector <vector <double> > d;
public:
   allSP(const Graph &G) : G(G), p(G.V()), d(G.V())
    { int V = G.V();
      for (int i = 0; i < V; i++)
       { p[i].assign(V, 0); d[i].assign(V, V); }
      for (int s = 0; s < V; s++)
       for (int t = 0; t < V; t++)
         if (G.edge(s, t))
          { p[s][t] = G.edge(s, t);
            d[s][t] = G.edge(s, t)->wt(); }
      for (int s = 0; s < V; s++) d[s][s] = 0;
      for (int i = 0; i < V; i++)
       for (int s = 0; s < V; s++)
         if (p[s][i])
          for (int t = 0; t < V; t++)
           if (s != t)
            if (d[s][t] > d[s][i] + d[i][t])
             { p[s][t] = p[s][i];
               d[s][t] = d[s][i] + d[i][t]; }
    }
   Edge *path(int s, int t) const
    { return p[s][t]; }
   double dist(int s, int t) const
    { return d[s][t]; }
};
```

Программа 21.5 представляет функцию АТД поиска кратчайших путей для всех пар, которая реализует алгоритм Флойда. Она явно использует матрицы из раздела 21.1 как приватные элементы данных: вектор V на V векторов **d** для матрицы расстояний и еще один вектор V на V векторов **p** для таблицы путей. Для каждой пары вершин **s** и **t** конструктор заполняет **d[s][t]** длинами кратчайших путей из **s** в **t** (возвращаемых функцией-эле-

ментом **dist**), а **p[s][t]** — индексами следующей вершины на кратчайшем пути из **s** в **t** (возвращаемых функцией-элементом **path**). Реализация основывается на операции ослабления пути, которая рассматривалась в разделе 21.1.

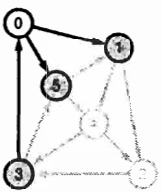
Свойство 21.8. *С помощью алгоритма Флойда можно найти все кратчайшие пути в сети за время, пропорциональное* V^3.

Доказательство: Время выполнения можно вычислить непосредственно из самого кода. Доказательство корректности алгоритма мы проводим точно таким же образом, т.е. методом индукции, как это делалось в отношении алгоритма Уоршалла. *i*-тая итерация цикла вычисляет кратчайший путь из *s* в *t* в сети, которая не включает никаких вершин с индексами больше *i* (кроме, возможно, конечных точек *s* и *t*). Полагая, что это утверждение истинно для *i*-той итерации, мы покажем, что оно истинно для (*i*+1)-й итерации цикла. Любой кратчайший путь из *s* в *t*, который не включает никаких вершин с индексами большими *i*+1, есть либо (*i*) путь из *s* в *t*, не включающий никаких вершин с индексами больше *i* и имеющий длину **d[s][t]**, который был найден на предыдущей итерации цикла по предположению индукции, либо (*ii*) заключает в себе пути из *s* в *i* и из *i* в *t*, ни один из которых не включает никаких вершин с индексами больше *i*, и в этом случае внутренний цикл устанавливает **d[s][t]**. ■

На рис. 21.14 показана детальная трассировка выполнения алгоритма Флойда для нашей типовой сети. Если отметить каждый пустой элемент как **0** (указав на отсутствие ребра), а каждый непустой элемент — как **1** (указав на наличие ребра), то эти матрицы описывают операции алгоритма Уоршалла точно так же, как это было сделано на рис. 19.15. В случае алгоритма Флойда непустые элементы указывают на большее, нежели существование пути, — они дают информацию об известном кратчайшем пути. Элемент в матрице расстояний представляет длину известного кратчайшего пути, который соединяет вершины, соответствующие данной строке и столбцу; соответствующий элемент в матрице путей дает следующую вершину на этом пути. По мере заполнения матриц ненулевыми элементами, алгоритм Уоршалла производит перепроверку, соединяют ли новые пути те пары вершин, которые уже соединены известными путями. В отличие от этого, алгоритм Флойда должен проверить (и при необходимости обновить) каждый новый путь, чтобы убедиться, что его использование приведет к более короткому пути.

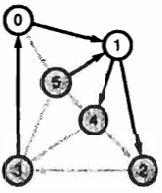
Сравнивая граничные оценки для худшего случая по времени исполнения алгоритмов Дейкстры и Флойда, мы можем прийти к тому же результату для алгоритмов поиска кратчайших путей для всех пар, как это делалось для соответствующих алгоритмов транзитивного замыкания в разделе 19.3. Выполнение алгоритма Дейкстры на каждой вершине, очевидно, является подходящим методом для разреженных сетей, поскольку время счета близко к *VE*. По мере возрастания плотности графа, конкурентоспособным становится алгоритм Флойда, который всегда требует времени, пропорционального V^3 (см. упражнение 21.67); он широко используется ввиду простоты реализации.

Более существенное различие между алгоритмами, которое подробно рассматривается в разделе 21.7, состоит в том, что алгоритм Флойда эффективен даже на сетях, которые имеют отрицательные веса (при условии, что нет отрицательных циклов). Как отмечалось в разделе 21.2, в таких графах метод Дейкстры не обязательно находит кратчайшие пути.



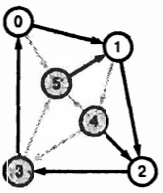
	0	1	2	3	4	5
0	0	.41				.29
1		0	.51		.32	
2			0	.50		
3	.46	**.86**		0		.33
4			.92	.96	0	
5		.29			.21	0

	0	1	2	3	4	5
0	0	1				5
1		1	2		4	
2			2	3		
3	0	**0**		3		5
4			2	3	4	
5		1			4	5



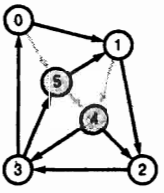
	0	1	2	3	4	5
0	0	.41	**.92**		**.73**	.62
1		0	.51		.32	
2			0	.50		
3	.45	.86	**1.37**	0	**1.18**	.83
4			.92	.96	0	
5		.29	**.80**		.21	0

	0	1	2	3	4	5
0	0	1	**1**		**1**	5
1		1	2		4	
2			2	3		
3	0	0	**0**	3	**0**	5
4			2	3	4	
5		1	**1**		4	5



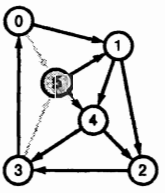
	0	1	2	3	4	5
0	0	.41	.92	**1.42**	.73	.29
1		0	.51	**1.01**	.32	
2			0	.50		
3	.45	.86	1.37	0	1.18	.83
4			.92	.96	0	
5		.29	.80	**1.3**	.21	0

	0	1	2	3	4	5
0	0	1	2	**1**	4	5
1		1	2	**2**	4	
2			2	3		
3	0	0	0	3	0	5
4			2	3	4	
5		1	1	**1**	4	5



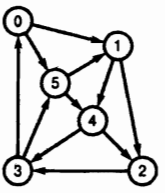
	0	1	2	3	4	5
0	0	.41	.92	1.42	.73	.29
1	**1.46**	0	.51	1.01	.32	**1.39**
2	**.95**	**1.36**	0	.50	**1.68**	**.88**
3	.45	.86	1.37	0	1.18	.83
4	**.81**	**1.22**	.92	.96	0	**.74**
5	**1.75**	.29	.80	1.3	.21	0

	0	1	2	3	4	5
0	0	1	2	1	4	5
1	**2**	1	2	2	4	**2**
2	**3**	**3**	2	3	**3**	**3**
3	0	0	0	3	0	5
4	**3**	**3**	2	3	4	**3**
5	**1**	1	1	1	4	5



	0	1	2	3	4	5
0	0	.41	.92	**1.09**	.73	.29
1	**1.13**	0	.51	**.68**	.32	**1.06**
2	.95	1.36	0	.50	1.68	.88
3	.45	.86	1.37	0	1.18	.83
4	.81	1.22	.92	.96	0	.74
5	**1.02**	.29	**.53**	**.57**	.21	0

	0	1	2	3	4	5
0	0	1	1	1	1	5
1	**4**	1	2	**4**	4	**4**
2	3	3	2	3	3	3
3	0	0	0	3	0	5
4	3	3	2	3	4	3
5	**4**	1	**4**	**4**	4	5



	0	1	2	3	4	5
0	0	.41	**.82**	**.86**	**.50**	.29
1	.13	0	.51	.68	.32	1.06
2	.95	**1.17**	0	.50	**1.09**	.68
3	.45	**.67**	**.91**	0	**.59**	.33
4	.81	**1.03**	.33	.36	0	.74
5	1.02	.29	.53	.57	.21	0

	0	1	2	3	4	5
0	0	1	**5**	**5**	**5**	5
1	4	1	2	4	4	4
2	3	3	3	3	3	3
3	**5**	**5**	**5**	3	**5**	5
4	3	3	2	4	4	3
5	4	1	4	4	4	5

РИСУНОК 21.14. АЛГОРИТМ ФЛОЙДА

Эта последовательность показывает построение матриц кратчайших путей для всех пар с помощью алгоритма Флойда. Для **i** *от* **0** *до* **5** *(сверху вниз), для всех* **s** *и* **t** *мы рассматриваем все пути из* **s** *в* **t**, *не имеющие промежуточных вершин, больших* **i** *(заштрихованные вершины). Вначале единственными такими путями будут ребра сети, так что матрица расстояний (в центре) — это матрица смежности графа, а в матрице путей (справа) для каждого ребра* **s-t** *установлено* **p[s][t] = t**. *Для вершины* **0** *(вверху) алгоритм находит, что* **3-0-1** *короче, чем сигнальное значение, которое говорит об отсутствии ребра* **3-1**, *и соответствующим образом обновляет матрицы. Он не делает этого для такого пути, как, например,* **3-0-5**, *который не короче известного пути* **3-5**. *Далее алгоритм рассматривает пути, проходящие через* **0** *и* **1** *(второй сверху) и находит новые более короткие пути* **0-1-2**, **0-1-4**, **3-0-1-2**, **3-0-1-4** *и* **5-1-2**. *Третий ряд сверху показывает обновления, соответствующие более коротким путям через* **0**, **1**, **2** *и т.д.*

Черные числа, находящиеся поверх серых в матрицах, указывают на ситуации, где алгоритм находит более короткий путь, чем найденный раньше. Например, **0.91** *находится поверх* **1.37** *в строке* **3** *и столбце* **2** *внизу схемы, поскольку алгоритм нашел, что путь* **3-5-4-2** оказался *короче пути* **3-0-1-2**.

Классические решения описанной задачи поиска кратчайших путей для всех пар предполагают, что мы располагаем доступным пространством памяти, достаточным для хранения матриц расстояний и путей. Огромные разреженные графы, для которых мы не можем позволить себе иметь произвольные матрицы V на V, представляют еще одну интересную и перспективную область. Как было показано в главе 19, имеется одна открытая задача, связанная со сведением расхода этого пространства к величине, пропорциональной V, при сохранении поддержки линейного времени запросов на длины кратчайших путей. Мы выяснили, что сходные проблемы трудноразрешимы даже для более простой задачи достижимости (где мы в учебных целях удовлетворились линейным временем независимо от того, есть ли *вообще* какой-либо путь, соединяющий данную пару вершин), так что нельзя ожидать простого решения задачи кратчайших путей для всех пар. Действительно, количество различных кратчайших длин путей в общем случае пропорционально V^2 даже для разреженных графов. Эта величина в некотором смысле служит мерой количества информации, которую требуется обработать, и, возможно, указывает, что если существует ограничение по пространству памяти, то следует ожидать больших временных затрат на каждый запрос (см. упражнения 21.48–21.50).

Упражнения

▷ **21.39**. Оцените с точностью до множителя 10 наибольший размер графа (измеряется количеством вершин), который ваш компьютер и система разработки программного обеспечения могли бы обработать за 10 секунд, если бы для вычисления кратчайших путей использовался алгоритм Флойда.

▷ **21.40**. Оцените с точностью до множителя 10 наибольший размер графа плотности 10 (измеряется количеством ребер), который ваш компьютер и система разработки программного обеспечения могли бы обработать за 10 секунд, если бы для вычисления кратчайших путей использовался алгоритм Дейкстры.

21.41. Покажите в стиле рис. 21.9 результаты применения алгоритма Дейкстры для вычисления всех кратчайших путей сети, определенной в упражнении 21.1.

21.42. Покажите в стиле рис. 21.14 результаты применения алгоритма Флойда для вычисления всех кратчайших путей сети, определенной в упражнении 21.1.

○ **21.43**. Объедините программу 20.6 с программой 21.4 для реализации интерфейса АТД поиска кратчайших путей для всех пар (на базе алгоритма Дейкстры) для насыщенных сетей, который поддерживает запросы пути, но явно не вычисляет обратную сеть. Не определяйте отдельную функцию для единственного источника – поместите код из программы 20.6 непосредственно во внутренний цикл и разместите результаты непосредственно в приватных элементах данных **d** и **p** подобно тому, как это делается в программе 21.5.

21.44. Прогоните эмпирические тесты в стиле таблицы 20.2, для сравнения алгоритма Дейкстры (программа 21.4 и упражнение 21.43) с алгоритмом Флойда (программа 21.5) для различных сетей (см. упражнения 21.4–21.8).

21.45. Прогоните эмпирические тесты для определения, сколько раз алгоритмы Флойда и Дейкстры обновляют значения в матрице расстояний для различных сетей (см. упражнения 21.4–21.8).

21.46. Дайте пример матрицы, в которой элемент в строке *s* и столбце *t* равен количеству различных простых направленных путей, соединяющих *s* и *t* на рис. 21.1.

21.47. Реализуйте класс, конструктор которого вычисляет матрицу числа путей, описанную в упражнении 21.46 так, чтобы за линейное время можно было получить запрос числа через общедоступную функцию-элемент.

21.48. Разработайте реализацию абстрактного класса АТД поиска кратчайших путей для разреженных графов, в которой расход памяти сокращается до величин, пропорциональных *V*, а время запроса увеличивается до значений, пропорциональных *V*.

• **21.49**. Разработайте реализацию абстрактного класса АТД поиска кратчайших путей для разреженных графов, в которой расход памяти существенно меньше $O(V^2)$, но которая поддерживает запросы за время, намного меньшее $O(V)$. *Указание*: Вычислите все кратчайшие пути для подмножества вершин.

• **21.50**. Разработайте реализацию абстрактного класса АТД поиска кратчайших путей для разреженных графов, в которой расход памяти существенно меньше $O(V^2)$, и (с использованием рандомизации) поддерживаются запросы за линейное *ожидаемое* время.

○ **21.51**. Разработайте реализацию абстрактного класса АТД поиска кратчайших путей, в которой используется *отложенный* подход применения алгоритма Дейкстры для построения SPT (и связанного с ним вектора расстояний) из каждой вершины *s*: алгоритм выполняется в первом запросе клиентом кратчайшего пути из *s*, а затем при ссылке на кратчайший путь в последующих запросах.

○ **21.52**. Измените АТД кратчайших путей и алгоритм Дейкстры, чтобы произвести вычисления кратчайших путей в сетях, в которых веса ставятся в соответствие как вершинам, так и ребрам. Не создавайте снова представление графа (этот метод описан в упражнении 21.4), а просто измените код.

21.53. Постройте небольшую модель маршрутов авиалиний и времен перелета, возможно, основанную на некоторых перелетах, совершенных вами. Воспользуйтесь своим решением упражнения 21.52 для вычисления наиболее быстрого пути для перемещения от одного из обслуживаемых адресатов к другому. Затем протестируйте полученную программу на реальных данных (см. упражнение 21.5).

21.4 Кратчайшие пути в ациклических сетях

В главе 19 мы обнаружили, что вопреки нашим предположениям о том, что DAG должны быть проще в обработке, чем обычные орграфы. Разработка алгоритмов с существенно большей производительностью для DAG, нежели для обычного ориентированного графа, является труднодостижимой целью. Для задач о кратчайших путях мы действительно имеем алгоритмы для DAG, которые проще и быстрее, чем методы, основанные на очереди с приоритетами, которые рассматривались для обычных орграфов. В частности, в этом разделе мы исследуем алгоритмы для ациклических сетей, которые:

- Решают задачу для единственного источника за линейное время.
- Решают задачу для всех пар за время, пропорциональное *VE*.
- Решают другие задачи, например, поиск наиболее длинных путей.

В первых двух случаях мы устраняем из времени выполнения логарифмический множитель, который присутствует в наших лучших алгоритмах для разреженных сетей; в третьем случае мы получаем простые алгоритмы решения задач, которые трудноразрешимы для сетей общего вида. Все эти алгоритмы являются прямыми расширениями алгоритмов для достижимости и транзитивного замыкания на DAG, которые рассматривались в главе 19.

Поскольку нет циклов вообще, то нет и отрицательных циклов, так что отрицательные веса не вызывают трудностей в задачах поиска кратчайших путей на DAG. Соответственно, в пределах этого раздела мы не накладываем ограничений на значения весов ребер.

Есть одно замечание, касающееся терминологии: у нас есть выбор, как называть ориентированные графы с весами на ребрах и без циклов — либо *взвешенными DAG*, либо *ациклическими сетями*. Мы применяем оба термина как для того, чтобы попеременно делать ударение на их эквивалентности, так и для того, чтобы избежать путаницы, когда мы обращаемся к литературе, где широко используются оба термина. Иногда удобно использовать первый из них, чтобы сделать акцент на отличии от невзвешенного DAG, когда имеется в виду взвешенность, и второй — чтобы сделать акцент на отличиях от обычных сетей, которые подразумеваются ациклическими.

Вот четыре базовых концепции, примененные нами в главе 19 для получения эффективных алгоритмов для невзвешенных DAG, которые оказываются даже более эффективными на взвешенных DAG:

- Использование DFS при решении задачи с единственным источником.
- Использование очереди источников при решении задачи с единственным источником.
- Вызов любого метода один раз для каждой вершины при решении задачи для всех пар.
- Использование единственного DFS (с динамическим программированием) при решении задачи для всех пар.

Эти методы решают задачу с единственным источником за время, пропорциональное E, и задачу для всех пар — за время, пропорциональное VE. Все они эффективны ввиду топологического упорядочения, которое позволяет вычислять кратчайшие пути для каждой вершины без необходимости повторной проверки предыдущих решений. В этом разделе мы рассматриваем по одной реализации для каждой задачи; остальные мы оставляем читателям на самостоятельную проработку (см. упражнения 21.62–21.65).

Начнем с небольшого ухищрения. Каждый DAG имеет, по крайней мере, один источник, но мог бы иметь несколько, так что вполне естественно обсудить следующую задачу о кратчайших путях.

Кратчайшие пути из нескольких источников. При заданном наборе начальных вершин, для каждой из остальных вершин w найти кратчайший путь среди всех кратчайших путей из каждой начальной вершины в w.

Эта задача по существу эквивалентна задаче о кратчайших путях для единственного источника. Мы можем свести задачу для нескольких источников к задаче для единственного источника, добавив фиктивную вершину-источник с ребрами нулевой длины к каждому источнику в сети. И наоборот, мы можем свести задачу для единственного источника к задаче для нескольких источников, работая с искусственной подсетью,

определяемой всеми вершинами и ребрами, достижимыми из данного источника. Мы редко создаем такие подсети явно, поскольку наши алгоритмы обрабатывают их автоматически, если мы обращаемся с начальной вершиной так, как если бы она была единственным источником в сети (даже если это не так).

Топологическая сортировка предоставляет непосредственное решение задачи кратчайших путей с несколькими источниками и множества других задач. Мы поддерживаем индексированный номерами вершин вектор **wt**, который содержит веса известных кратчайших путей из любого источника в каждую вершину. Чтобы решить задачу поиска кратчайших путей с несколькими источниками, мы инициализируем вектор **wt** значениями **0** для источников и большим сигнальным значением для всех остальных вершин. Затем мы обрабатываем эти вершины в топологическом порядке. Для обработки вершины v, для каждого отходящего ребра v-w мы выполняем операцию ослабления, которая обновляет кратчайший путь в w, если v-w лежит на более коротком пути из источника в w (проходящем через v). В процессе выполнения этих операций проверяются все пути из любой исходной вершины в каждую вершину в графе; операция ослабления отслеживает минимальную длину таких путей, и топологическая сортировка гарантирует, что мы обрабатываем вершины в подходящем направлении.

Мы можем реализовать этот метод непосредственно одним из двух способов. Первый заключается в добавлении нескольких строк кода к коду топологической сортировки в программе 19.8: просто после удаления вершины v из исходной очереди мы выполняем указанную операцию ослабления для каждого из ребер (см. упражнение 21.56). Второй способ предполагает топологическое упорядочение вершин и последующий проход по ним с выполнением операций ослабления так, как описано в предыдущем абзаце.

Программа 21.6. Наиболее длинные пути в ациклической сети

Для нахождения наиболее длинных путей в ациклической сети мы рассматриваем вершины в топологическом порядке, и за счет выполнения для каждого ребра шага ослабления сохраняем в индексированном номером вершины векторе **wt** веса наиболее длинного известного пути в каждую вершину. Вектор **lpt** определяет остовный лес наиболее длинных путей (с корнями в источниках) так, что **path(v)** возвращает последнее ребро в наиболее длинном пути, ведущем в v.

```
#include  "dagTS.cc"
template  <class  Graph,  class  Edge>  class  LPTdag
{  const  Graph  &G;
   vector<double>  wt;
   vector<Edge  *>  lpt;
public:
   LPTdag(const  Graph  &G)  :  G(G),
     lpt(G.V()),  wt(G.V(),  0)
   {  int  j,  w;
     dagTS<Graph>  ts(G);
     for  (int  v  =  ts[j  =  0];  j  <  G.V();  v  =  ts[++j])
     {  typename  Graph::adjIterator  A(G,  v);
        for  (Edge*  e  =  A.beg();  !A.end();  e  =  A.nxt())
         if  (wt[w  =  e->w()]  <  wt[v]  +  e->wt())
           {  wt[w]  =  wt[v]  +  e->wt();  lpt[w]  =  e;  }
     }
   }
   Edge  *pathR(int  v)  const  {  return  lpt[v];  }
   double  dist(int  v)  const  {  return  wt[v];  }
};
```

Подобным же образом (с другими операциями ослабления) можно решить многие задачи обработки графов. Например, программа 21.6 демонстрирует реализацию второго подхода (сортировать, а затем сканировать) для решения задачи *поиска наиболее длинных путей с несколькими источниками*: для каждой вершины в сети определяется, какой путь является наиболее длинным из некоторого источника в эту вершину. Мы интерпретируем элемент **wt**, связываемый с каждой вершиной, как длину наиболее длинного известного пути из любого источника в эту вершину, инициализируем все веса значениями **0** и изменяем смысл сравнения в операции ослабления. На рис. 21.15 показана трассировка выполнения программы 21.6 на типовой ациклической сети.

Свойство 21.9 *Мы можем решить задачу поиска кратчайших путей с несколькими источниками и задачу поиска наиболее длинных путей с несколькими источниками в ациклических сетях за линейное время.*

Доказательство: Одно и то же доказательство сохраняется для наиболее длинного пути, кратчайшего пути и многих других свойств путей. Ориентируясь на программу 21.6, мы проведем доказательство для случая наиболее длинных путей. Мы покажем методом индукции для параметра цикла **i**, что для всех вершин **v = ts [j]** для **j < i**, которые уже были обработаны, **wt[v]** есть длина наиболее длинного пути из источника в **v**. При **v = ts[i]** пусть **t** будет вершиной, предшествующей **v** на некотором пути из источника в **v**. Поскольку вершины в векторе **ts** расположены в топологически отсортированном порядке, то **t** наверняка была уже обработана. По предположению индукции, **wt[t]** представляет собой длину наиболее длинного пути, ведущего в **t**, и шаг ослабления в программе проверяет, будет ли путь в **v** через **t** более длинным. Из предположения индукции также следует, что все пути, ведущие в **v**, будут проверены, как только **v** будет обработана. ■

Это свойство существенно, поскольку оно говорит нам, что обработка ациклических сетей значительно проще, чем обработка сетей, имеющих циклы. Для кратчайших путей этот способ быстрее алгоритма Дейкстры на коэффициент, пропорциональный стоимости операций обработки очереди с приоритетами в алгоритме Дейкстры. В случае задачи о наиболее длинных путях мы имеем линейный алгоритм для ациклических сетей, но трудноразрешимую задачу в случае сетей общего вида. Кроме того, отрицательные веса не представляют здесь дополнительной трудности, однако они являются труднопреодолимым барьером в алгоритмах для сетей общего вида, о чем будет сказано в разделе 21.7.

Только что описанный метод зависит только от того факта, что мы обрабатываем вершины в топологическом порядке. Следовательно, любой алгоритм топологической сортировки можно адаптировать для решения задач о кратчайших и наиболее длинных путях и других задач этого типа (см., например, упражнения 21.56 и 21.62).

Как мы знаем из главы 19, абстракция DAG является общим понятием, используемым во многих приложениях. Например, в разделе 21.6 мы увидим приложение, которое кажется не связанным с сетями, но которое может непосредственно задействовать программу 21.6.

Далее мы снова возвращаемся к задаче о кратчайших путях для всех пар в ациклических сетях. Как и в разделе 19.3, один из методов, которые можно было бы использовать для решения этой задачи, предполагает выполнение в отношении каждой вершины алгоритма для единственного источника (см. упражнение 11.65). Столь же эффективный подход, который мы рассматриваем здесь, заключается в применении единственного DFS с

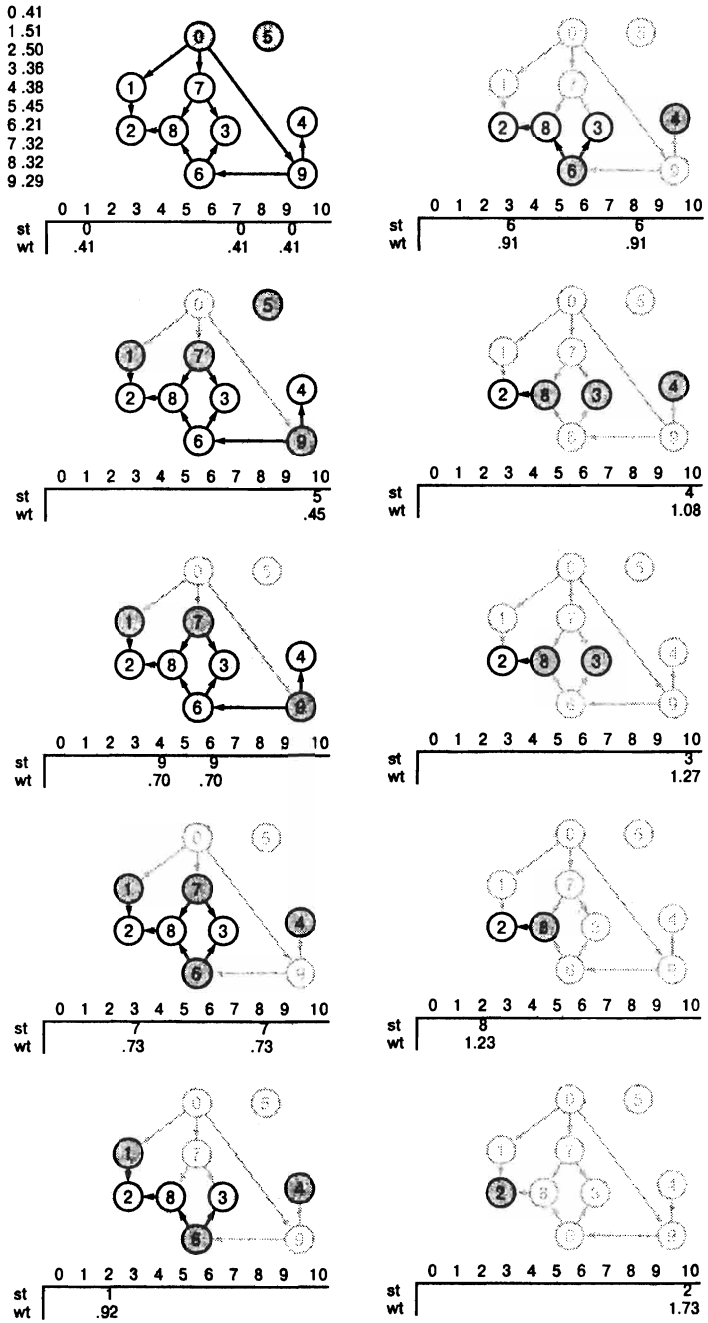


РИСУНОК 21.15. ВЫЧИСЛЕНИЕ НАИБОЛЕЕ ДЛИННЫХ ПУТЕЙ В АЦИКЛИЧЕСКОЙ СЕТИ

В этой сети каждое ребро имеет вес (показанный вверху слева), связываемый с вершиной, из которой оно исходит. Стоки имеют ребра к фиктивной вершине **10**, *которая на рисунках не показана. Массив* **wt** *содержит длину наиболее длинного известного пути в каждую вершину из некоторого источника, а массив* **st** *хранит предыдущую вершину на наиболее длинном пути. Этот рисунок иллюстрирует действие программы 21.6, которое производит выбор из источников (затененные узлы на каждой диаграмме) по дисциплине FIFO, хотя на каждом шаге могли бы выбираться любые источники. Мы начинаем с удаления* **0** *и проверки каждого из ее смежных ребер, обнаруживая однореберные пути длины* **0.41**, *ведущие в* **1**, **7** *и* **9**. *Затем мы удаляем* **5** *и записываем однореберный путь из* **5** *в* **10** *(слева, второй сверху). Далее, мы удаляем* **9** *и записываем пути* **0-9-4** *и* **0-9-6** *длины* **0.70** *(слева, третий сверху). Мы продолжаем эти действия, изменяя массивы каждый раз, когда находим более длинные пути. Например, когда мы удаляем* 7 *(слева, второй снизу), мы записываем пути длины* **0.73**, *ведущие в* **8** *и* **3**; *затем, позже, когда мы удаляем* **6**, *мы записываем более длинные пути (длины* **0.91**) *в* **8** *и* **3** *(справа, вверху). Цель вычислений заключается в нахождении наиболее длинного пути к фиктивному узлу* **10**. *В данном случае результатом будет путь* **0-9-6-8-2** *длины* **1.73**.

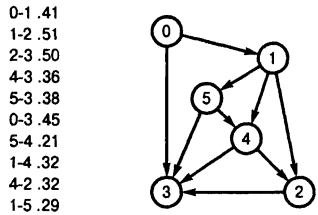
динамическим программированием, подобно тому, как это делалось при вычислении транзитивного замыкания DAG в разделе 19.5 (см. программу 19.9). Если мы рассматриваем вершины в конце рекурсивной функции, то обрабатываем их в топологически обратном порядке и можем получить вектор кратчайшего пути для каждой вершины из векторов кратчайшего пути для каждой соседней вершины, просто используя каждое ребро на шаге ослабления.

Программа 21.7. Все кратчайшие пути в ациклической сети

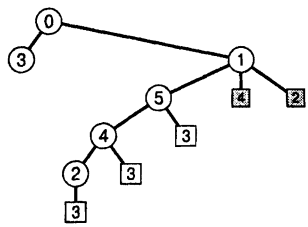
Эта реализация интерфейса из программы 21.2 для взвешенного DAG получена путем добавления соответствующих операций ослабления к основанной на динамическом программировании функции транзитивного замыкания из программы 19.9.

```
template <class Graph, class Edge> class allSPdag
{ const Graph &G;
  vector <vector <Edge *> > p;
  vector <vector <double> > d;
  void dfsR(int s)
  { typename Graph::adjIterator A(G, s);
    for (Edge* e = A.beg(); !A.end(); e = A.nxt())
      { int t = e->w(); double w = e->wt();
        if (d[s][t] > w)
          { d[s][t] = w; p[s][t] = e; }
        if (p[t][t] == 0) dfsR(t);
        for (int i = 0; i < G.V(); i++)
          if (p[t][i])
            if (d[s][i] > w + d[t][i])
              { d[s][i] = w + d[t][i]; p[s][i] = e; }
      }
  }
public:
  allSPdag(const Graph &G) : G(G),
    p(G.V()), d(G.V())
  { int V = G.V();
    for (int i = 0; i < V; i++)
      { p[i].assign(V, 0); d[i].assign(V, V); }
    for (int s = 0; s < V; s++)
      if (p[s][s] == 0) dfsR(s);
  }
  Edge *path(int s, int t) const
    { return p[s][t]; }
  double dist(int s, int t) const
    { return d[s][t]; }
};
```

Программа 21.7 является реализацией сказанного. Применение этой программы к типовому взвешенному DAG иллюстрируется на рис. 21.16. Если не касаться вопроса ослабления, то можно сказать, что есть одно важное различие между этим вычислением и вычислением транзитивного замыкания для DAG: в программе 19.9 у нас была возможность игнорировать ребра в дереве DFS, поскольку они не предоставляют новой информации о достижимости, однако в программе 21.7, тем не менее, мы должны рассмотреть все ребра, поскольку любое ребро могло бы привести к более короткому пути.



	0	1	2	3	4	5
3				0		
2			0	.50		
4			.32	.36	0	
5			.53	.38	.21	0
1		0	.51	.67	.32	.29
0	0	.41	.92	.45	.73	.70



	0	1	2	3	4	5
0	0	.41	.92	.45	.73	.70
1		0	.51	.67	.32	.29
2			0	.50		
3				0		
4			.32	.36	0	
5			.53	.38	.21	0

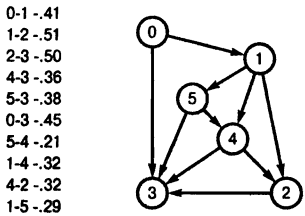
РИСУНОК 21.16. КРАТЧАЙШИЕ ПУТИ В АЦИКЛИЧЕСКОЙ СЕТИ

Эта схема показывает вычисление матрицы (справа внизу) всех кратчайших расстояний для типового взвешенного DAG (вверху слева); каждая строка вычисляется в результате последнего действия в рекурсивной функции DFS. Каждая строка вычисляется из строк для смежных вершин, которые появляются раньше в списке, поскольку строки вычисляются в топологически обратном порядке (обратный порядок обхода дерева DFS, которое изображено слева внизу).
Массив сверху справа показывает строки матрицы в порядке их вычисления. Например, чтобы вычислить каждый элемент в строке для **0**, *мы добавляем* **0.41** *к соответствующему элементу в строке для* **1** *(чтобы получить расстояние к этой вершине из* **0** *после получения* **0-1**), *затем добавляем* **0.45** *к соответствующему элементу в строке для* **3** *(чтобы получить расстояние к этой вершине из* **0** *после получения* **0-3**) *и, наконец, выбираем меньшее из этих двух значений. Вычисления, по существу, оказываются теми же, что и в случае транзитивного замыкания DAG (см., например, рис. 19.23). Наиболее значительное различие между ними состоит в том, что алгоритм транзитивного замыкания мог бы игнорировать ребра (как например,* **1-2** *в этом примере), поскольку они ведут к вершинам, относительно которых известно, что они достижимые, в то время как алгоритм кратчайших путей должен проверить, являются ли пути, связываемые с нижними ребрами, более короткими, чем уже известные пути. Если бы мы должны были игнорировать* **1-2** *в этом примере, то должны были бы пропустить кратчайшие пути* **0-1-2** *и* **1-2**.

Свойство 21.10. *Мы можем решить задачу поиска кратчайших путей для всех пар на ациклических сетях с единственным DFS за время, пропорциональное VE.*

Доказательство: Справедливость этого утверждения вытекает непосредственно из стратегии решения задачи с единственным источником для каждой вершины (см. упражнение 21.65). Мы можем также установить его методом индукции, из программы 21.7. После рекурсивных вызовов для вершины v мы знаем, что вычислены все кратчайшие пути для каждой вершины в списке смежных с v, так что мы можем найти кратчайшие пути из v в каждую вершину за счет проверки каждого ребра, инцидентного v. Мы выполняем V шагов ослабления для каждого ребра, а в общей сложности VE шагов ослабления. ■

Таким образом, топологическая сортировка для ациклических сетей позволяет избежать затрат на очередь с приоритетами в алгоритме Дейкстры. Подобно алгоритму Флойда, программа 21.7 также решает задачи, более общие, чем те, которые решаются алго-



	0	1	2	3	4	5
3				0		
2			0	-.50		
4			-.32	-.82	0	
5			-.53	-1.03	-.21	0
1		0	-.82	-1.32	-.50	-.29
0	0	-.41	-1.23	-1.73	-.91	-.70

РИСУНОК 21.17. ВСЕ НАИБОЛЕЕ ДЛИННЫЕ ПУТИ В АЦИКЛИЧЕСКОЙ СЕТИ

Наш метод для вычисления всех кратчайших путей в ациклических сетях работает даже при наличии отрицательных весов. Следовательно, его можно использовать для вычисления наиболее длинных путей прежде всего за счет инвертирования всех весов, как показано здесь для сети из рис. 21.16. Наиболее длинным простым путем веса **1.73** *в этой сети является* **0-1-5-4-2-3**.

ритмом Дейкстры, поскольку, в отличие от алгоритма Дейкстры (см. раздел 21.7), этот алгоритм работает правильно даже при наличии отрицательных весов ребер. Если мы выполняем этот алгоритм после инвертирования всех весов в ациклической сети, он находит все наиболее длинные пути, как показано на рис. 21.17. Или же можно найти наиболее длинные пути с помощью обращения проверки неравенства в алгоритме ослабления, как в программе 21.6.

Остальные алгоритмы для нахождения кратчайших путей в ациклических сетях, которые упоминаются в начале этого раздела, некоторым образом обобщают методы из главы 19 подобно другим алгоритмам, которые рассматривались в этой главе. Их реализация является путем, приводящим в результате к укреплению вашего понимания как DAG, так и кратчайших путей (см. упражнения 21.62–21.65). Все эти методы выполняются за время, пропорциональное VE в худшем случае, с фактическими затратами, зависящими от структуры DAG. В принципе, мы могли бы дать даже лучшую оценку для некоторых разреженных взвешенных DAG (см. упражнение 19.117).

Упражнения

▷ **21.54**. Постройте решения задач о кратчайших и наиболее длинных путях с несколькими источниками для сети, определенной в упражнении 21.1, с обращенными направлениями для ребер **2-3** и **1-0**.

▷ **21.55**. Модифицируйте программу 21.6 таким образом, чтобы она решала задачу поиска кратчайших путей с несколькими источниками для ациклических сетей.

▷ **21.56**. Реализуйте класс с тем же интерфейсом, что и в программе 21.6, который является производным от базового класса, основанного на очереди источников с топологической сортировкой из программы 19.8, с выполнением операций ослабления для каждой вершины сразу после того, как эта вершина удаляется из исходной очереди.

○ **21.57**. Определите АТД для операции ослабления, постройте его реализацию и модифицируйте программу 11.6 так, чтобы полученный АТД можно было использовать в программе 21.6 для решения задачи о кратчайших путях с несколькими источниками, задачи о наиболее длинных путях с несколькими источниками и других задач, только за счет изменения реализации операции ослабления.

21.58. Воспользуйтесь созданным семейством реализаций из упражнения 21.57 для разработки класса с функциями-элементами, которые возвращают длину наиболее длинного пути из любого источника в любую другую вершину в DAG, длину аналогичного кратчайшего пути и количество вершин, достижимых в таких путях, длины которых расположены внутри заданного интервала.

• **21.59**. Определите свойства ослабления таким образом, чтобы можно было модифицировать доказательство свойства 21.9 с применением абстрактной версии программы 21.6 (как, например, это описано в упражнении 21.57).

▷ **21.60**. Представьте в стиле рис. 21.16 вычисление матриц кратчайших путей для всех пар в сети, определенной в упражнении 21.54, с использованием программы 21.7.

○ **21.61**. Дайте верхнюю границу количества весов ребер, доступных в программе 21.7, как функцию базовых структурных свойств сети. Напишите программу, вычисляющую эту функцию, и воспользуйтесь ею для оценки точности границы *VE* в различных ациклических сетях (добавьте веса так же, как в моделях из главы 19).

○ **21.62**. Напишите основанное на DFS решение задачи о кратчайших путях с несколькими источниками для ациклических сетей. Получите ли вы правильное решение при наличии отрицательных весов ребер? Обоснуйте ответ.

▷ **21.63**. Расширьте решение упражнения 21.62, дав реализацию интерфейса АТД кратчайших путей для всех пар в ациклических сетях, которая строит все пути и матрицы всех расстояний за время, пропорциональное *VE*.

21.64. Представьте вычисления в стиле рис. 21.9 всех кратчайших путей сети, определенной в упражнении 21.54, используя основанный на DFS метод из упражнения 21.63.

▷ **21.65**. Модифицируйте программу 21.6 так, чтобы она решала задачу поиска кратчайших путей для единственного источника в ациклических сетях, затем воспользуйтесь ею для разработки реализации интерфейса АТД кратчайших путей для всех пар в ациклических сетях, который строит все пути и матрицы всех расстояний за время, пропорциональное *VE*.

○ **21.66**. Выполните упражнение 21.61 для основанной на DFS (упражнение 21.63) и для основанной на топологической сортировке (упражнение 21.65) реализации АТД кратчайших путей для всех пар. Какие предположения можно сделать о сравнительных расходах этих трех методов?

21.67. Прогоните эмпирические тесты в стиле таблицы 20.2 с целью сравнения трех реализаций класса для задачи поиска кратчайших путей для всех пар, описанной в этом разделе (см. программу 21.7, упражнение 21.63 и упражнение 21.65), для различных ациклических сетей (добавьте веса, как это делалось в моделях из главы 19).

21.5 Эвклидовы сети

В приложениях, где сеть является моделью карты, наш основной интерес часто направлен на нахождение лучшего маршрута из одной точки в другую. В этом разделе мы рассматриваем стратегию решения упомянутой задачи: быстрый алгоритм для задачи кратчайшего пути источник-сток в *эвклидовых сетях,* т.е. таких, вершины которых являются точками на плоскости и веса ребер которых определяются геометрическими расстояниями между этими точками.

Эти сети обладают двумя важными свойствами, которые не обязательно присущи весам ребер в общем случае. Во-первых, расстояния удовлетворяют неравенству треугольника: расстояние от *s* до *d* никогда не больше, чем расстояние от *s* до *x* плюс расстояние от *x* до *d*. Во-вторых, координаты вершин дают нижнюю границу длины пути: не существует пути из *s* в *d*, который был бы короче, чем расстояние между *s* и *d*. Алгоритм поиска кратчайших путей для задачи источник-сток, который мы рассматриваем в этом разделе, использует особенности этих двух свойств, что улучшает его эффективность.

Часто эвклидовы сети также *симметричны*: их ребра являются двунаправленными. Как упоминалось в начале этой главы, такие сети возникают всякий раз, когда, например, мы интерпретируем представление взвешенного неориентированного эвклидового графа в форме матрицы смежности или списков смежности (см. раздел 20.7) как взвешенный орграф (сеть). Когда мы рисуем неориентированную эвклидову сеть, мы делаем это ввиду того, что при такой интерпретации не происходит размножения стрелок на рисунках.

Основная идея проста: поиск по приоритету (PFS) предоставляет нам общий механизм для поиска путей в графах. С помощью алгоритма Дейкстры мы рассматриваем пути в порядке увеличения их расстояния от начальной вершины. Это упорядочение гарантирует, что, когда мы достигнем стока, будут рассмотрены все более короткие пути в графе, ни один из которых не ведет в сток. Но в эвклидовом графе мы располагаем дополнительной информацией: если мы ищем путь из источника *s* в сток *d* и при этом проходим через третью вершину *v*, то мы знаем, что нам не только необходимо учесть путь, найденный из *s* в *v*, но также и то, что лучшее, на что можно рассчитывать на пути из *v* в *d*, это, во-первых, взять вес ребра *v*-*w* и затем вычислить путь, длина которого равна прямолинейному расстоянию от *w* до *d* (см. рис. 21.18). В случае приоритетного поиска мы свободно можем принять во внимание эту дополнительную информацию для повышения эффективности вычислений. Мы используем стандартный алгоритм, но в качестве приоритета для каждого ребра *v*-*w* принимаем сумму следующих трех величин: длина известного пути из *s* в *v*, вес ребра *v*-*w*, и расстояние от *w* до *t*. Если мы всегда выбираем ребро, для которого это число наименьшее, то при достижении *t* можем быть уверены, что в данном графе не существует более короткого пути из *s* в *t*. Кроме того, в типовых сетях мы получаем этот результат после выполнения гораздо меньшего количества попыток, чем пришлось бы сделать в алгоритме Дейкстры.

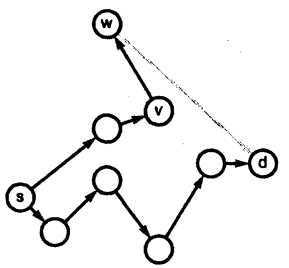


РИСУНОК 21.18. ОСЛАБЛЕНИЕ РЕБРА (ЭВКЛИДОВО)

*Когда мы вычисляем кратчайшие пути в эвклидовом графе, в операции ослабления мы можем учитывать расстояния до адресата. На этом примере мы могли бы сделать вывод, что представленный путь из **s** в **v**, плюс **v**-**w** не может привести к более короткому пути из **s** в **d**, чем уже найденный, **поскольку** длина любого такого пути должна быть, по крайней мере, длиной пути из **s** в **v** плюс длина **v**-**w** и плюс прямолинейное расстояние от **w** до **d**, что больше длины известного пути из **s** в **d**. Подобного рода проверки могут существенно уменьшить количество путей, которые необходимо рассмотреть.*

Для реализации данного подхода мы используем стандартную PFS-реализацию алгоритма Дейкстры (см. программу 21.1, а также упражнение 21.73, поскольку эвклидовы графы обычно разрежены) с двумя изменениями. Во-первых, вместо инициализации **wt[s]** в начале поиска значениями **0.0** он заполняется величинами

distance(s, d), где **distance()** есть функция, возвращающая расстояние между двумя вершинами. Во-вторых, мы определяем приоритет **P** как функцию:

```
(wt[v] + e->wt() + distance(w, d) - distance(v, d))
```

вместо функции **(wt[v] + e->wt0)**, которая использовалась в программе 21.1 (вспомните, что **v** и **w** — это локальные переменные, которые заполняются, соответственно, величинами **e->v()** и **e->w()**). Приведенные изменения, которые мы называем также эвклидовой эвристикой, поддерживают такой инвариант, как величина **wt[v] - distance(v, d)**, являющийся длиной кратчайшего пути в сети из **s** в **v** для каждой вершины **v** дерева (и, следовательно, **wt[v]** представляет собой нижнюю границу длины возможного кратчайшего пути из **s** в **d** через **v**). Мы вычисляем **wt[w]** за счет добавления к этой величине веса (расстояние до **w**) ребра плюс расстояние от **w** до стока **d**.

Свойство 21.11. *Приоритетный поиск с эвклидовой эвристикой решает задачу кратчайших путей источник-сток в эвклидовых графах.*

Доказательство: Здесь применяется доказательство свойства 21.2. Когда мы добавляем вершину **x** к дереву, добавление к приоритету расстояния из **x** в **d** не влияет на доказательство того, что путь в дереве из **s** в **x** является кратчайшим путем в графе из **s** в **x**, поскольку к длине всех путей, ведущих в **x**, добавляется одна и та же величина. При добавлении **d** к дереву мы знаем, что никакой другой путь из **s** в **d** не короче, чем данный путь в дереве, поскольку любой такой путь должен состоять из некоторого пути в дереве, за которым следует ребро, ведущее в некоторую вершину **w**, которая не лежит на дереве, и завершаться путем из **w** в **d** (длина которого не может быть короче расстояния из **w** в **d**). Однако, по построению мы знаем, что длина пути из **s** в **w** плюс расстояние из **w** в **d** не меньше, чем длина пути в дереве из **s** в **d**. ■

В разделе 21.6 мы обсудим другой простой способ реализации эвклидовой эвристики. Прежде всего, мы делаем проход по графу, изменяя вес каждого ребра: для каждого ребра **v-w** мы добавляем величину **distance(w, d) - distance(v, d)**. Затем мы выполняем стандартный алгоритм поиска кратчайшего пути, начиная с **s** (с **wt[s]**, инициализированного значениями **distance(s, d)**), и останавливаемся по достижении **d**. Этот метод в вычислительном отношении эквивалентен уже описанному методу (который в принципе по ходу вычислений считает те же веса) и является типовым примером базовой операции, известной как *повторное взвешивание (reweighting)* сети. Повторное взвешивание играет существенную роль при решении задач поиска кратчайших путей с отрицательными весами; мы обсудим его подробно в разделе 21.6.

Эвклидова эвристика оказывает влияние на эффективность, но не на правильность алгоритма Дейкстры для вычисления кратчайших путей в модели источник-сток. Как говорилось в доказательстве свойства 21.2, применение стандартного алгоритма для решения задачи источник-сток означает построение SPT, в котором все вершины ближе к началу, чем сток **d**. При использовании эвклидовой эвристики SPT содержит только те вершины, для которых длина пути из **s** плюс расстояние до **d** меньше, чем длина кратчайшего пути из **s** в **d**. Мы полагаем, что это дерево будет существенно меньшим для многих приложений, поскольку при данной эвристике отбрасывается значительное число длинных путей. Точная экономия зависит от структуры графа и геометрии расположения вершин. На рис. 21.19 показано действие эвклидовой эвристики на нашем типовом графе, когда достигается существенная экономия. Мы называем этот метод эвристическим,

поскольку нет гарантии, что какая-либо экономия вообще будет иметь место: всегда возможен случай, когда имеется единственный достаточно длинный путь из источника в сток, который перед возвратом в сток отклонится произвольно далеко в сторону от источника (см. упражнение 21.80).

Рисунок 21.20 иллюстрирует интуитивное описание базовой геометрии, лежащей в основе эвклидовой эвристики: если длина кратчайшего пути из s в d есть z, то вершины, просматриваемые этим алгоритмом, лежат в основном внутри эллипса, определяемого как местоположение тех точек x, для которых расстояние от s до x плюс расстояние от x до d равняется z. Для типовых эвклидовых графов мы ожидаем, что количество вершин в этом эллипсе намного меньше, чем количество вершин в круге радиуса z с центром в источнике (тех, которые должны просматриваться алгоритмом Дейкстры).

Строгий анализ получаемой экономии является трудной аналитической задачей и зависит как от конфигурации наборов случайных точек, так и от вида случайных графов (*см. раздел ссылок*). Для типовых ситуаций мы ожидаем, что если в стандартном алгоритме при вычислении кратчайшего пути источник-сток рассматривается X вершин, то эвклидова эвристика сократит расход ресурсов, доведя его до величины, пропорциональной $\sqrt{X}$, что приведет ожидаемое время выполнения к величине, пропорциональной V для насыщенных и $\sqrt{V}$ — для разреженных графов. Этот пример показывает, что трудность разработки подходящей модели или анализ связанных с нею алгоритмов ни в коем случае не должны разубеждать нас от использования преимуществ существенной экономии, которая будет достигаться во многих приложениях, особенно, когда реализация тривиальна.

РИСУНОК 21.19. КРАТЧАЙШИЙ ПУТЬ В ЭВКЛИДОВОМ ГРАФЕ

Во время поиска кратчайшего пути в вершину назначения можно ограничить поиск вершин внутри относительно малого эллипса, примыкающего к пути, как показано на этих трех примерах, на которых изображено поддерево SPT из примеров на рис. 21.12.

Доказательство свойства 21.11 применимо в отношении *любой* функции, которая дает нижнюю границу расстояния от каждой вершины до d. Существуют ли другие функции, для которых алгоритм будет рассматривать даже меньше вершин, чем с использованием эвклидовой эвристики? Этот вопрос изучался в общей постановке применительно к широкому классу алгоритмов комбинаторного поиска. Действительно, эвклидова эвристика является характерным примером алгоритма, называемого A* (произносится "эй-стар"). Из теории известно, что *оптимальным* будет использование функции, дающей наилучшую возможную нижнюю границу; другими словами, чем лучше эта граничная функция, тем более эффективным окажется поиск. В данном случае оптимальность A* говорит о том, что при использовании эвклидовой эвристики,

несомненно, будет просматриваться не больше вершин, чем в случае алгоритма Дейкстры (который представляет собой A* с нулевой нижней границей). Результаты аналитических исследований дают более точную информацию для конкретных сетевых моделей.

Свойства эвклидовых сетей можно также использовать и для построения абстрактных АТД поиска кратчайших путей, более эффективно реализующих компромисс между используемым временем и пространством, нежели для сетей общего вида (см. упражнения 21.48–21.50). Такие алгоритмы важны в приложениях, подобных обработке карт, где сети огромны и разрежены. Например, предположим, что требуется разработать навигационную систему, определяющую кратчайшие пути на карте с миллионами дорог. Возможно, хотелось бы хранить карту непосредственно в малом бортовом компьютере, однако расстояния и матрицы путей зачастую слишком велики, чтобы их можно было втиснуть в память (см. упражнения 21.39 и 21.40); следовательно, алгоритмы для поиска всех путей из раздела 21.3 не применимы. Алгоритм Дейкстры также не может дать ответа за достаточно короткое время для случая огромных карт. Упражнения 21.77 и 21.78 исследуют стратегии рационального соотношения между объемом предварительной обработки и объемом памяти, которые обеспечивают быструю реакцию на запросы о кратчайших путях в модели источник-сток.

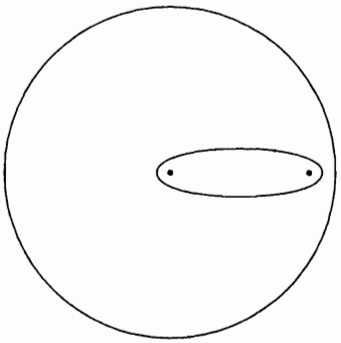


РИСУНОК 21.20. ЭВКЛИДОВА ЭВРИСТИКА И ГРАНИЦЫ СТОИМОСТИ

Когда нам нужно отыскать кратчайший путь в вершину-адресат, при поиске мы можем ограничиться вершинами внутри эллипса, описанного вокруг пути, вместо круга с центром в **s**, *что требовалось бы в алгоритме Дейкстры. Радиус круга и форма эллипса определяются длиной кратчайшего пути.*

Упражнения

• **21.68**. Найдите большой эвклидов граф, возможно, карту с таблицей расстояний между пунктами, телефонную сеть со стоимостями переговоров или расписание авиарейсов с указанными стоимостями билетов.

21.69. Используя стратегии, описываемые в упражнениях от 17.71 до 17.73, напишите программы, которые генерируют случайные эвклидовы графы за счет соединения вершин, упорядоченных на решетке $\sqrt{V}$ на $\sqrt{V}$.

▷ **21.70**. Покажите, что частичное SPT, вычисленное с использованием эвклидовой эвристики, не зависит от значений, которыми инициализируется **wt[s]**. Объясните, как вычислять длины кратчайших путей из начального значения.

▷ **21.71**. Покажите в стиле рис. 21.10 каков будет результат, если воспользоваться эвклидовой эвристикой для вычисления кратчайшего пути из **0** в **6** в сети из упражнения 21.1.

○ **21.72**. Опишите, что случится, если функция **distance(s, t)**, реализующая эвклидову эвристику, возвращает фактическую длину кратчайшего пути из *s* в *t* для всех пар вершин.

21.73. Разработайте реализацию класса для поиска кратчайших путей в насыщенных эвклидовых графах, которая базируется на представлении графа, поддерживающем

функцию **edge** и реализацию алгоритма Дейкстры (программа 20.6 с соответствующей функцией вычисления приоритета).

21.74. Выполните эмпирические исследования с целью проверки эффективности эвклидовой эвристики в задачах о кратчайших путях для различных эвклидовых сетей (см. упражнения 21.9, 21.68, 21.69 и 21.80). Для каждого графа сгенерируйте $V/10$ случайных пар вершин и распечатайте таблицу, которая показывает среднее расстояние между вершинами, среднюю длину кратчайшего пути между вершинами, среднее отношение количества вершин, просматриваемых с применением эвклидовой эвристики, к количеству вершин, просматриваемых алгоритмом Дейкстры, а также среднее отношение площади эллипса, соответствующего эвклидовой эвристике, к площади круга, соответствующего алгоритму Дейкстры.

21.75. Разработайте реализацию класса для задачи поиска кратчайших путей источник-сток на эвклидовых графах, которая основывается на двунаправленном поиске, описываемом в упражнении 21.35.

o **21.76**. Воспользуйтесь геометрической интерпретацией для получения оценки отношения количества вершин в SPT, создаваемых алгоритмом Дейкстры для задачи источник-сток, к количеству вершин в SPT, создаваемых двунаправленной версией, описанной в упражнении 21.75.

21.77. Разработайте реализацию класса для поиска кратчайших путей в эвклидовых графах, которая выполняет в конструкторе следующий шаг предварительной обработки: покрывает регион карты сеткой W на W, а затем при помощи алгоритма поиска кратчайших путей Флойда для всех пар вычисляет матрицу размером W^2 на W^2, в которой строка i и столбец j содержат длину кратчайшего пути, соединяющего любую вершину в квадрате сетки i с любой вершиной в квадрате сетки j. Используйте эти длины кратчайших путей в качестве нижних границ для усовершенствования эвклидовой эвристики. Проведите эксперименты для нескольких различных значений W так, чтобы ожидаемое количество вершин в квадрате решетки оказалось небольшим.

21.78. Разработайте реализацию АТД поиска кратчайших путей для всех пар для эвклидовых графов, которая бы объединила идеи, изложенные в упражнениях 21.75 и 21.77.

21.79. Выполните эмпирические исследования с целью сравнения эффективности эвристик, описанных в упражнениях 21.75–21.78, для различных эвклидовых сетей (см. упражнения 21.9, 21.68, 21.69 и 21.80).

21.80. Расширьте эмпирические исследования, включив в них эвклидовы графы, которые получаются в результате устранения всех вершин и ребер из круга радиуса r в центре, для $r = 0.1$, 0.2, 0.3 и 0.4. (Эти графы обеспечивают серьезную проверку эвклидовой эвристики.)

21.81. Постройте прямую реализацию алгоритма Флойда с АТД сети для неявных эвклидовых графов, определяемых N точками на плоскости с ребрами, которые соединяют точки внутри области диаметра d друг с другом. Не представляйте эти графы явным образом; вместо этого для заданных двух вершин вычислите расстояние между ними, чтобы определить, существует ли ребро, и если да, то чему равна его длина.

21.82. Разработайте реализацию для сценария, описанного в упражнении 21.81, когда строится граф с близкими связями, и затем примените алгоритм Дейкстры к каждой вершине (см. программу 21.1).

21.83. Выполните эмпирические исследования с целью сравнения времени выполнения и пространства памяти, требуемых для алгоритмов в упражнениях 21.81 и 21.82, для $d = 0.1$, 0.2, 0.3 и 0.4.

• **21.84.** Напишите клиентскую программу графической анимацией динамики эвклидовой эвристики. Ваша программа должна создать изображения, подобные рис. 21.19 (см. упражнение 21.38). Протестируйте программу на случайных эвклидовых сетях (см. упражнения 21.9, 21.68, 21.69 и 21.80).

21.6 Сведение

Вернемся к задаче о кратчайших путях, в частности, к общему случаю, когда допускаются отрицательные веса (тема раздела 21.7), – представим общую математическую модель, которую можно использовать для решения широкого круга различных задач, кажущихся не имеющими отношения к обработке графов. Эта модель является первой среди ряда таких общих моделей, с которыми предстоит встретиться чуть ниже. Поскольку мы подходим к более трудным задачам и все более общим моделям, одна из сложностей, с которой доведется столкнуться, связана с точной характеристикой взаимосвязей между различными задачами. Для каждой новой задачи мы задаемся вопросом, можем ли мы решить ее просто путем сведения к задаче с известным методом решения. Если у нас имеются ограничения на задачу, способны ли мы решить ее более простым путем? Чтобы ответить на вопросы подобного рода, в этом разделе мы ненадолго отклонимся от темы, дабы обсудить термины, используемые для описания таких видов взаимосвязи между задачами.

Определение 21.3. Мы говорим, что некоторая задача A **сводится к (reduces to)** другой задаче B, если есть возможность использовать алгоритм решения задачи B для разработки алгоритма, дающего решение задачи A, за суммарное время, которое в худшем случае не более чем в константное число раз превышает худшее время выполнения алгоритма решения задачи B. Мы говорим, что две задачи **эквивалентны**, если они сводятся одна к другой.

Отложим до части 8 строгое определение того, что означают слова "использовать" один алгоритм для "разработки" другого. Для большинства приложений мы довольствуемся следующим простым приемом. Мы показываем, что *A* сводится к *B*, демонстрируя возможность решения любого экземпляра задачи *A* за три шага:

- Преобразование ее к задаче *B*.
- Решение полученной задачи *B*.
- Преобразование решения задачи *B* в решение задачи *A*.

Поскольку мы можем выполнить преобразования (и решить задачу *B*) максимально эффективно, не менее эффективно можно решить также и задачу *A*. Дабы продемонстрировать такую методику доказательства, рассмотрим два примера.

Свойство 21.12. *Задача транзитивного замыкания сводится к задаче поиска кратчайших путей для всех пар с неотрицательными весами.*

Доказательство: Мы уже отмечали прямую связь между алгоритмами Уоршалла и Флойда. Другой способ исследования их взаимоотношений в настоящем контексте заклю-

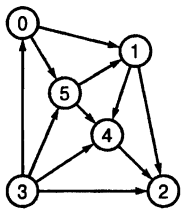
чается в представлении, что требуется вычислить транзитивное замыкание орграфов с использованием библиотечной функции поиска всех кратчайших путей в сетях. Для этого мы добавляем петли, если их нет в орграфе, а затем строим сеть непосредственно из матрицы смежности орграфа с произвольным весом (скажем, **0.1**), соответствующим каждой **1** и сигнальным весом, соответствующим каждому **0**. Затем мы обращаемся к функции поиска кратчайших путей для всех пар. Далее, из вычисленной функцией матрицы кратчайших путей для всех пар можно легко определить транзитивное замыкание: для любых двух заданных вершин *u* и *v* путь в орграфе из *u* в *v* существует тогда и только тогда, когда длина пути в сети из *u* в *v* будет ненулевой (см. рис. 21.21). ■

Это свойство является формальным утверждением о том, что задача транзитивного замыкания не является более трудной, чем задача кратчайших путей для всех пар. Ввиду существования алгоритмов транзитивного замыкания, которые являются даже более быстрыми, чем известные нам алгоритмы для задач поиска кратчайших путей для всех пар, это утверждение не должно вызывать удивления. Сведение представляет интерес тогда, когда оно используется для установки взаимосвязей между задачами, решение которых нам не известно, или между такими задачами и задачами, которые мы можем решить.

Свойство 21.13. *В сетях без ограничений на веса ребер задачи поиска наиболее длинного пути и кратчайшего пути (для единственного источника или для всех пар) эквивалентны.*

Доказательство: Для заданной задачи поиска кратчайшего пути проинвертируем все веса. Наиболее длинный путь (путь с наибольшим весом) в модифицированной сети есть кратчайший путь в исходной сети. Тот же самый аргумент показывает, что задача кратчайшего пути сводится к задаче наиболее длинного пути. ■

Это доказательство тривиально, но данное свойство также показывает, что осторожность при объявлении и доказательстве сведений оправдана, поскольку легко посчитать сведение само собой разумеющимся и таким образом ввести себя в заблуждение. Например, утверждение о том, что задачи о наиболее длинном и кратчайшем путях эквивалентны в сетях с неотрицательными весами, категорически не соответствует действительности.



	0	1	2	3	4	5
0	.10	.10				.10
1		.10	.10		.10	
2			.10			
3	.10		.10	.10	.10	.10
4			.10		.10	
5		.10			.10	.10

	0	1	2	3	4	5
0	.10	.10	.20		.20	.10
1		.10	.10		.10	
2			.10			
3	.10	.20	.10	.10	.10	.10
4			.10		.10	
5		.10	.20		.10	.10

	0	1	2	3	4	5
0	1	1	1	0	1	1
1	0	1	1	0	1	0
2	0	0	1	0	0	0
3	1	1	1	1	1	1
4	0	0	1	0	1	0
5	0	1	1	0	1	1

РИСУНОК 21.21. СВЕДЕНИЕ ТРАНЗИТИВНОГО ЗАМЫКАНИЯ

Для заданного орграфа (слева) мы можем преобразовать матрицу смежности (содержащую петли) в матрицу смежности, представляющую сеть, путем присваивания произвольного веса каждому ребру (матрица слева). Как обычно, пустые элементы матрицы представляют сигнальные значения, указывающие на отсутствие ребра. Для заданной матрицы длин кратчайших путей для всех пар этой сети (матрица в центре) транзитивное замыкание орграфа (матрица справа) суть просто матрица, образованная подстановкой **0** *для каждого сигнального значения и* **1** *для всех других элементов.*

В начале этой главы мы обрисовали аргументацию, показывающую, что задача нахождения кратчайших путей в неориентированных взвешенных графах сводится к задаче нахождения кратчайших путей в сетях, так что наши алгоритмы для сетей можем использовать для решения задач поиска кратчайших путей в неориентированных взвешенных графах. В этом контексте имеет смысл рассмотреть два дополнительных замечания, касающихся сведения. Во-первых, обратное утверждение не справедливо: знание того, как решить задачи кратчайших путей в неориентированных взвешенных графах, не помогает решать их в сетях. Во-вторых, мы видим дефект в аргументации: если бы веса ребер могли быть отрицательными, сведение привело бы к сети с отрицательными циклами, а мы не знаем, как искать в них кратчайшие пути. Даже если сведение завершается аварийно, все еще может оказаться возможным с помощью неопределенно сложного алгоритма отыскать кратчайшие пути в неориентированных взвешенных графах без отрицательных циклов (*см. раздел ссылок)*. Поскольку эта задача не сводится к направленной версии, данный алгоритм бесполезен при решении задачи поиска кратчайшего пути в сетях общего вида.

Концепция сведения, по существу, описывает процесс использования одного АТД для реализации другого, как это обычно делается современными системными программистами. Если две задачи эквивалентны, то известно, что если есть возможность эффективно решить какую-то из них, то можно эффективно решить и другую. Мы часто находим простые взаимно однозначные соответствия, такие как, например, в свойстве 21.13, где показана эквивалентность двух задач. В этом случае, даже до обсуждения способа решения каждой из задач, очень важен тот факт, что если удается отыскать эффективное решение одной задачи, то оно применимо и для решения другой. Еще один пример приводился в главе 17: когда мы встретились с задачей определения, имеет ли граф нечетный цикл, то обратили внимание, что данная задача эквивалентна определению, можно ли раскрасить этот граф двумя цветами.

Сведение имеет два основных применения в проектировании и анализе алгоритмов. Во-первых, оно помогает классифицировать задачи по их сложности на соответствующем абстрактном уровне без обязательной разработки и анализа полной реализации. Во-вторых, сведения часто делаются для установки нижних границ трудности решения различных задач, дабы получить понятие, когда прекращать поиск лучших алгоритмов. Мы видели примеры использования упомянутых приемов в разделах 19.3 и 20.7; позже в этом разделе будут рассматриваться и другие применения концепции сведения.

Помимо этой ориентации на практическое использование, принцип сведения также широко распространен в самой теории вычислений (со всеми вытекающими последствиями). Эти результаты важны потому, что позволяют понять, насколько возрастает трудность наших задач. Эта тема вкратце обсуждается в конце настоящего раздела, а вот подробно и совершенно формально — в части 8.

То, что стоимость преобразований не должна доминировать, является естественным фактором, который часто принимается во внимание. Во многих случаях, тем не менее, можно было бы принять решение использовать сведение даже при том, что стоимость преобразований оказывается действительно большой. Одно из наиболее важных применений концепции сведения заключается в преобразовании к понятной задаче, для которой известно эффективное решение, что обеспечивает эффективные решения таких задач, которые противном случае пришлось бы отнести к классу трудноразрешимых.

Сведение *A* к *B*, даже если выполнение этих преобразований обойдется намного дороже, чем собственно решение *B*, может дать намного более эффективный алгоритм решения, чем тот, который мы могли бы придумать другим путем. Может существовать множество различных вариантов. Возможно, в большей степени важен фактор средней ожидаемой стоимости, нежели фактор стоимости для худшего случая. Возможно, для решения *A* потребуется решить две задачи, *B* и *C*. Может быть, необходимо многократно решать задачу *B*. Мы оставляем дальнейшее обсуждение таких вариантов до части 8, поскольку все примеры, которые мы рассмотрим к тому времени, являются упрощенными вариантами только что упомянутых примеров.

В частности, когда мы решаем задачу *A*, упрощая другую задачу *B*, мы знаем, что *A* можно свести к *B*, но не обязательно наоборот. Например, выбор сводится к сортировке, поскольку есть возможность отыскать наименьший *k*-тый элемент в файле с помощью сортировки файла и затем перейти по индексу (или последовательно) в *k*-тую позицию; тем не менее, из этого факта, конечно же, не следует, что сортировка сводится лишь только к выбору. В данном контексте как задача поиска кратчайших путей для взвешенного DAG, так и задача поиска кратчайших путей для сетей с положительными весами сводятся к общей задаче вычисления кратчайших путей. Такое использование сведения соответствует интуитивному понятию некоторой более общей задачи. Любой алгоритм сортировки решает любую задачу выбора и, если можно решить задачу поиска кратчайших путей на сетях общего вида, то, разумеется, можно воспользоваться этим решением для сетей с различными ограничениями. Естественно, обратное утверждение не обязательно будет верным.

Подобное применение концепции сведения полезно, однако сама идея становится более полезной, если использовать ее для получения информации о взаимосвязях между задачами в различных областях. Например, рассмотрим следующие задачи, которые на первый взгляд кажутся далекими от обработки графов. При помощи сведения можно прояснить определенные взаимоотношения между этими задачами и задачей поиска кратчайших путей.

Календарное планирование. Пусть необходимо выполнить большой набор работ с разными продолжительностями. Мы можем выполнять любое количество работ одновременно, однако для каждой пары работ существует множество отношений предшествования, которые регламентируют, какая из работ должна быть завершена до того, как можно будет начать следующую работу. Каково минимальное время, которое необходимо для завершения всех работ, при условии выполнения всех ограничений предшествования? А именно, для данного набора работ (с указанием длительности каждой работы) и набора ограничений предшествования, требуется составить такой календарный план из выполнения (найти момент начала для каждой работы), чтобы достичь упомянутого выше минимума.

На рис. 21.22 приведен пример задачи календарного планирования. Здесь используется естественное представление в виде сети, которое в данном случае служит базой для сведения. Этот вариант задачи является, возможно, наиболее простым из буквально сотен вариантов, которые служили предметом исследований, т.е. вариантов, включающих другие характеристики работ и другие ограничения, подобные назначению работам персонала или других ресурсов, определению различных расходов, связанных с работами, конечных сроков выполнения и т.п. В этом контексте описанный выше вариант обычно

называется *календарным планированием с ограничением предшествования и произвольным параллелизмом*. Для краткости в дальнейшем будет применяться термин *календарное планирование*.

Для упрощения разработки алгоритма, решающего задачу планирования работ, мы рассмотрим следующую задачу, которая представляет интерес и сама по себе.

Разностные ограничения. Присвоить неотрицательные значения множеству переменных от x_0 до x_n с целью минимизации значения x_n при наличии множества *ограничений на разность* переменных, каждое из которых определяет, что разность между двумя этими переменными должна быть больше или равна заданной константе.

На рис. 21.23 показан пример задачи такого типа. Здесь представлена исключительно абстрактная математическая формулировка, которая может послужить основой для решения многих практических задач (*см. раздел ссылок*).

Задача разностных ограничений является частным случаем гораздо более общей задачи, где в уравнениях допускаются произвольные линейные комбинации переменных.

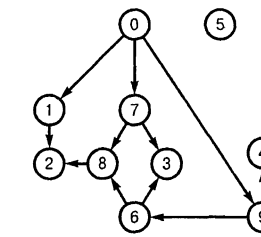


РИСУНОК 21.22. КАЛЕНДАРНОЕ ПЛАНИРОВАНИЕ

В этой сети вершины представляют работы, которые требуется выполнить (с весами, указывающими суммарное требуемое время), а ребра — отношения предшествования между работами. Например, ребра из **7** *в* **8** *и в* **3** *означают, что работа* **7** *должна быть завершена перед тем, как может быть начата работа* **8** *или работа* **3**. *Каково минимальное время, необходимое для завершения всех работ?*

Линейное программирование. Присвоить неотрицательные значения множеству переменных от x_0 до x_n с целью минимизации значения заданной линейной комбинации переменных при наличии множества ограничений на переменные, каждое из которых определяет, что заданная линейная комбинация переменных должна быть больше или равна заданной константе.

Линейное программирование является хорошо известным общим подходом к решению широкого класса задач оптимизации, поэтому мы не будем подробно его обсуждать, отложив это до части 8. Очевидно, что, как и многие другие задачи, задача разностных ограничений сводится к линейному программированию. В данный момент наши интересы сосредоточены на взаимосвязях между разностными ограничениями, календарным планированием работ и задачами поиска кратчайших путей.

Свойство 21.14. *Задача планирования работ сводится к задаче разностных ограничений.*

Доказательство: Для каждой работы добавим фиктивную работу и ограничения предшествования таким образом, чтобы данная работа должна была закончиться перед тем, как начнется фиктивная работа. Для этой задачи календарного планирования работ определим систему неравенств в конечных разностях, где каждой работе i соответствуют переменная x_i, и ограничение, согласно которому j не может начаться, пока не закончится i, в соответствии с неравенством $x_j \geq x_i + c_i$, где c_i есть продолжительность работы i. Решение задачи разностных ограничений дает точное решение задачи планирования работ, со значением каждой переменной, задающим время начала соответствующей работы. ■

Рисунок 21.23 иллюстрирует систему неравенств в конечных разностях, создаваемых этим сведением для задачи планирования работ из рис. 21.22. Практическое значение этого

сведения состоит в том, что для решения задачи планирования работ можно воспользоваться любым алгоритмом, позволяющим решить задачи разностных ограничений.

Полезно рассмотреть вопрос, можно ли провести аналогичное построение в обратном направлении: если задан алгоритм планирования работ, то можно ли воспользоваться им для решения задачи разностных ограничений? Ответ на этот вопрос состоит в следующем: доказательство свойства 21.14 не позволяет показать, что задача разностных ограничений сводится к задаче планирования работ, поскольку системы неравенств в конечных разностях, которые получаются из задачи календарного планирования, имеют свойства, которые не обязательно сохраняются для каждой задачи разностных ограничений. А именно, если два неравенства имеют одинаковые вторые переменные, то они имеют те же самые константы. Следовательно, алгоритм для календарного планирования работ непосредственно не дает прямого пути решения системы неравенств в конечных разностях, которая содержит два таких неравенства: $x_i - x_j \geq a$ и $x_k - x_j \geq b$, где $a \neq b$. Проверяя возможность сведения, следует иметь в виду ситуации, подобные следующей: при доказательстве возможности сведения A к B необходимо показать, что мы можем использовать алгоритм решения задачи B для решения *любого* варианта задачи A.

Конструктивно константы в задачах с разностными ограничениями, создаваемые построением при доказательстве свойства 21.14, всегда являются неотрицательными. Этот факт имеет существенное значение.

$$
\begin{aligned}
x_1 - x_0 &\geq .41 \\
x_7 - x_0 &\geq .41 \\
x_9 - x_0 &\geq .41 \\
x_2 - x_1 &\geq .51 \\
x_8 - x_6 &\geq .21 \\
x_3 - x_6 &\geq .21 \\
x_8 - x_7 &\geq .32 \\
x_3 - x_7 &\geq .32 \\
x_2 - x_8 &\geq .32 \\
x_4 - x_9 &\geq .29 \\
x_6 - x_9 &\geq .29 \\
x_{10} - x_2 &\geq .50 \\
x_{10} - x_3 &\geq .36 \\
x_{10} - x_4 &\geq .38 \\
x_{10} - x_5 &\geq .45
\end{aligned}
$$

РИСУНОК 21.23. РАЗНОСТНЫЕ ОГРАНИЧЕНИЯ

Нахождение неотрицательных значений, присваиваемых переменным, при которых с учетом данного множества неравенств минимизируется значение x_{10}, эквивалентно варианту задачи календарного планирования работ, который показан на рис. 21.22. Например, неравенство $x_8 \geq x_7 + 0.32$ означает, что работа **8** *не может начаться, пока выполняется работа* **7**.

Свойство 21.15. *Задача с разностными ограничениями при положительных константах эквивалентна задаче поиска наиболее длинных путей для единственного источника в ациклической сети.*

Доказательство: Для заданной системы уравнений в конечных разностях построим сеть, где каждая переменная x_i соответствует вершине i и каждое неравенство $x_i - x_j \geq c$ соответствует ребру i-j веса c. Например, назначение каждому ребру в орграфе на рис. 21.22 веса исходной вершины дает сеть, соответствующую набору неравенств в конечных разностях на рис. 21.23. Добавим фиктивную вершину к сети с ребрами нулевого веса, направленными в каждую из остальных вершин. Если сеть имеет цикл, то система уравнений в конечных разностях не имеет решения (поскольку из положительности весов следует, что значения переменных, соответствующих каждой вершине, строго уменьшаются по мере продвижения вдоль пути и, следовательно, цикл покажет, что некоторая переменная меньше себя самой). Иначе, если сеть не имеет цикла, мы решаем задачу поиска наиболее длинных путей для единственного источника в фиктивной вершине. Для каждой вершины существует наиболее длинный путь, поскольку сеть является ациклической (см. раздел 21.4). Присвоим каждой переменной длину наиболее длинного пути из фиктивной вершины в соответствующую

вершину в данной сети. Очевидно, что для каждой переменной значение этого пути удовлетворяет ограничениям, но никакое меньшее значение ограничениям не удовлетворяет.

В отличие от доказательства свойства 21.14, данное доказательство направлено на то, чтобы показать, что данные две задачи эквивалентны, поскольку построение работает в обоих направлениях. Мы не накладываем того ограничения, что два неравенства с одинаковыми вторыми переменными в неравенстве должны иметь одни и те же константы, и нет ограничений на то, чтобы ребра, исходящие из любой заданной вершины в сети, имели одни и те же веса. Для любой заданной ациклической сети с положительными весами такое же соответствие дает систему разностных ограничений с положительными константами, решение которой непосредственно приводит к решению задачи поиска наиболее длинных путей для единственного источника в сети. Детали этого доказательства оставляем на самостоятельную проработку (см. упражнение 21.90). ■

В сети на рис. 21.22 это соответствие показано для нашей типовой задачи, а на рис. 21.15 демонстрируется вычисление наиболее длинных путей в сети с использованием программы 21.6 (фиктивная начальная вершина скрыта в реализации). Календарный план, который вычисляется этим способом, приведен на рис. 21.24.

Программа 21.8 представляет собой реализацию, которая является примером применения рассмотренной теории к практической задаче. Она позволяет представить любой образец задачи календарного планирования работ как образец задачи поиска наиболее длинного пути в ациклических сетях, а затем использует программу 21.6 для ее решения.

```
0  0
1  .41
2  1.23
3  .91
4  .70
5  0
6  .70
7  .41
8  .91
9  .41
10 1.73
```

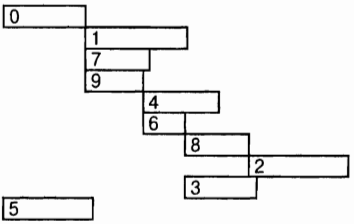


РИС. 21.24. КАЛЕНДАРНЫЙ ПЛАН

Этот рисунок иллюстрирует решение задачи планирования работ из рис. 21.22, полученное из соответствия между задачей о наиболее длинных путях во взвешенном DAG и задачей календарного планирования. Длины наиболее длинных путей в массиве ***wt****, которые вычисляются по алгоритму поиска наиболее длинных путей в программе 21.6 (см. рис. 21.15), есть в точности необходимые времена начала работ (правая колонка сверху). Мы начинаем работы* ***0*** *и* ***5*** *в момент* ***0****, работы* ***1****,* ***7*** *и* ***9*** *— в момент* ***0.41****, работы* ***4*** *и* ***6*** *— в момент* ***0.70*** *и т.д.*

Программа 21.8. Календарное планирование

Эта реализация читает из стандартного ввода список работ с их длительностями, за которым следует список ограничений предшествования, затем выводит на стандартный выход список начальных времен работ, которые удовлетворяют данным ограничениям. Используя свойства 21.14 и 21.15, а также программу 21.6, она решает задачу календарного планирования работ за счет сведения ее к задаче поиска наиболее длинных путей для ациклических сетей.

```
#include   "GRAPHbasic.cc"
#include   "GRAPHio.cc"
#include   "LPTdag.cc"
typedef  WeightedEdge  EDGE;
typedef  DenseGRAPH<EDGE>  GRAPH;
```

```
int main(int argc, char *argv[])
  { int i, s, t, N = atoi(argv[1]);
    double duration[N];
    GRAPH G(N, true);
    for (int i = 0; i < N; i++)
       cin >> duration[i];
    while (cin >> s >> t)
       G.insert(new EDGE(s, t, duration[s]));
    LPTdag<GRAPH, EDGE> lpt(G);
    for (i = 0; i < N; i++)
       cout << i << " " << lpt.dist(i) << endl;
}
```

Неявно мы предполагали, что решение существует для любой постановки задачи календарного планирования работ; тем не менее, если во множестве ограничений предшествования имеется цикл, то не существует способа составления расписания работ, удовлетворяющего таким ограничениям. Перед тем, как искать наиболее длинные пути, мы должны проверить выполнение этого условия путем проверки, имеет ли соответствующая сеть цикл (см. упражнение 21.100). Данная ситуация является типичной, и для ее описания обычно используется специальный технический термин.

Определение 21.4. *Говорят, что экземпляр задачи, который не допускает никакого решения, является* **невыполнимым**.

Другими словами, для задач планирования работ вопрос определения, является ли некоторый экземпляр задачи планирования выполнимым, сводится к задаче определения, является ли орграф ациклическим. Поскольку мы продвигаемся ко все более сложным задачам, вопрос разрешимости становится все более важной (и более трудной!) частью затрат вычислительных ресурсов.

К данному моменту мы рассмотрели три взаимосвязанных задачи. Можно было бы показать непосредственно, что задача планирования работ сводится к задаче поиска наиболее длинных путей для единственного источника в ациклических сетях, однако мы также показали, что подобным же образом можно решить любую задачу разностных ограничений (с положительными константами) (см. упражнение 21.94), равно как и любую другую задачу, которая сводится к задаче разностных ограничений или задаче планирования работ. С другой стороны, можно было бы разработать алгоритм для решения задачи разностных ограничений и воспользоваться им для решения других задач, но мы *не* показали, что решение задачи календарного планирования работ могло бы дать способ решения других задач.

Эти примеры иллюстрируют использование сведения, которое позволяет расширить область применимости проверенных реализаций. Действительно, при построении реализаций современное системное программирование делает упор на необходимости многократного использования программного обеспечения за счет разработки новых интерфейсов и использования ресурсов существующего программного обеспечения. Этот важный процесс, который иногда называется *библиотечным программированием*, является практическим воплощением концепции сведения.

Библиотечное программирование представляется чрезвычайно важным в практическом отношении, однако оно составляет только часть последствий применения сведения. Что-

бы иллюстрировать это замечание, рассмотрим следующую версию задачи календарного планирования работ.

Планирование работ с конечными сроками завершения. Допускает в задаче планирования работ ограничения дополнительного типа, которые определяют, что работа должна начаться *до* истечения заданного промежутка времени относительно другой работы. (Условные конечные сроки отсчитываются относительно начальной работы.) Такие ограничения обычно требуются в критичных ко времени производственных процессах и во множестве других приложений; разумеется, они могут существенно затруднить решение задачи календарного планирования работ.

Предположим, что нам нужно добавить ограничение к примеру, представленному на рис. 21.22–21.24, состоящее в том, что работа 2 должна начинаться не позже заданного количества временных единиц с после начала работы 4. Если с больше 0.53, то построенное расписание укладывается в заданные рамки, поскольку оно предписывает начать работу 2 в момент времени 1.23, что соответствует задержке на 0.53 после окончания работы 4 (которая начинается в 0.70). Если с меньше 0.53, то можно сдвинуть начало работы 4 на более позднее время, дабы удовлетворить это ограничение. Если работа 4 является достаточно длинной, то это изменение могло бы увеличить время завершения всего календарного графика. Хуже, если существуют другие ограничения на работу 4 и мы не можем переместить время ее начала. Действительно, могут встретиться ограничения, которые не может удовлетворить ни одно расписание: например, в нашем примере не удалось бы удовлетворить такие ограничения, при которых работа 2 должна начаться раньше, чем через d единиц времени после начала работы 6, для d меньшего 0.53, поскольку из ограничения, что за 2 должна следовать 8 и за 8 должна следовать 6, следует, что 2 должна начаться позже, чем через .53 единицы времени после начала 6.

Если добавить в пример оба ограничения, рассмотренные в предыдущем абзаце, то, в зависимости от значений *c* и *d*, каждое ограничение окажет влияние на время, когда 4 может быть поставлена в расписание, на время завершения всего графика, а также на то, существует ли выполнимый календарный график. Добавление дополнительных ограничений подобного типа увеличивает число возможностей и превращает легкую задачу в трудноразрешимую. Следовательно, при поиске подхода к решению задачи имеются все основания сводить ее к некоторой известной задаче.

Свойство 21.16. *Задача календарного планирования работ с конечными сроками завершения сводится к задаче поиска кратчайших путей (с возможностью существования отрицательных весов).*

Доказательство: Преобразуем ограничения предшествования к неравенствам, используя то же сведение, что и описанное в свойстве 21.14. Для каждого ограничения конечного срока добавим неравенство $x_i - x_j \leq d_j$, или, что эквивалентно, $x_j - x_i \geq -d_j$, где d_j есть положительная константа. Преобразуем набор неравенств в сеть, применив то же сведение, что и описанное в свойстве 21.15. Изменим все веса на отрицательные. С помощью того же построения, что и при доказательстве свойства 21.15, можно показать, что любое дерево кратчайшего пути в сети с корнем в 0 будет соответствовать расписанию. ■

Это сведение подводит нас к вопросу о кратчайших путях с отрицательными весами. Оно основано на том, что, если можно найти эффективное решение задачи поиска крат-

чайших путей с отрицательными весами, то можно отыскать и эффективное решение задачи планирования работ с конечными сроками завершения. (Опять-таки, соответствие в доказательстве свойства 21.16 *не* допускает обратное утверждение (см. упражнение 21.91).)

Добавление конечных сроков к задаче планирования работ соответствует допущению отрицательных констант в задаче с разностными ограничениями и отрицательным весам в задаче о кратчайших путях. (Это изменение также требует модификации задачи разностных ограничений, чтобы должным образом рассмотреть аналог отрицательных циклов в задаче о кратчайших путях.) Эти более общие варианты задач более трудноразрешимы, чем те варианты, которые мы рассматривали вначале, но, по всей видимости, они также и более полезны как обобщенные модели. Приемлемое приближение к решению всех их, скорее всего, следует искать в эффективном решении задачи поиска кратчайших путей с отрицательными весами.

К сожалению, с этим подходом связана принципиальная трудность, и она иллюстрирует другую сторону вопроса использования сведения, позволяющую оценить относительную трудность задач. Мы уже задействовали положительные свойства сведения для расширения применимости решений общих задач, однако оно обладает также и отрицательными сторонами, показывающими ограниченность возможности такого расширения.

Эта трудность заключается в том, что решение общей задачи поиска кратчайших путей слишком сложно. Далее мы увидим, как с помощью концепции сведения сформулировать данное утверждение точно и обоснованно. В разделе 17.8 было рассмотрено множество задач, известных как NP-трудные, которые мы посчитали неразрешимыми, поскольку все известные алгоритмы для их решения в худшем случае требуют экспоненциального времени. Сейчас будет показано, что в общем случае задача поиска кратчайших путей является NP-трудной.

Как кратко упоминалось в разделе 17.8 и подробно обсуждается в части 8, в общем случае, когда мы утверждаем, что задача является NP-трудной, это означает не только то, что не известно эффективного алгоритма, гарантирующего решение этой задачи, но также и то, что все-таки остается некоторая надежда отыскать таковой. В этом контексте мы используем термин *эффективный,* ссылаясь на алгоритмы, время счета которых ограничено в худшем случае некоторой полиномиальной функцией значения входной переменной. Мы предполагаем, что открытие эффективного алгоритма решения любой NP-трудной задачи должно быть ошеломляющим научным достижением. Концепция NP-трудности является важной для трудноразрешимых задач идентификации, поскольку часто легко доказать, что задача NP-трудна, используя следующую методику.

Свойство 21.17. *Задача является NP-трудной, если существует любая NP-трудная задача, эффективно сводящаяся к ней.*

Это свойство зависит от строгого определения *эффективного сведения* одной задачи A к другой задаче B. Мы откладываем такие определения до части 8 (обычно применяются два различных определения). В данный момент мы просто используем этот термин, чтобы описать случай, когда существуют эффективные алгоритмы как для преобразования образца A к образцу B, так и для преобразования решения B к решению A.

Теперь предположим, что имеется эффективное сведение NP-трудной задачи A к конкретной задаче B. Доказательство проведем от противного: если мы имеем эффективный алгоритм для B, то мы могли бы воспользоваться им для решения любого экземпляра A за полиномиальное время с помощью сведения (преобразуем данный экземпляр A к экземпляру B, решаем эту задачу, затем преобразуем полученное решение). Однако не существует известного алгоритма, который гарантировал бы такое решение для A (поскольку A является NP-трудной), так что допущение о существовании алгоритма с полиномиальным временем для задачи B некорректно – B также является NP-трудной задачей. ■

Продемонстрированная методика является чрезвычайно важной, поскольку она использовалась для того, чтобы показать, что огромное множество задач относятся к NP-трудным, предоставляя на выбор широкий круг задач, с помощью которых строились доказательства NP-трудности новой задачи. Например, в разделе 17.7 мы столкнулись с одной из классических NP-трудных задач – с задачей поиска гамильтонова пути, в которой требуется определить, существует ли простой путь, содержащий все вершины данного графа. Она была одной из первых задач, в отношении которой было показано, что она является NP-трудной (*см. раздел ссылок*). Довольно просто получить формулировку, в соответствии с которой из свойства 21.17 следует, что задача кратчайших путей относится к NP-трудным:

Свойство 21.18. *В сетях, в которых веса ребер могут быть отрицательными, задачи поиска кратчайших путей являются NP-трудными.*

Доказательство: Наше доказательство состоит в сведении задачи о гамильтоновом пути к задаче о кратчайших путях. Другими словами, мы покажем, что для решения задачи о гамильтоновом пути можно использовать любой алгоритм, с помощью которого можно найти кратчайшие пути в сетях с отрицательными весами ребер. Для заданного неориентированного графа мы строим сеть с ребрами в обоих направлениях, соответствующими каждому ребру в графе, и со всеми ребрами, имеющими веса -1. Кратчайший (простой) путь, начинающийся в любой вершине этой сети, имеет длину $1 - V$ тогда и только тогда, когда граф имеет гамильтонов путь. Обратите внимание, что эта сеть изобилует отрицательными циклами. Не только каждый цикл в графе соответствуют отрицательному циклу в сети, но также каждое *ребро* в графе соответствует циклу с весом -2 в данной сети.

Результат этих рассуждений состоит в том, что задача поиска кратчайших путей является NP-трудной, поскольку, если бы мы могли разработать эффективный алгоритм для задачи поиска кратчайших путей в сетях, то мы должны были бы иметь эффективный алгоритм для задачи поиска гамильтонова пути в графах. ■

Как только будет обнаружено, что данная задача является NP-трудной, естественным будет разыскать те варианты этой задачи, которые *можно* решить. Для задач поиска кратчайших путей мы стоим перед выбором между наличием массы эффективных алгоритмов для ациклических сетей или для сетей, в которых веса ребер не являются отрицательными, и отсутствием хорошего решения для сетей, которые могли бы содержать циклы и отрицательные веса. Есть ли какие-либо другие виды сетей, которые нас могут заинтересовать? Это является темой раздела 21.7. Там, например, будет показано, что задача календарного планирования работ с конечными сроками сводится к варианту задачи по-

иска кратчайших путей, которую можно эффективно решить. Типична следующая ситуация: поскольку нас интересуют наиболее трудные вычислительные задачи, мы направляем усилия на определение вариантов тех задач, которые мы можем надеяться решить.

Как показывают эти примеры, сведение — это простая технология, которая полезна при разработке алгоритмов, поэтому мы часто прибегаем к нему. При этом мы либо сможем решить новую задачу путем доказательства того, что она сводится к задаче с известным решением, либо мы сможем доказать, что новая задача будет сложной, показав, что задача, о которой известно, что она сложная, сводится к рассматриваемой задаче.

В таблице 21.3 дан более подробный обзор различных последствий применения сведения между четырьмя общими классами задач, которые обсуждались в главе 17. Обратите внимание, что существует несколько случаев, когда сведение не дает новой информации; например, хотя выбор и сводится к сортировке, а задача нахождения наиболее длинных путей в ациклических сетях сводится к задаче нахождения кратчайших путей в общих сетях, эти утверждения не проливают свет на относительную трудность задачи. В других случаях сведение либо может, либо нет, предоставлять новую информацию; еще в некоторых случаях последствия сведения действительно глубоки. Чтобы продолжить развитие этих концепций, мы нуждаемся в строгом и формальном описании концепции сведения, о чем мы подробно поговорим в части 8; здесь мы просто неформально подведем итог наиболее важных практических применений сведения вместе с примерами, которые уже были представлены.

Таблица 21.3. Последствия сведения

Эта таблица подводит итог некоторых последствий сведения задачи *A* к другой задаче *B*, с примерами, которые обсуждались в этом разделе. Глубокие последствия случаев 9 и 10 заходят настолько далеко, что мы в общем случае предполагаем, что нет возможности обосновать такие сведения (см. часть 8). Сведение наиболее полезно в следующих случаях: 1, 6, 11 и 16 – для изучения нового алгоритма для *A* или обоснования нижней границы для *B*; 13 – 15 – для изучения новых алгоритмов для *A*; 12 – для того, чтобы убедиться в сложности *B*.

	A	B	Результаты сведения $A \Rightarrow B$	Пример
1	Простая	Простая	Новая нижняя граница *B*	сортировка ⇒ EMST
2	Простая	Разрешимая	Нет	TC ⇒ APSP(+)
3	Простая	Неразрешимая	Нет	SSSP(DAG) ⇒ SSSP(±)
4	Простая	Решение не известно	Нет	
5	Разрешимая	Простая	*A* простая	
6	Разрешимая	Разрешимая	Новое решение *A*	DC(+) ⇒ SSSP(DAG)
7	Разрешимая	Неразрешимая	Нет	
8	Разрешима	Решение не известно	Нет	
9	Неразрешимая	Простая	Глубокие	
10	Неразрешимая	Разрешимая	Глубокие	
11	Неразрешимая	Неразрешимая	Так же, как 1 или 6	SSLP(±) ⇒ SSSP(±)

12	Неразрешимая	Решение не известно	*B* неразрешимая	HP ⇒ SSSP(±)
13	Решение не известно	Простая	*A* простая	JS ⇒ SSSP(DAG)
14	Решение не известно	Разрешимая	*A* разрешимая	
15	Решение не известно	Неразрешимая	*A* разрешимая	
16	Решение не известно	Решение не известно	Так же, как 1 или 6	JSWD ⇒ SSSP(±)

Обозначения:

EMST	минимальное эвклидово остовное дерево
TC	транзитивное замыкание
APSP	кратчайшие пути для всех пар
SSSP	кратчайшие пути для единственного источника
SSLP	наиболее длинные пути для единственного источника
(+)	(в сетях с неотрицательными весами)
(±)	(в сетях с весами, которые могут быть неотрицательными)
(DAG)	(в ациклических сетях)
DC	разностные ограничения
HP	гамильтоновы пути
JS(WD)	планирование заданий (с конечными сроками)
⇒	применение сведения

Верхние границы. Если мы имеем эффективный алгоритм решения задачи *B* и можем доказать, что *A* сводится к *B*, то мы имеем эффективный алгоритм и для решения задачи *A*. Возможно, что существует другой лучший алгоритм для *A*, однако эффективность *B* является верхней границей того, что можно достичь для *A*. Например, наше доказательство того, что задача планирования работ сводится к задаче поиска наиболее длинных путей в ациклических сетях, превращает наш алгоритм решения второй задачи в эффективный алгоритм решения первой задачи.

Нижние границы. Если известно, что любой алгоритм для задачи *A* требует определенных ресурсов, и можно доказать, что *A* сводится к *B*, то мы знаем, что *B* имеет, по крайней мере, те же самые требования к ресурсам, поскольку из существования лучшего алгоритма для *B* следовало бы существование лучшего алгоритма для *A* (до тех пор пока стоимость сведения будет ниже стоимости алгоритма *B*). Другими словами, эффективность *A* в лучшем случае достигает нижней границы того, что можно получить для *B*. Например, эта технология применялась в разделе 19.3 для того, чтобы показать, что вычисление транзитивного замыкания является таким же трудоемким, как умножение логических матриц, и в разделе 20.7 – чтобы показать, что вычисление эвклидового MST столь же трудоемко, как сортировка.

Неразрешимость. В частности, можно доказать, что некоторая задача неразрешима, показав, что к ней сводится другая неразрешимая задача. Например, свойство 21.18 говорит о том, что задача поиска кратчайших путей неразрешима, поскольку к ней сводится задача поиска гамильтонова пути, которая неразрешима.

Помимо этих общих результатов, ясно, что более детальная информация об эффективности определенных алгоритмов для решения специфических задач может непосредствен-

но быть применена к другим задачам, которые сводятся к первым. Когда мы ищем верхнюю границу, мы можем анализировать алгоритм, связанный с данным, проводить эмпирические исследования и т.п., дабы определить, получено ли лучшее решение задачи. Во время разработки качественного универсального алгоритма можно затратить усилия на разработку и тестирование качественной реализации и затем создать соответствующий АТД, что позволит расширить применимость.

Мы используем сведение в качестве базового инструментального средства в этой и следующей главах. Акцент делается на общей применимости рассматриваемых задач и общей применимости решающих их алгоритмов – за счет сведения к ним других задач. Важно также иметь представление о взаимоотношениях между моделями в иерархической структуре их проблемных формулировок. Например, линейное программирование является общей формулировкой, которая важна не только потому, что к ней сводятся многие задачи, но также и из-за того, что известно, что эта задача не относится к NP-трудным. Другими словами, не известно способа сведения общей задачи поиска кратчайших путей (или любой другой NP-трудной задачи) к линейному программированию. Результаты подобного рода подробно обсуждаются в части 8.

Не все задачи являются разрешимыми, но к настоящему времени уже наработаны хорошие общие модели, подходящие для широкого класса задач, для которых известны методы решения. Кратчайшие пути в сетях — это наш первый пример такой модели. По мере продвижения ко все более общим проблемным областям, мы попадаем в область *исследования операций (OR, operations research)*, которая занимается анализом математических методов принятия решений и главной целью которой является развитие и изучение таких моделей. Одна ключевая проблема при исследовании операций заключается в нахождении модели, которая в наибольшей степени подходит для решения задачи и которая умещает эту задачу в данную модель. Эта область исследований известна также как *математическое программирование* (название, данное до наступления эры компьютеров, т.е. перед новой трактовкой слова "программирование"). Сведение – это современная концепция, которая представляет по сути то же, что и математическое программирование, и является основой нашего понимания стоимости вычислений для широкого круга приложений.

Упражнения

▷ **21.85**. При помощи сведения, описанного в свойстве 21.12, разработайте реализацию транзитивного замыкания (с тем же интерфейсом, что и в программах 19.3 и 19.4), которая использует АТД поиска кратчайших путей для всех пар из раздела 21.3.

21.86. Покажите, что задача вычисления числа сильных компонентов в орграфе сводится к задаче поиска кратчайших путей для всех пар с неотрицательными весами.

21.87. Сформулируйте задачи разностных ограничений и кратчайших путей, которые соответствуют (по примеру построения свойств 21.14 и 21.15) задаче планирования работ, где работы от 0 до 7 имеют, соответственно, длины:

0.4 0.2 0.3 0.4 0.2 0.5 0.1

и ограничения:

5-1 4-6 6-0 3-2 6-1 6-2

▷ **21.88**. Дайте решение задачи планирования работ из упражнения 21.87.

○ **21.89**. Предположите, что работы в упражнении 21.87 имеют также ограничения, состоящие в том, что работа 1 должна начаться перед окончанием работы 6, а работа 2 должна начаться перед окончанием работы 4. Сформулируйте задачу поиска кратчайших путей, к которой сводится эта задача, воспользовавшись рассуждениями, приведенными при доказательстве свойства 21.16.

21.90. Покажите, что задача поиска наиболее длинных путей для всех пар в ациклических сетях с положительными весами сводится к задаче разностных ограничений с положительными константами.

▷ **21.91**. Объясните, почему методика доказательства свойства 21.16 не распространяется на доказательство того, что задача кратчайших путей сводится к задаче планирования работ с конечными сроками.

21.92. Модернизируйте программу 21.8, чтобы на работы можно было ссылаться с использованием символических имен вместо целых чисел (см. программу 17.10).

21.93. Разработайте интерфейс АТД, который дает возможность клиентам ставить и решать задачи разностных ограничений.

21.94. Разработайте класс, который реализует интерфейс из упражнения 21.93, основывая решение задачи разностных ограничений на сведении к задаче поиска кратчайших путей в ациклических сетях.

21.95. Предоставьте реализацию класса, который решает задачу поиска кратчайших путей для единственного источника в ациклических сетях с отрицательными весами, основанную на сведении к задаче разностных ограничений и использующую интерфейс из упражнения 21.93.

○ **21.96**. Ваше решение задачи кратчайших путей в ациклических сетях из упражнения 21.95 предполагает существование реализации, которая решает задачу разностных ограничений. Что случится, если вы воспользуетесь реализацией из упражнения 21.94, которая предполагает существование реализации для задачи кратчайших путей в ациклических сетях?

○ **21.97**. Докажите эквивалентность любых двух NP-трудных задач (т.е., выберите две задачи и докажите, что они сводятся друг к другу).

●● **21.98**. Сформулируйте явные рассуждения, которые сводят задачу поиска кратчайших путей в сетях с целочисленными весами к задаче поиска гамильтонова пути.

● **21.99**. При помощи сведения разработайте класс, использующий АТД сети и решающий задачу поиска кратчайших путей для единственного источника, чтобы решить следующую задачу: для заданного орграфа, вектора положительных весов, индексированного вершинами, и начальной вершины v найти такие пути из v в каждую другую вершину, чтобы сумма весов вершин в пути была минимальной.

○ **21.100**. Программа 21.8 не проверяет, выполнима ли задача планирования работ, которую она выбирает в качестве входной (например, может существовать цикл). Охарактеризуйте календарные графики, которые программа будет выводить для трудноразрешимых задач.

21.101. Разработайте интерфейс АТД, который дает клиентам возможность ставить и решать задачи календарного планирования работ. Напишите класс, который реализует полученный интерфейс, основав решение задачи календарного планирования работ на сведении к задаче поиска кратчайших путей в ациклических сетях (как в программе 21.8).

○ **21.102**. Добавьте в класс из упражнения 21.101 функцию (вместе с реализацией), которая печатает наиболее длинный путь в календарном графике. (Такой путь обычно называется *критическим путем*.)

21.103. Напишите клиент для интерфейса из упражнения 21.101, который осуществляет вывод в PostScript-программу для изображения календарного графика в стиле рис. 21.24 (см. раздел 4.3).

○ **21.104**. Разработайте модель для генерации задач календарного планирования работ. Воспользуйтесь полученной моделью для тестирования результатов выполнения упражнений 21.101 и 21.103 на приемлемом множестве размерностей задач.

21.105. Напишите класс, который реализует интерфейс из упражнения 21.101, основав решение задачи календарного планирования работ на сведении к задаче разностных ограничений.

○ **21.106**. Диаграмма PERT (Project Evaluation and Review Technique – сетевое планирование и управление) – это сеть, которая представляет задачу календарного планирования работ, с *ребрами* в качестве работ (см. рис. 21.25). Напишите класс, реализующий интерфейс календарного планирования работ из упражнения 21.101, который основывается на диаграммах PERT.

21.107. Сколько вершин будет в диаграмме PERT для задачи планирования работ с V работами и E ограничениями?

21.108. Напишите программы для преобразования между представлением календарного планирования работ, основанном на ребрах (диаграммы PERT) (см. упражнение 21.106), и представлением, основанном на вершинах (см. рис. 21.22).

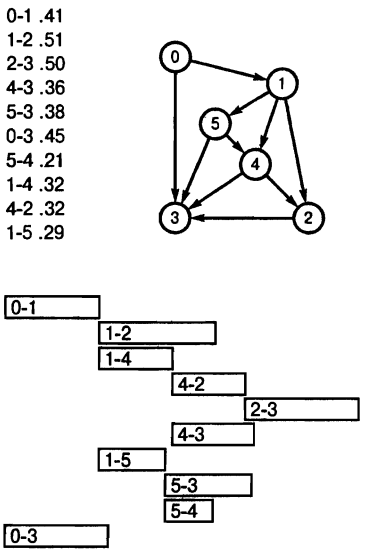


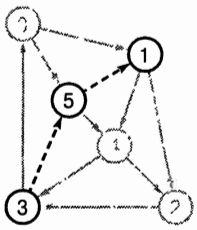
РИСУНОК 21.25. ДИАГРАММА PERT

Диаграмма PERT — это сетевое представление задачи календарного планирования работ, где работы изображаются ребрами. Сеть в верхней части рисунка есть представление задачи планирования работ из рис. 21.22, при этом работы от **0** *до* **9** *представлены, соответственно, ребрами* **0-1**, **1-2**, **2-3**, **4-3**, **5-3**, **0-3**, **5-4**, **1-4**, **4-2** *и* **1-5**. *Критическим путем в этом календарном графике является наиболее длинный путь в данной сети.*

21.7 Отрицательные веса

Теперь обратимся к теме, связанной с отрицательными весами в задачах поиска кратчайших путей. Возможно, отрицательные веса ребер покажутся неестественными, поскольку на протяжении большей части этой главы мы на интуитивных примерах привыкли, что веса представляют расстояния или стоимости. Однако, как было показано в разделе 21.6, отрицательные веса ребер возникают естественным путем, когда к задаче поиска кратчайших путей сводятся другие задачи. Отрицательные веса являются не только математической абстракцией; напротив, они существенно расширяют применимость задач кратчайших путей в качестве модели для решения других задач. Эта потенциальная польза побуждает нас искать эффективные алгоритмы для решения сетевых задач, допускающих наличие отрицательных весов.

0-1 .41
1-2 .51
2-3 .50
4-3 .36
3-5 -.38
3-0 .45
0-5 .29
5-4 .21
1-4 .32
4-2 .32
5-1 -.29



	0	1	2	3	4	5
0	0	0	.51	.68	.32	.29
1	1.13	0	.51	.68	.32	.30
2	.95	-.17	0	.50	.15	.12
3	.45	-.67	-.16	0	-.35	-.38
4	.81	-.31	.20	.36	0	-.02
5	.84	-.29	.22	.39	.03	0

	0	1	2	3	4	5
0	0	5	5	5	5	5
1	4	1	2	4	4	4
2	3	3	2	3	3	3
3	0	5	5	3	5	5
4	3	3	3	3	4	3
5	1	1	1	1	1	5

РИСУНОК 21.26. ТИПОВАЯ СЕТЬ С ОТРИЦАТЕЛЬНЫМИ РЕБРАМИ

Это та же типовая сеть, что и сеть, показанная на рис. 21.1, за исключением того, что ребра **3-5** *и* **5-1** *являются отрицательными. Естественно, данное изменение существенным образом воздействует на структуру кратчайших путей, что легко заметить путем сравнения матриц расстояний и путей справа с соответствующими матрицами на рис. 21.9. Например, кратчайший путь из* **0** *в* **1** *в этой сети есть* **0-5-1**, *и он имеет длину* **0**, *тогда как кратчайший путь из* **2** *в* **1** *есть* **2-3-5-1** *с длиной* **-0.17**.

Рисунок 21.26 представляет небольшой пример, который иллюстрирует влияние введения отрицательных весов в задаче о наиболее коротких путях в сети. Возможно, при этом наиболее важный эффект состоит в том, что когда присутствуют отрицательные веса, то *наиболее короткие пути с малыми весами имеют тенденцию содержать больше ребер, чем пути с более высокими весами.* В случае положительных весов нашей целью было искать *кратчайшие расстояния;* однако, при наличии отрицательных весов мы ищем *обходные пути (detours),* которые используют столько ребер с отрицательными весами, сколько мы сможем отыскать. Этот эффект производит переворот в нашем интуитивном понимании поиска алгоритмов "коротких" путей, так что нам потребуется преодолеть этот барьер интуиции и рассмотреть данную задачу на базовом абстрактном уровне.

Отношение между кратчайшими путями в сетях и гамильтоновыми путями в графах, показанное при доказательстве свойства 21.18, связано с нашим наблюдением, что поиск путей низкого веса (которые мы называем "короткими") равноценен поиску путей с большим количеством ребер (которые мы могли бы считать "длинными"). При наличии отрицательных весов мы ищем скорее длинные пути, нежели короткие.

Первая мысль, которая приходит на ум, дабы исправить ситуацию, – это найти наименьший (наиболее отрицательный) вес ребра, затем добавить абсолютное значение этого числа ко всем весам ребер, чтобы превратить сеть в такую, которая не имеет отрицательных весов. Это наивное приближение вообще не работает из-за того, что кратчайшие пути в новой сети не имеют никакого отношения к кратчайшим путям в прежних сетях. Например, в сети приведенной на рис. 21.26, кратчайший путь из 4 в 2 есть 4-3-5-1-2. Если увеличить на 0.38 веса всех ребер в графе, чтобы сделать их все положительными, вес этого пути возрастет от 0.20 до 1.74. Однако вес 4-2 возрастет в действительности с 0.32 до 0.70, так что это ребро станет кратчайшим путем из 4 в 2. Чем больше ребер имеет путь, тем больше он страдает от такого преобразования, поэтому результат, по сравнению с тем, что мы наблюдали в предыдущем параграфе, как раз противоположен тому, что требуется. Даже если эта наивная идея не работает, цель преобразования сети в эквивалентную, без отрицательных весов, но с теми же кратчайшими путями, вполне достойна внимания; в конце раздела мы рассмотрим алгоритм достижения упомянутой цели.

Все наши алгоритмы поиска кратчайших путей до сих пор содержали одно из двух ограничений на задачу кратчайших путей, так чтобы с их помощью можно было получить эффективное решение: они либо запрещали циклы, либо запрещали отрицательные веса. Существуют ли менее строгие ограничения, которые можно было бы наложить на сети, содержащие как циклы, так и отрицательные веса, при которых мы все еще получали бы разрешимые задачи кратчайших путей? Мы затронули ответ на этот вопрос в начале главы, когда нам требовалось добавить ограничение, состоящее в том, что пути должны быть настолько просты, чтобы задача имела бы смысл в случае наличия отрицательных циклов. Возможно, нам следует ограничить внимание к сетям, которые не имеют таких циклов?

Кратчайшие пути в сетях без отрицательных циклов. Для заданной сети, которая может содержать ребра с отрицательными весами, но не содержит циклов отрицательного веса, решим одну из следующих задач: найти кратчайший путь, соединяющий две заданных вершины (задача поиска кратчайшего пути), найти кратчайшие пути из заданной вершины во все другие вершины (задача с единственным источником), или найти кратчайшие пути, соединяющие все пары вершин (задача всех пар).

Доказательство свойства 21.18 оставляет двери открытыми для эффективных алгоритмов решения этой задачи, поскольку оно терпит неудачу, если запретить отрицательные циклы. Чтобы решить задачу поиска гамильтонова пути, необходимо иметь возможность решать задачи поиска кратчайших путей в сетях, которые содержат огромные количества отрицательных циклов.

Кроме того, много практических задач сводятся в точности к задаче нахождения кратчайших путей в сетях, которые не содержат отрицательных циклов. Мы уже видели один такой пример.

Свойство 21.19. *Задача календарного планирования работ с конечными сроками сводится к задаче поиска кратчайших путей в сетях, которые не содержат отрицательных циклов.*

Доказательство: Аргументация, которую мы использовали при доказательстве свойства 21.15, показала, что построение в доказательстве свойства 21.16 ведет к сетям, не содержащим отрицательных циклов. Из задачи календарного планирования работ мы создаем задачу разностных ограничений с переменными, которые соответствуют временам начала работ, а из задачи разностных ограничений мы создаем сеть. Мы делаем отрицательными все веса, чтобы перейти от задачи поиска наиболее длинных путей к задаче поиска кратчайших путей – преобразование, которое соответствует изменению знаков всех неравенств. Любой простой путь в сети из i в j соответствует последовательности неравенств, включающих переменные. Из существования такого пути, за счет сокращения этих неравенств, получается, что $x_i - x_j \leq w_{ij}$, где w_{ij} есть сумма весов на пути из i в j. Отрицательный цикл соответствует 0 в левой части этого неравенства и отрицательному значению в правой части, так что существование такого цикла является противоречием. ■

Как мы отмечали, когда впервые обсуждали задачу календарного планирования работ в разделе 21.6, это утверждение безоговорочно предполагает, что наши задачи календарного планирования работ являются выполнимыми (т.е. имеют решение). На практике же мы не должны делать подобного допущения, и часть вычислительных затрат должна быть отведена на определение выполнимости задачи календарного планирования с конечны-

ми сроками. В построении доказательства свойства 21.19 отрицательный цикл в сети предполагает, что задача невыполнима, так что данная задача соответствует следующей.

Обнаружение отрицательных циклов. Содержит ли данная сеть отрицательный цикл? Если это так, то найти один такой цикл.

С одной стороны, эта задача не обязательно простая (простые алгоритмы проверки циклов для орграфов не применяются); с другой стороны, она не обязательно сложная (сведение из свойства 21.16 в отношении задачи поиска гамильтонова пути не применяется). Первое, что мы предпримем, так это разработаем алгоритм решения этой задачи.

В приложениях задач календарного планирования с конечными сроками отрицательные циклы соответствуют состояниям ошибки, которые по-видимому редки, но наличие которых мы должны проверять. Мы могли бы даже разработать алгоритмы, которые удаляют ребра, чтобы разорвать отрицательный цикл, и повторяются до тех пор, пока существует хотя бы один такой цикл. В других приложениях обнаружение отрицательных циклов является основной целью, как в следующем примере.

Арбитражные операции (скупка и продажа ценных бумаг с целью получения выгоды). Многие газеты печатают таблицы курсов обращения мировых валют (см., например, рис. 21.27). Мы можем рассматривать такие таблицы как матрицы смежности, представляющие полные сети. Ребро *s-t* с весом *x* означает, что мы можем конвертировать *одну* единицу валюты *s* в *x* единиц валюты *t*. Пути в сети задают многошаговые конверсии. Например, если существует также ребро *t-w* с весом *y*, то путь *s-t-w* представляет способ, позволяющий конвертировать *одну* единицу валюты *s* в *xy* единиц валюты *w*. Можно было бы ожидать, что *xy* будет равен весу *s-w* во всех случаях, однако рассматриваемые таблицы являются сложной динамичной системой, где такая последовательность не может быть гарантирована. Если мы находим случай, где *xy* меньше, чем вес *s-w*, то мы в состоянии перехитрить систему. Предположим, что вес *w-s* есть *z* и $xyz > 1$, тогда цикл *s-t-w-s* дает способ конвертировать *одну* единицу валюты *s* в более чем *одну* единицу (*xyz*) валюты *s*. То есть, мы можем получить доход в $100(xyz - 1)$ процентов за счет конверсии *s* в *t*, затем в *w* и обратно в *s*. Данная ситуация является ярким примером *арбит-*

	$	P	Y	C	S
$	1.0	1.631	0.669	0.008	0.686
P	0.613	1.0	0.411	0.005	0.421
Y	1.495	2.436	1.0	0.012	1.027
C	120.5	197.4	80.82	1.0	82.91
S	1.459	2.376	0.973	0.012	1.0

	$	P	Y	C	S
$	0.0	0.489	-0.402	-4.791	-0.378
P	-0.489	0.0	-0.891	-5.278	-0.865
Y	0.402	0.89	0.0	-4.391	0.027
C	4.791	5.285	4.392	0.0	4.418
S	0.378	0.865	-0.027	-4.415	0.0

РИС. 21.27. АРБИТРАЖНЫЕ ОПЕРАЦИИ

Таблица в верхней части задает переводные коэффициенты одной валюты в другую. Например, второй элемент в верхнем ряду говорит, что за $1 можно купить 1.631 единиц валюты P. Конверсия $1000 в валюту P и обратно должна дать $1000(1.631)*(0.613) = $999, т.е. потеря составляет $1. Однако конверсия $1000 в валюту P, затем в валюту Y и возврат обратно приносит прибыль $1000*(1.631)*(0.411)*(1.495) = $1002, т.е. 0.2%. Если сделать отрицательным логарифм всех чисел в таблице (внизу), можно считать, что получена матрица смежности для полной сети с весами ребер, которые могут быть как положительными, так и отрицательными. В этой сети узлы соответствуют валютам, ребра — конверсиям, а пути — последовательностям конверсий. Только что описанная конверсия соответствует циклу $-P-Y-$ в графе, причем вес это цикла составляет -0.489 + 0.890 − 0.402 = -0.002. Лучшая возможность для совершения арбитражных операций соответствует наиболее короткому циклу в графе.*

ражных операций (arbitrage), которые позволили бы нам получать безграничные доходы, если бы не существовало сил, действующих вне пределов модели, таких, например, как ограничения на размер сделок. Чтобы преобразовать эту задачу в задачу кратчайших путей, мы логарифмируем все числа так, чтобы веса путей соответствовали сумме весов ребер вместо их умножения, а затем мы делаем их отрицательными, дабы обратить сравнение. Тогда веса ребер могли бы оказаться отрицательными или положительными, а кратчайший путь из *s* в *t* дал бы лучший способ конверсии валюты *s* в валюту *t*. Цикл с наименьшим весом указал бы лучшую возможность для арбитражных операций, однако интерес представляет любой отрицательный цикл.

Можно ли обнаружить отрицательные циклы в сети или найти кратчайшие пути в сетях, которые не содержат отрицательных циклов? Существование эффективных алгоритмов для решения этих задач не противоречит NP-трудности общей задачи, которая была доказана в свойстве 21.18, поскольку сведение задачи поиска гамильтонова пути к произвольной задаче не известно. В частности, сведение свойства 21.18 говорит о том, чего мы не можем сделать: перехитрить алгоритм, который гарантирует, что можно эффективно найти путь с наименьшим весом в любой заданной сети, если в ней допускаются отрицательные веса ребер. Такая постановка задачи представляется слишком общей. Но мы можем решить ограниченные версии только что упомянутой задачи, хотя это и не так легко, как это имеет место для других ограниченных версий данной задачи (положительные веса и ациклические сети), которыми мы занимались ранее в этой главе.

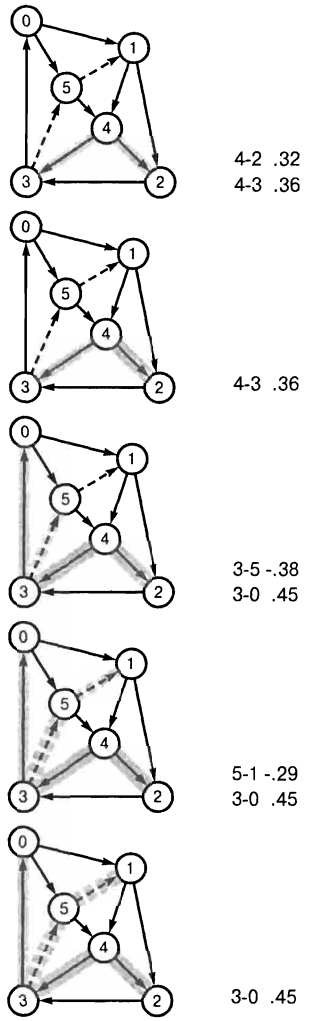


РИСУНОК 21.28. СБОЙ АЛГОРИТМА ДЕЙКСТРЫ (ОТРИЦАТЕЛЬНЫЕ ВЕСА)
В этом примере алгоритм Дейкстры решает, что **4-2** *есть кратчайший путь из* **4** *в* **2** *(длиной 0.32) и упускает более короткий путь* **4-3-5-1-2** *(длиной 0.20).*

В общем случае, как отмечалось в разделе 21.2, алгоритм Дейкстры не работает при наличии отрицательных весов, даже когда мы ограничиваемся рассмотрением сетей, которые не содержат отрицательных циклов. Рисунок 21.28 иллюстрирует это утверждение. Основное затруднение состоит в том, что в этом алгоритме пути рассматриваются в порядке возрастания их длины. Доказательство правильности алгоритма (см. свойство 21.2) предполагает, что добавление ребра к пути делает этот путь более длинным.

Алгоритм Флойда не делает такого допущения и эффективен даже тогда, когда веса ребер могут быть отрицательными. Если нет отрицательных циклов, он вычисляет кратчайшие пути; замечателен тот факт, что в случае существования отрицательных циклов, этот аглоритм обнаруживает, по крайней мере, один из них.

Свойство 21.20. *Алгоритм Флойда решает задачу обнаружения отрицательного цикла и задачу поиска кратчайших путей для всех пар в сетях, которые не содержат отрицательных циклов, за время, пропорциональное* V^3.

Доказательство: Доказательство свойства 21.8 не зависит от того, будут ли веса ребер отрицательными, тем не менее, нам нужно интерпретировать результаты по-иному, когда присутствуют ребра с отрицательными весами. Каждый элемент в матрице свидетельствует о том, что алгоритм обнаружил путь этой длины. В частности, любой отрицательный элемент на диагонали матрицы расстояний суть свидетельство наличия, по крайней мере, одного отрицательного цикла. При наличии отрицательных циклов мы не можем непосредственно делать какие-либо дальнейшие выводы, поскольку пути, которые алгоритм неявно проверяет, не обязательно просты, поскольку они могут содержать один и более обходов по одному и более отрицательных циклов. Однако, если отрицательных циклов нет, то пути, которые вычисляются алгоритмом, просты, поскольку для любого пути с циклом можно предположить существование такого пути, который соединяет те же две точки, но который содержит меньше ребер и не обладает более высоким весом (тот же путь с удаленным циклом). ■

Доказательство свойства 21.20 не дает конкретной информацию о том, как найти конкретный отрицательный цикл из матриц расстояний и путей, вычисляемых по алгоритму Флойда. Мы оставляем это задание читателям на самостоятельную проработку (см. упражнение 21.122).

Алгоритм Флойда решает задачу поиска кратчайших путей для всех пар в графах, которые не содержат отрицательных циклов. Учитывая неудачу алгоритма Дейкстры в сетях, которые, возможно, содержат отрицательные веса, мы могли бы воспользоваться алгоритмом Флойда для решения задачи для всех пар в разреженных сетях, не содержащих отрицательных циклов, за время, пропорциональное V^3. Если мы имеем задачу с одним источником в таких сетях, можно применить это V^3-решение задачи для всех пар. И хотя это сродни пальбе из пушки по воробьям, но все же лучше того, что мы наблюдали для задачи с одним источником. Можем ли мы разработать более быстрые алгоритмы для этих задач – такие, время выполнения которых сравнимо с алгоритмом Дейкстры, когда веса ребер положительны ($E \lg V$ для кратчайших путей из единственного источника и $VE \lg V$ для кратчайших путей всех пар)? На этот вопрос можно ответить утвердительно в отношении задачи всех пар, кроме того, есть возможность снизить стоимость худшего случая до VE для задачи с единственным источником. А вот вопрос преодоления барьера VE для общей задачи поиска кратчайших путей с единственным источником является по-прежнему открытым.

Следующий подход, развитый Р. Беллманом (R. Bellman) и Л. Фордом (L. Ford) в конце пятидесятых годов, предоставляет простую и эффективную основу для начала решения задач поиска кратчайших путей с единственным источником в сетях, не содержащих отрицательных циклов. Чтобы вычислить кратчайшие пути из вершины **s**, мы поддерживаем (как обычно) индексированный вершинами вектор **wt**, такой, что **wt [t]** содержит длину кратчайшего пути из **s** в **t**. Мы инициализируем **wt [s]** значениями **0**, а все другие элементы **wt** — большим сигнальным значением, после чего вычисляем кратчайшие пути, как показано ниже:

Просматривая ребра сети в любом порядке, произвести ослабление вдоль каждого ребра. Выполнить V таких действий.

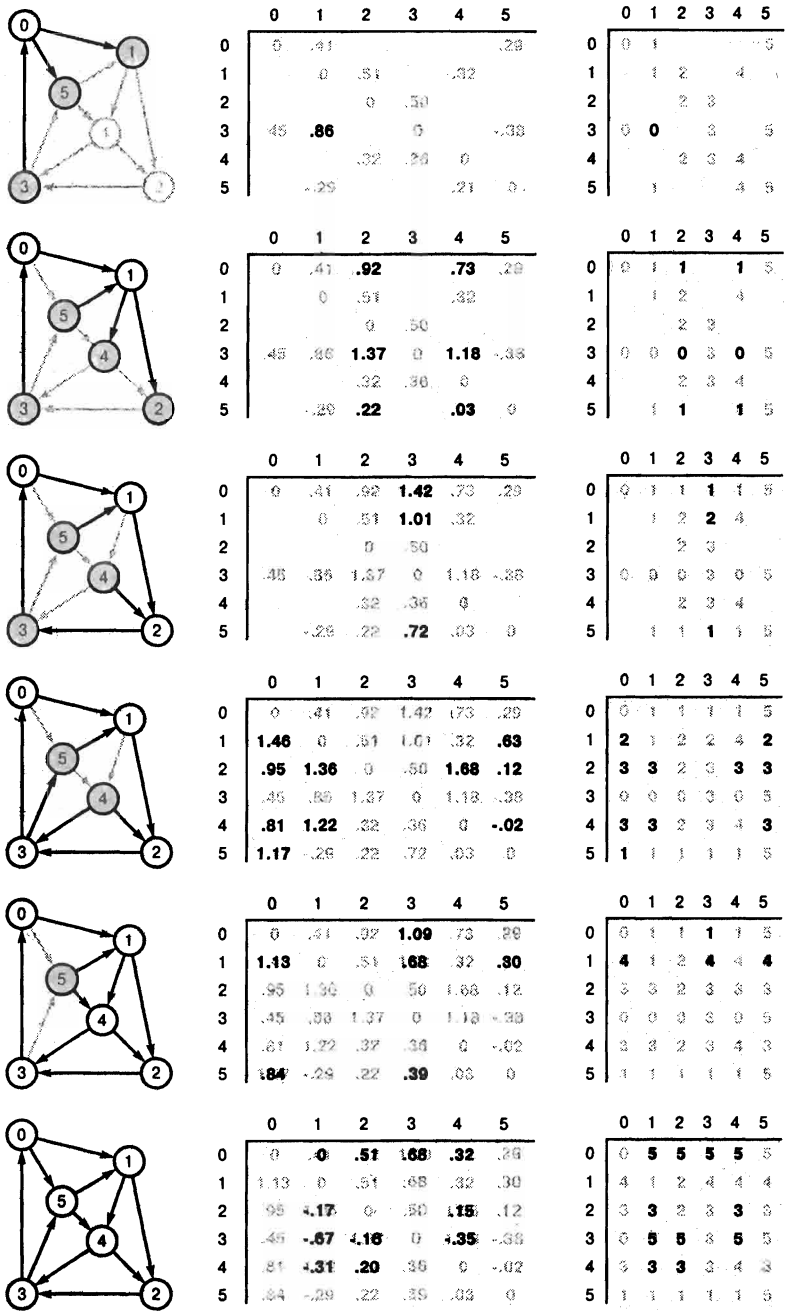


РИСУНОК 21.29. АЛГОРИТМ ФЛОЙДА (ОТРИЦАТЕЛЬНЫЕ ВЕСА)

Эта последовательность показывает построение матриц всех кратчайших путей для орграфа с отрицательными весами по алгоритму Флойда. Первый шаг является таким же, как показанный на рис. 21.14. На втором шаге в игру вступает отрицательное ребро **5-1** *и вскрываются пути* **5-1-2** *и* **5-1-4**. *Алгоритм включает точно ту же последовательность шагов ослабления для любых весов ребра, однако результаты отличаются.*

Для обозначения общего метода получения V проходов по ребрам с просмотром ребер в любом порядке мы будем использовать термин *алгоритм Беллмана-Форда*. Некоторые авторы применяют этот термин для описания более общего метода (см. упражнение 21.130).

Например, для графа, представленного списками смежных вершин, алгоритм Беллмана-Форда поиска кратчайших путей из стартовой вершины s реализуется с помощью инициализации элементов **wt** значениями, большими, чем любая длина пути, и элементов **spt** — нулевыми указателями, и затем в соответствие с кодом, показанным ниже:

```
wt[s] = 0;
for (i = 0; i < G->V(); i++)
  for (v = 0; v < G->V(); v++)
  {
    if (v != s && spt[v] == 0) continue;
    typename Graph::adjIterator A(G, v);
    for (Edge* e = A.beg(); !A.end(); e = A.nxt())
      if (wt[e->w()] > wt[v] + e->wt())
      { wt[e->w()] = wt[v] + e->wt(); st[e->w()] = e; }
  }
```

Этот код демонстрирует простоту базового метода. Однако на практике он не используется из-за того, что простые модификации, как вскоре будет показано, дают реализации, которые являются более эффективными для большинства графов.

Свойство 21.21. С помощью алгоритма Беллмана-Форда можно решить задачу поиска кратчайшего пути для единственного источника в сетях, не содержащих отрицательных циклов, за время, пропорциональное VE.

Доказательство: Мы делаем V проходов через все E ребер, так что полное время будет пропорционально VE. Чтобы доказать, что это вычисление достигнет желаемого результата, методом индукции по i мы показываем, что, для всех вершин v, после i-того прохода **wt[v]** оказывается не больше, чем длина кратчайшего пути из s в v, который содержит i или менее ребер. Утверждение несомненно истинно для i равного 0. Полагая, что требование истинно для i, рассмотрим два возможных случая для каждой данной вершины v: Среди путей из s в v с $i + 1$ или меньшим количеством ребер там либо может, либо не может существовать кратчайший путь с $i + 1$ ребрами. Если наиболее короткий из путей с $i + 1$ или меньше ребрами из s в v имеет длину i или меньше, то **wt[v]** не будет изменяться и останется допустимым. В противном случае, существует путь из s в v с $i + 1$ ребрами, более короткий, чем любой путь из s в v с i или менее ребрами. Этот путь должен сложиться из пути с i ребрами из s в некоторую вершину w плюс ребро w-v. По предположению индукции, **wt[w]** суть верхняя граница на наиболее коротком расстоянии из s в w, и (i+1)-й проход проверяет каждое ребро, является ли оно конечным ребром в новом кратчайшем пути к адресату этого ребра. В частности, проверяется ребро w-v.

После $V - 1$ итераций **wt[v]** определяет нижнюю границу длины любого кратчайшего пути с $V - 1$ или менее ребрами, ведущего из s в v, для всех вершин v. Мы можем остановиться после $V - 1$ итераций из-за того, что любой путь с V или более ребрами должен содержать цикл (с положительной или нулевой стоимостью) и за счет удаления этого цикла мы могли бы отыскать путь с $V - 1$ или менее ребрами, который имеет ту же длину или является более коротким. Поскольку **wt[v]** есть длина *некоторого* пути

из s в v, это значение является также верхней границей длины кратчайшего пути, и таким образом, должно быть равно длине кратчайшего пути.

Хотя это и не рассматривалось явно, то же самое доказательство показывает, что вектор **spt** содержит указатели на ребра в дереве кратчайших путей с корнем в s. ■

Для типовых графов проверка каждого ребра на каждом проходе достаточно расточительна. Действительно, мы можем легко определить априори, что многие ребра не приводят к успешному ослаблению в любом проходе. Фактически, к изменениям могли бы привести только ребра, исходящие из вершины, значение которой изменялось на предыдущем проходе.

Программа 21.9. Алгоритм Беллмана-Форда

Эта реализация алгоритма Беллмана-Форда поддерживает очередь FIFO всех вершин, для которых ослабление вдоль исходящего ребра могло бы оказаться эффективным. Мы берем вершину из очереди и производим ослабление вдоль всех ее ребер. Если любое из них приводит к более короткому пути в некоторую вершину, мы помещаем ее в очередь. Сигнальное значение **G->V** отделяет текущую серию вершин (которые изменились на последней итерации) от следующей серии (которые изменятся на данной итерации) и позволяет остановиться после прохода **G->V**.

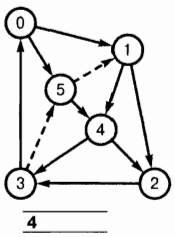
```
SPT(Graph &G, int s) : G(G),
   spt(G.V()), wt(G.V(), G.V())
{ QUEUE<int> Q; int N = 0;
   wt[s] = 0.0;
   Q.put(s); Q.put(G.V());
   while (!Q.empty())
   { int v;
      while ((v = Q.get()) == G.V())
        { if (N++ > G.V()) return; Q.put(G.V()); }
      typename Graph::adjIterator A(G, v);
      for (Edge* e = A.beg(); !A.end(); e = A.nxt())
        { int w = e->w();
          double P = wt[v] + e->wt();
          if (P < wt[w])
            { wt[w] = P; Q.put(w); spt[w] = e; }
        }
   }
}
```

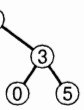
Программа 21.9 является прямой реализацией с использованием очереди FIFO, в которой содержатся эти ребра так, чтобы на каждом проходе проверялись только они. На рис. 21.30 показан пример действия этого алгоритма.

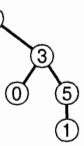
Программа 21.9 эффективна для решения задачи поиска кратчайших путей в реальных сетях с единственным источником, однако быстродействие худшего случая все-таки остается пропорциональной VE. Для плотных графов время выполнения не лучше, чем для алгоритма Флойда, который находит скорее все кратчайшие пути, а не только те, которые исходят из единственного источника. Для разреженных графов реализация алгоритма Беллмана-Форда в программе 21.9 достигает коэффициента V, т.е. является более быстрой, нежели алгоритм Флойда. Тем не менее, она близка к коэффициенту V, более медленному, чем худший случай времени выполнения, которого может показать алгоритм Дейкстры на сетях без ребер с отрицательными весами (см. таблицу 19.2).

РИСУНОК 21.30. АЛГОРИТМ БЕЛЛМАНА-ФОРДА (С ОТРИЦАТЕЛЬНЫМИ ВЕСАМИ)

Этот рисунок показывает результаты применения алгоритма Беллмана-Форда, который предназначен для поиска кратчайших путей из вершины **4** *в сети, изображенной на рис. 21.26. Алгоритм действует в режиме просмотра, где проверяются все ребра, исходящие из всех вершин, помещенных в очередь FIFO. Содержимое очереди показано ниже каждого рисунка графа с использованием затененных элементов, представляющих содержимое очереди для предыдущего прохода. Когда мы находим ребро, которое может уменьшить длину пути из вершины* **4** *в адресат, мы выполняем операцию ослабления, которая помещает вершину-адресат в очередь, а ребро — в SPT. Серые ребра на рисунках графа составляют SPT после каждого этапа, которое показано также в ориентированной форме в центре (все стрелки ребер направлены вниз). Мы начинаем с пустого SPT и вершины* **4** *в очереди (верхняя часть рисунка). На втором проходе мы выполняем ослабление вдоль ребер* **4-2** *и* **4-3**, *оставляя в очереди вершины* **2** *и* **3**. *На третьем проходе мы проверяем, но не выполняем ослабления вдоль* **2-3** *и затем также не выполняем ослабления вдоль* **3-0** *и* **3-5**, *оставляя в очереди вершины* **0** *и* **5**. *На четвертом проходе мы выполняем ослабление вдоль* **5-1** *и затем проверяем, но не выполняем ослабления вдоль* **1-0** *и* **1-5**, *оставляя в очереди вершину* **1**. *На последнем проходе (внизу) мы выполняем ослабление вдоль* **1-2**. *Алгоритм изначально действует подобно BFS, однако в отличие от всех других методов поиска на графе, он может изменять ребра дерева, как это имело место на последнем шаге.*







0-1 .41
1-2 .51
2-3 .50
4-3 .36
3-5 -.38
3-0 .45
0-5 .29
5-4 .21
1-4 .32
4-2 .32
5-1 -.29

	0	1	2	3	4	5
st			4	4	0	
wt			.32	.36	0	

	0	1	2	3	4	5
st	3		4	4	0	3
wt	.81		.32	.36	0	-.02

	0	1	2	3	4	5
st	3	5	4	4	0	3
wt	.81	-.31	.32	.36	0	-.02

	0	1	2	3	4	5
st	3	5	1	4	0	3
wt	.81	-.31	.20	.36	0	-.02

Разумеется, исследовались и другие вариации алгоритма Беллмана-Форда, причем некоторые из них оказываются более быстрыми для задачи с одним источником, чем версия с очередью FIFO в программе 21.9, однако все они в худшем случае требуют времени, пропорционального, по крайней мере, VE (см., например, упражнение 21.132). Базовый алгоритм Беллмана-Форда был разработан десятки лет тому назад, и несмотря на впечатляюще большие шаги в плане достижения эффективности, которые мы наблюдали для множества других задач на графах, до сих пор не просматриваются алгоритмы с лучшей эффективностью для худшего случая на сетях с отрицательными весами.

По сравнению с алгоритмом Флойда, алгоритм Беллмана-Форда является также более эффективным методом для обнаружения, содержит ли сеть отрицательные циклы.

Свойство 21.22. *С помощью алгоритма Беллмана-Форда можно решить задачу обнаружения отрицательного цикла за время, пропорциональное VE.*

Доказательство: Основные положения доказательства свойства 21.21 справедливы даже при наличии отрицательных циклов. Если мы выполняем V-тую итерацию алгоритма, за которой следует любой шаг ослабления, то мы уже обнаружили кратчайший путь с V ребрами, который соединяет s с некоторой вершиной в сети. Любой такой путь должен содержать цикл (соединяющий некоторую вершину w с собой), и в соответствии с предположением индукции, этот цикл должен быть отрицательным, поскольку для того, чтобы w была включена в путь второй раз, путь из s во второе вхождение w должен быть более коротким, чем путь из s в первое вхождение w. Этот цикл будет также присутствовать в дереве, поэтому обнаружить циклы можно было бы и с помощью периодической проверки ребер из **spt** (см. упражнение 21.134).

Сказанное остается в силе только для тех вершин, которые находятся в том же сильно связном компоненте, что и источник s. Чтобы обнаружить отрицательные циклы вообще, можно также вычислить сильно связные компоненты и инициализировать веса для одной вершины в каждом компоненте значениями 0 (см. упражнение 21.126) или добавить фиктивную вершину с ребрами в каждую из остальных вершин (см. упражнение 21.127). ■

В заключение этого раздела рассмотрим задачу поиска кратчайших путей для всех пар. Можно ли улучшить алгоритм Флойда, который выполняется за время, пропорциональное V^3? Использование алгоритма Беллмана-Форда для решения задачи всех пар с помощью решения для каждой вершины задачи с одним источником приводит к худшему случаю времени выполнения, при котором время оказывается пропорциональным V^2E. Мы не рассматриваем это решение более подробно, поскольку существует способ, гарантирующий возможность решения задачи для всех путей за время, пропорциональное $VE \log V$. Он основан на идее, рассмотренной в начале этого раздела: превращение заданной сети в сеть, которая содержит только неотрицательные веса и имеет ту же структуру кратчайших путей.

Фактически, у нас имеется большая свобода в превращении одной сети в другую с отличными весами ребер, но с теми же кратчайшими путями. Предположим, что индексированный вершинами вектор **wt** содержит произвольное распределение весов на вершинах сети G. Для этих весов определим операцию *повторного взвешивания, или переназначение весов, (reweighting)* графа следующим образом:

- Чтобы переназначить вес ребра, необходимо добавить к весу этого ребра разность между весами начальной и конечной вершин ребра.
- Чтобы переназначить вес сети, необходимо переназначить веса всех ребер сети.

Например, следующий простой код переназначает вес сети в соответствие с принятыми соглашениями:

```
for (v = 0; v < G->V(); v++)
  { typename Graph::adjIterator A(G, v);
    for (Edge* e = A.beg(); !A.end(); e = A.nxt())
      e->wt() = e->wt() + wt[v] - wt[e->w()]
  }
```

Эта операция является простым линейным во времени процессом, который строго определен для всех сетей, независимо от весов. Замечательно, что кратчайшие пути в полученной сети остаются теми же, что и кратчайшие пути в исходной сети.

Свойство 21.23. *Переназначение весов сети не воздействует на кратчайшие пути.*

Доказательство: Для любых двух заданных вершин s и t переназначение весов изменяет вес любого *пути* из s в t за счет добавления разности между весами s и t. Это утверждение легко доказать методом индукции по длине пути. При переназначении веса сети вес *каждого* пути из s в t изменяется на одно и то же значение каким бы путь ни был, длинным или коротким. В частности, этот факт предполагает непосредственно, что длина кратчайшего пути между любыми двумя вершинами в преобразованной сети остается такой же, как и длина кратчайшего пути между ними в исходной сети. ■

Поскольку в путях между различными парами вершин веса могут быть переназначены по-разному, переназначение весов могло бы повлиять на такие факторы, которые включают сравнение длин кратчайших путей (например, вычисление диаметра сети). В таких случаях нам следует обратить переназначение весов после завершения вычисления кратчайших путей, но перед использованием результатов.

Переназначение весов не помогает в случае сетей с отрицательными циклами: эта операция не изменяет веса ни одного цикла, так что с ее помощью мы не можем удалить отрицательные циклы. Но для сетей без отрицательных циклов мы можем стремиться обнаружить такое множество вершин, что переназначение весов приведет к весам ребер, которые являются неотрицательными, независимо от того, каковы были первоначальные веса. С неотрицательными весами ребер мы можем затем решить задачу поиска кратчайших путей для всех пар с помощью версии алгоритма Дейкстры для всех пар. Например, рис. 21.31 дает такой пример для нашей типовой сети, а рис. 21.32 показывает вычисление кратчайших путей по алгоритму Дейкстры на преобразованной сети без отрицательных ребер. Следующее свойство показывает, что мы всегда можем найти такое множество весов.

Свойство 21.24. *В произвольной сети, не содержащей отрицательных циклов, выберем произвольную вершину* **s** *и присвоим каждой вершине* **v** *вес, равный длине кратчайшего пути в* **v** *из* **s**. *Переназначение весов сети с этими весами вершин дает в результате неотрицательные веса ребер для каждого ребра, которое соединяет вершины, достижимые из* **s**.

Доказательство: Для заданного произвольного ребра v-w вес v суть длина кратчайшего пути в v, а вес w суть длина кратчайшего пути в w. Если v-w есть конечное ребро

на кратчайшем пути в *w*, то разность между весом *w* и весом *v* в точности составит вес *v-w*. Другими словами, переназначение веса ребра даст вес **0**. Если кратчайший путь через *w* не проходит через *v*, то вес *v* плюс вес *v-w* должен быть больше или равен весу *w*. Другими словами, переназначение веса ребра даст положительный вес. ■

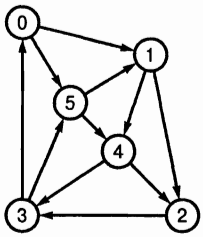
Подобно тому, как это было в случае применения алгоритма Беллмана-Форда для обнаружения отрицательных циклов, существуют два способа, чтобы в произвольной сети без отрицательных циклов сделать вес каждого ребра неотрицательным. Либо можно начать с источника в каждой сильно связной компоненте, либо добавить к каждой вершине сети фиктивную вершину с ребром длины 0. В любом случае результатом будет остовный лес кратчайших путей, которым можно воспользоваться для присвоения весов вершинам (а именно, веса пути в SPT из корня в вершину).

Например, значения весов, выбираемые на рис. 21.31, являются в точности длинами кратчайших путей из 4, поэтому ребра в дереве кратчайших путей с корнем в 4 имеют веса 0 для сети с переназначенными весами.

В конечном итоге мы можем решить задачу поиска кратчайших путей для всех пар в сетях, которые содержат отрицательные веса ребер, но не имеют отрицательных циклов, продолжая следующим образом:

- При помощи алгоритма Беллмана-Форда находим лес кратчайших путей в исходной сети.
- Если алгоритм выявляет отрицательный цикл, сообщаем об этом факте и завершаем работу.
- Переназначаем вес сети на основе леса.
- Применяем версию алгоритма Дейкстры для всех пар к сети с переназначенными весами.

После этих вычислений матрица путей дает кратчайшие пути в обеих сетях, а матрица расстояний дает длины путей в сети с переназначенными весами. Эта последовательность шагов иногда известна как *алгоритм Джонсона (Johnson's algorithm) (см. раздел ссылок).*



	0	1	2	3	4	5
wt	.81	-.31	.20	.36	0	-.02

0-1 .41 + .81 -(-.31) = 1.53
1-2 .51 +(-.31)- .20 = 0
2-3 .50 + .20 - .36 = .34
4-3 .36 + 0 - .36 = 0
3-5 -.38 + .36 -(-.02) = 0
3-0 .45 + .36 - .81 = 0
0-5 .29 + .81 -(-.02) = 1.12
5-4 .21 +(-.02)- 0 = .19
1-4 .32 +(-.31)- 0 = .01
4-2 .32 + 0 - .20 = .12
5-1 -.29 +(-.02)-(-.31) = 0

РИСУНОК 21.31. ПЕРЕНАЗНАЧЕНИЕ ВЕСОВ СЕТИ

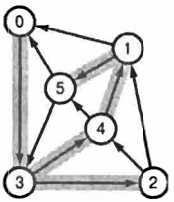
Для любого произвольно заданного распределения весов на вершинах (верхняя часть рисунка) можно переназначить веса всех ребер в сети за счет добавления к каждому весу ребра разности весов его начальной и конечной вершин. Переназначение весов не воздействует на кратчайшие пути, поскольку оно вносит одно и то же изменение в веса всех путей, соединяющих каждую пару вершин. Например, рассмотрим путь **0-5-4-2-3**, *его вес в первоначальной сети есть* $\mathbf{0.29 + 0.21 + 0.32 + 0.50 = 1.32}$, *а в переназначенной сети –* $\mathbf{1.12 + 0.19 + 0.12 + 0.34 = 1.77}$; *эти веса отличаются на* $\mathbf{0.45 = 0.81 - 0.36}$, *т.е. на разность весов вершин* **0** *и* **3**. *При этом веса всех путей между* **0** *и* **3** *изменились на одну и ту же величину.*

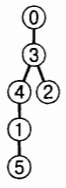
Свойство 21.25. *С помощью алгоритма Джонсона можно решить задачу поиска кратчайших путей для всех пар в сетях, которые не содержат отрицательных циклов, за время, пропорциональное* $VE \log_d V$, *где* $d = 2$, *если* $E < 2V$, *и* $d = E/V$ *в противном случае.*

РИСУНОК 21.32. ВСЕ КРАТЧАЙШИЕ ПУТИ В СЕТИ С ПЕРЕНАЗНАЧЕННЫМИ ВЕСАМИ

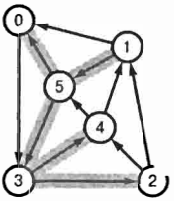
Эти диаграммы показывают SPT для каждой вершины в сети, обратной для сети с переназначенными весами из рис. 21.31, которые были бы получены по алгоритму Дейкстры при вычислении кратчайших путей в исходной сети из рис. 21.26. Эти пути являются такими же, как для сети перед переназначением весов (см. рис. 21.9). Векторы **st** *в этих схемах представляют собой столбцы матрицы путей из рис. 21.26. Векторы* **wt** *в этой схеме соответствуют столбцам в матрице расстояний, но нам необходимо отменить переназначение весов для каждого элемента за счет вычитания веса исходной вершины и добавления веса конечной вершины в пути (см. рис. 21.31). Например, из третьей строки снизу можно видеть, что в обоих сетях кратчайшим путем из* **0** *в* **3** *будет* **0-5-1-4-3**, *а его длина составляет* **1.13** *в показанной здесь сети с переназначенными весами. Сравнивая с рис. 21.31, можно вычислить его длину в исходной сети, вычтя вес* **0** *и добавив вес* **3**, *получив при этом результат* **1.13 − 0.81 + 0.36 = 0.68**, *т.е. элемент в строке* **0** *и столбце* **3** *матрицы расстояний на рис. 21.26. Все кратчайшие пути, ведущие в* **4** *в этой сети, имеют длину* **0**, *поскольку эти пути использовались для переназначения весов.*

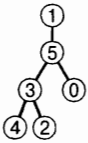
1-0 1.51
2-1 0
3-2 .34
3-4 0
5-3 0
0-3 0
5-0 1.12
4-5 .23
4-1 .01
2-4 .12
1-5 0



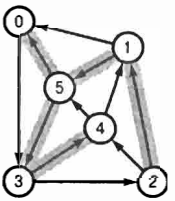


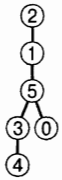
	0	1	2	3	4	5
st	0	4	3	0	3	1
wt	0	.01	.34	0	0	.01



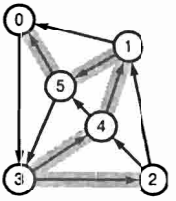


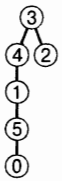
	0	1	2	3	4	5
st	5	1	3	5	3	1
wt	.41	0	1.17	.67	1.03	.29



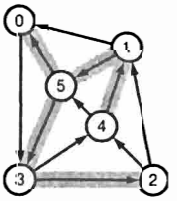


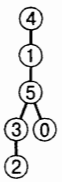
	0	1	2	3	4	5
st	5	2	2	5	3	1
wt	1.12	0	.34	0	0	0



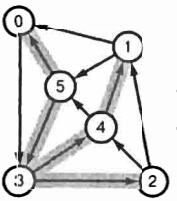


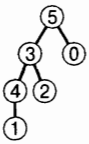
	0	1	2	3	4	5
st	5	4	3	3	3	1
wt	1.13	.01	.34	0	0	.01





	0	1	2	3	4	5
st	5	4	3	5	4	1
wt	0	0	0	0	0	0





	0	1	2	3	4	5
st	5	4	3	5	3	5
wt	1.12	.01	.34	0	0	0

Доказательство: См. свойства 21.22–21.24 и резюме, подведенное в предыдущем абзаце. Граница худшего случая времени выполнения непосредственно вытекает из свойств 21.7 и 21.22. ■

Для реализации алгоритма Джонсона мы объединяем реализацию программы 21.9, код переназначения весов, рассмотренный перед свойством 21.23, и реализацию алгоритма Дейкстры поиска кратчайших путей для всех пар из программы 21.4 (или из программы 20.6 для случая плотных графов). Как отмечалось в доказательстве свойства 21.22, мы должны соответствующим образом модифицировать алгоритм Беллмана-Форда для сетей, которые не являются сильно связными (см. упражнения 21.135–21.137). Для завершения реализации интерфейса поиска кратчайших путей для всех пар можно либо вычислить истинные длины путей за счет вычитания веса начальной и добавления веса конечной вершины (т.е. отменить операцию переназначения весов для путей) при копировании двух векторов в матрицы расстояний и путей в алгоритме Дейкстры, либо поместить эти вычисления в **GRAPHdist** в реализации АТД.

Для сетей, не содержащих отрицательных весов, задача выявления циклов решается более просто, чем задача вычисления кратчайших путей из единственного источника ко всем другим вершинам; при этом последняя решается более просто, чем задача вычисления кратчайших путей, соединяющих все пары вершин. Данные факты соответствуют нашим интуитивным представлениям. В противоположность этому, аналогичные факты для сетей, которые содержат отрицательные веса, нам кажутся противоестественными: алгоритмы, которые мы обсудили в этом разделе, показывают, что для сетей, которые содержат отрицательные веса, лучшие из известных алгоритмы решения трех упомянутых задач обладают сходными характеристиками быстродействия для худшего случая. Например, в худшем случае, определение, содержит ли сеть единственный отрицательный цикл, приблизительно столь же сложно, как и поиск всех кратчайших путей в сети того же размера, которая не имеет отрицательных циклов.

Упражнения

▷ **21.109**. Измените, введя масштабирование, генераторы случайных сетей из упражнений 21.6 и 21.7, чтобы выдавать веса из диапазона между a и b (где a и b принимают значения между -1 и 1).

▷ **21.110**. Измените генераторы случайных сетей из упражнений 21.6 и 21.7, чтобы порождать отрицательные веса за счет вычитания фиксированного процента (значение которого поставляется клиентом) от весов ребер.

○ **21.111**. Разработайте клиентские программы, которые используют генераторы из упражнений 21.109 и 21.110 для порождения сетей, содержащих большой процент отрицательных весов, но имеющих не более чем несколько отрицательных циклов, для возможно большего интервала значений V и E.

21.112. Найдите таблицу конверсии валют. Воспользуйтесь ею для построения таблицы арбитражных операций. *Примечание*: Избегайте таблиц, которые выводятся (рассчитываются) из небольшого количества значений и которые, таким образом, не дают достаточно точной информации о курсах. *Бесплатный совет*: Играйте на валютной бирже!

● **21.113**. Постройте последовательность таблиц арбитражных операций, используя источник, выбранный в упражнении 21.112 (каждый источник периодически публикует

различные таблицы). Найдите в таблицах все возможности для совершения арбитражных операций и попытайтесь вскрыть их характерные особенности. Например, сохраняются ли отклонения на протяжении нескольких дней, или же они быстро восстанавливаются сразу после возникновения?

21.114. Разработайте модель для генерации случайных задач арбитражных операций. Ваша цель состоит в порождении таблиц, которые как можно более похожи на таблицы, используемые в упражнении 21.113.

21.115. Разработайте модель для генерации случайных задач календарного планирования работ, включающих конечные сроки. Ваша цель состоит в порождении нетривиальных задач, которые, по возможности, должны быть выполнимыми.

21.116. Измените интерфейс и реализацию из упражнения 21.101, чтобы дать клиентам возможность формулировать и решать задачи планирования работ, которые включают конечные сроки, используя сведение к задаче поиска кратчайших путей.

○ **21.117**. Объясните, почему следующее рассуждение ошибочно: задача поиска кратчайших путей сводится к задаче разностных ограничений построением, используемым в доказательстве свойства 21.15, а задача разностных ограничений тривиально сводится к линейному программированию, следовательно, принимая во внимание свойство 21.17, линейное программирование является NP-трудным.

21.118. Сводится ли задача поиска кратчайших путей в сетях без отрицательных циклов к задаче планирования работ с конечными сроками? (Эквивалентны ли эти две задачи?) Обоснуйте ответ.

○ **21.119**. Найдите цикл с наименьшим весом (лучшую возможность для совершения арбитражных операций) в примере, показанном на рис. 21.27.

▷ **21.120**. Докажите, что задача поиска цикла наименьшего веса в сети, которая может иметь ребра с отрицательными весами, является NP-трудной.

▷ **21.121**. Покажите, что алгоритм Дейкстры работает правильно для сети, в которой ребра, исходящие из источника, являются единственными ребрами с отрицательными весами.

▷ **21.122**. Разработайте класс, основанный на алгоритме Флойда, который обеспечивает клиентов возможностью проверить существование в сети отрицательных циклов.

21.123. Воспользовавшись алгоритм Флойда, покажите в стиле рис. 21.29 вычисления всех кратчайших путей для сети, определяемой в упражнении 21.1, с отрицательными весами ребер 5-1 и 4-2.

● **21.124**. Является ли алгоритм Флойда оптимальным для полных сетей (сети с V^2 ребрами)? Обоснуйте ответ.

21.125. Покажите в стиле рис. 21.30–21.32 вычисление по алгоритму Беллмана-Форда всех кратчайших путей сети, определенной в упражнении 21.1, с отрицательными весами на ребрах 5-1 и 4-2.

▷ **21.126**. Разработайте класс, основанный на алгоритме Беллмана-Форда, который обеспечивает клиентов возможностью проверить существование в сети отрицательных циклов, используя метод старта из источника в каждой сильно связной компоненте.

▷ **21.127**. Разработайте класс, основанный на алгоритме Беллмана-Форда, который обеспечивает клиентов возможностью проверить существование в сети отрицательных циклов, используя фиктивную вершину с ребрами во все вершины сети.

○ **21.128**. Приведите семейство графов, для которых программа 21.9 на поиск отрицательных циклов затрачивает время, пропорциональное *VE*.

▷ **21.129**. Покажите расписание, которое вычисляется программой 21.9 для задачи календарного планирования с конечными сроками, описанной в упражнении 21.89.

○ **21.130**. Докажите, что следующий общий алгоритм решает задачу поиска кратчайших путей с единственным источником: "ослабить *произвольное* ребро; продолжать так до тех пор, пока существуют ребра, которые можно ослабить".

21.131. Измените реализацию алгоритма Беллмана-Форда в программе 21.9, воспользовавшись рандомизированной очередью вместо очереди FIFO. (Результаты решения упражнения 21.130 доказывают, что этот метод допустимый.)

○ **21.132**. Измените реализацию алгоритма Беллмана-Форда в программе 21.9, чтобы вместо очереди FIFO использовать такую очередь с двусторонним доступом, в которую ребра помещаются в соответствии со следующим правилом: если ребро уже находится в очереди с двусторонним доступом, поместить его в ее начало (как в стеке), а если оно встречается в первый раз, то поместить его в ее конец (как в обычной очереди).

21.133. Проведите эмпирические исследования с целью сравнения производительности реализаций из упражнений 21.131 и 21.132 с программой 21.9 на различных общих сетях (см. упражнения 21.109–21.111).

○ **21.134**. Измените реализацию алгоритма Беллмана-Форда в программе 21.9, чтобы реализовать функцию, которая возвращает индекс произвольной вершины в произвольном отрицательном цикле или -1, если сеть не содержит отрицательных циклов. Когда отрицательный цикл присутствует, эта функция должна также построить вектор **spt**, такой что за счет обычного следования по связям в этом векторе (начиная с возвращаемого значения), можно пройти вдоль цикла.

○ **21.135**. Измените реализацию алгоритма Беллмана-Форда в программе 21.9, чтобы установить веса вершин так, как это требуется для алгоритма Джонсона, воспользовавшись следующим методом. Каждый раз, когда очередь становится пустой, сканируется вектор **spt**, чтобы найти вершину, вес которой еще не установлен, и алгоритм запускается повторно с этой вершиной в качестве источника (для установки весов всех вершин, находящихся в той же сильно связной компоненте, что и новый источник); описанная процедура продолжается, пока не будут обработаны все сильно связные компоненты.

▷ **21.136**. Разработайте реализацию интерфейса АТД поиска кратчайших путей для всех пар в разреженных сетях (основанную на алгоритме Джонсона) за счет внесения соответствующих изменений в программы 21.9 и 21.4.

21.137. Разработайте реализацию интерфейса АТД поиска кратчайших путей для всех пар в плотных сетях (основанную на алгоритме Джонсона) (см. упражнения 21.136 и 21.43). Проведите эмпирические исследования с целью сравнения полученной реализации с алгоритмом Флойда (программа 21.5) на различных общих сетях (см. упражнения 21.109–21.111).

- **21.138**. Добавьте функцию-элемент к решению упражнения 21.137, которая дала бы клиенту возможность уменьшить стоимость ребра. Она должна возвращать флаг, который указывает, создает ли это действие отрицательный цикл. Если нет, то должны обновиться матрицы путей и расстояний, отразив любой новый кратчайший путь. Функция должна выполняться за время, пропорциональное V^2.
- **21.139**. Расширьте решение упражнения 21.138 функциями-элементами, которые разрешают клиентам вставлять и удалять ребра.
- **21.140**. Разработайте алгоритм, который преодолевает барьер VE для задачи поиска кратчайших путей в общих сетях с единственным источником для частного случая, когда известно, что абсолютные значения весов ограничены некоторой константой.

21.8 Перспективы

Таблица 21.4 подводит итог алгоритмам, которые обсуждались в настоящей главе, и приводит характеристики их производительности для худшего случая. Эти алгоритмы широко применимы, поскольку, как упоминалось в разделе 21.6, задачи поиска кратчайших путей связаны с большим числом других задач, имеющих конкретное прикладное значение, что непосредственно приводит к эффективным алгоритмам для решения всего класса задач, или, по крайней мере, указывает на существование таких алгоритмов.

Таблица 21.4 Стоимости алгоритмов поиска кратчайших путей

В этой таблице сравниваются стоимости (времена выполнения в худшем случае) различных алгоритмов поиска кратчайших путей, которые рассматривались в настоящей главе. Границы худшего случая, обозначенные как умеренные, не могут оказаться полезными в прогнозировании производительности на реальных сетях, в частности, сказанное относится к алгоритму Беллмана-Форда, который обычно выполняется за линейное время.

Ограничения веса	Алгоритм	Стоимость	Комментарий
С единственным источником			
Неотрицательный	Дейкстры	V^2	Оптимальный (плотные сети)
Неотрицательный	Дейкстры (PFS)	$E \lg V$	Консервативная граница
Ациклический	Исходная очередь	E	Оптимальный
Бе отрицательных циклов	Беллман ± Форд	VE	Есть возможности для усовершенствования?
Нет	Вопрос пока открыт	?	NP-трудный
Все пары			
Неотрицательный	Флойда	V^3	Идентичный для всех сетей
Неотрицательный	Дейкстры (PFS)	$VE \lg V$	Консервативная граница
Ациклический	DFS	VE	Идентичный для всех сетей
Без отрицательных циклов	Флойда	V^3	Идентичный для всех сетей
Без отрицательных циклов	Джонсона	$VE \lg V$	Консервативная граница
Нет	Вопрос пока открыт	?	NP-трудный

Общая задача поиска кратчайших путей в сетях, в которых веса ребер могут принимать отрицательные значения, является неразрешимой. Задачи поиска кратчайших путей представляют собой хорошую иллюстрацию той черты, которая часто отделяет неразрешимые задачи от легких, поскольку у нас имеются многочисленные алгоритмы для решения различных вариантов этой задачи, когда мы накладываем ограничения на сети: то ли на присутствие ребер с положительными весами, то ли на ацикличность, или даже когда мы накладываем ограничения на подзадачи, где допускаются отрицательные веса ребер, но нет отрицательных циклов. Некоторые алгоритмы оптимальны или близки к таковым, хотя есть существенный разрыв между наилучшей известной нижней границей и лучшим известным алгоритмом для задачи с одним источником в сетях, которые не содержат отрицательных циклов, и для задачи всех пар в сетях с неотрицательными весами.

Эти алгоритмы полностью основаны на небольшом количестве абстрактных действий и могут быть приведены в общей постановке. Говоря конкретно, единственными двумя действиями, которые мы выполняем над весами ребер, являются сложение и сравнение: любая постановка, в которой эти действия имеют смысл, может служить платформой для алгоритмов поиска кратчайших путей. Как уже отмечалось ранее, эта точка зрения объединяет наши алгоритмы для вычисления транзитивного замыкания орграфа с алгоритмами поиска кратчайших путей в сетях. Сложность, привносимая отрицательными весами ребер, соответствует свойству монотонности на этих абстрактных операциях: если есть возможность гарантировать, что сумма двух весов никогда не меньше любого из весов, то можно использовать алгоритмы из разделов 21.2–21.4. Если же подобной гарантии дать нельзя, необходимо использовать алгоритмы из раздела 21.7. Инкапсуляция упомянутых особенностей в АТД не составляет особого труда, к тому же она расширяет применимость данных алгоритмов.

Задачи поиска кратчайших путей выводят нас на перекресток между элементарными алгоритмами обработки графов и задачами, которые мы не можем решить. Они дают начало ряду других классов задач аналогичного характера, в том числе *задачи о потоках в сетях* и *линейное программирование*. Как и при поиске кратчайших путей, в этих областях имеется четкая грань между легкими и неразрешимыми задачами. Существуют не только многочисленные эффективные алгоритмы, доступные при соответствующих ограничениях, но также остаются широкие возможности поиска лучших алгоритмов, равно как и встречаются случаи, когда доводится лицом к лицу сталкиваться с однозначно NP-трудными задачами.

Многие из таких задач были четко сформулированы как задачи из области исследования операций еще до прихода компьютеров и соответствующих им алгоритмов. Исторически дисциплина исследования операций фокусировалась на общих математических и алгоритмических моделях, тогда как информатика – на конкретных алгоритмических решениях и базовых абстракциях, которые могут как иметь выход на эффективные реализации, так и составить основу построения общих решений. Поскольку как модели из области исследования операций, так и базовые алгоритмические абстракции из области информатики были ориентированы на разработку компьютерных реализаций, которые могут решить практические задачи большой размерности, в некоторых областях деятельности невозможно провести четкую границу между исследованием операций и информатикой. Например, в обеих областях исследователи до сих пор ищут эффективные решения задач, подобных проблеме поиска кратчайших путей.

Глава 22

Потоки в сетях

Графы, орграфы и сети – это всего лишь математические абстракции, однако они приносят практическую пользу, поскольку позволяют решать множество важных задач. В этой главе мы расширим сетевую модель решения задач с таким расчетом, чтобы она охватывала динамические ситуации, отождествляющие в нашем воображении движение материалов по сети, в которой различным маршрутам назначаются различные стоимости. Такие расширения позволяют решать удивительно широкие классы задач с длинным списком применений.

Мы увидим, что эти задачи и приложения могут быть решены с помощью нескольких естественных моделей, при этом мы можем переходить от одной из них к другой посредством метода *сведения к другим задачам (reduction)*. Существует несколько различных технически эквивалентных способов формулирования базовых задач. Для реализации алгоритмов, которые их решают, мы выбираем две конкретные задачи, составляем эффективные алгоритмы их решения, а затем разрабатываем алгоритмы, которые решают другие задачи, отыскивая возможности сведения их к уже известным задачам.

В реальной жизни мы не всегда располагаем свободой выбора, какую предполагает этот идеализированный сценарий, поскольку возможность сведения решения одной задачи к решению другой установлена не для каждой пары задач и поскольку известны лишь немногие оптимальные алгоритмы решения любой из этих задач. Возможно, что пока еще не найдено прямого решения заданной задачи, и вполне вероятно, что пока еще не удалось найти эффективного метода, позволяющего сводить решение одной задачи из заданной пары задач к решению другой. Постановка той или иной задачи в виде задачи определения потоков в сетях, ис-

следование которой мы проведем в данной главе, имела успех не только потому, что к ней легко сводится решение многих практических задач, но также и потому, что были разработаны многочисленные эффективные алгоритмы решения базовых задач потоков в сетях.

Приводимый ниже пример служит иллюстрацией широты круга задач, которые могут решаться с помощью моделей потоков в сетях, предлагаемых ею алгоритмов и реализаций. Его можно разбить на общие категории, известные как задачи *распределения (distribution)*, *сочетания (matching)* и *поиска сечения (cut)*; мы поочередно рассмотрим каждую из них. Мы не будем сосредоточивать внимание на конкретных деталях этих примером, зато выделим несколько различных связанных между собой задач. Далее в этой главе, когда наступит пора разработки и реализации алгоритмов, мы дадим строгие формулировки многих из упомянутых здесь задач.

В задаче распределения производится перемещение объектов из одного места сети в другое. Производится ли перевозка гамбургеров или цыплят в экспресс-закусочные, либо игрушек или одежды в магазины уцененных товаров по шоссе через всю страну, или же распределение программных продуктов по компьютерам либо информационных бит для отображения на мониторах во всех уголках света — по сути дела, это одна и та же задача. Задачи распределения сводит трудные проблемы, с которыми мы сталкиваемся при выполнении сложных операций, к решению типовых задач. Алгоритмы, предназначенные для их решения, нашли широкое применение, а без некоторых их них достаточно многочисленные приложения просто не будут работать.

Распределение товаров. У крупной компании имеются заводы, на которых изготавливаются товары, оптовые базы, предназначенные для временного хранения товаров, и розничные торговые точки, в которых товары продаются конечным потребителям. Компания должна регулярно доставлять товары с заводов в розничные торговые точки через оптовые базы, используя для этой цели каналы распределения, которые обладают различными пропускными способностями и различной стоимостью доставки единицы продукции. Можно ли организовать доставку товаров со складов в розничные торговые точки таким образом, чтобы спрос был удовлетворен везде? Каким будет маршрут, выбранный по критерию наименьшей стоимости? Рисунок 22.1 служит иллюстрацией задачи распределения товаров.

На рис. 22.2 показана *транспортная задача (transportation problem)*, представляющей собой частный случай задачи распределения товаров, в которой игнорируются центры распределения и пропускные способности каналов. Эта версия задачи распределения товаров важна сама по себе и представляет интерес (как мы увидим в разделе 22.7) не только в плане непосредственного применения, но и в силу того обстоятельства, что она не является каким-то "специальным случаем" — в самом деле, по степени сложности решения она эквивалентна общей версии рассматриваемой задачи.

Обмен данными. Коммуникационная сеть получает некоторое множество запросов на передачу сообщений между серверами, подсоединенными к этой сети через каналы связи (абстрактные провода), которые передают данные с различной скоростью передачи. Какой должны быть максимальная скорость передачи данных, при которой информация может передаваться между конкретными серверами в такой сети? Если с каждым каналом передачи данных связана конкретная стоимость передачи, то каким будет минимальный по стоимости вариант передачи данных с заданной скоростью, которая меньше максимально возможной?

РИСУНОК 22.1. ЗАДАЧА РАСПРЕДЕЛЕНИЯ

*В этом примере задачи распределения мы имеем три вершины снабжения (вершины от **0** до **2**), четыре распределительных пункта (вершины от **3** до **6**), три вершины потребления (вершины от **7** до **9**) и двенадцать каналов. Каждая вершина снабжения характеризуется собственным уровнем производства, каждая вершина потребления характеризуется собственным уровнем потребления, и каждый канал обладает определенной максимальной пропускной способностью и устанавливает собственную стоимость доставки единицы продукции. Проблема заключается в том, чтобы минимизировать стоимость доставки материалов по каналам доставки (при условии, что пропускная способность каналов нигде не будет превышена) таким образом, чтобы общий объем материала, поставляемого из каждой вершины снабжения, соответствовал уровню производимой продукции; чтобы общий объем материала, доставляемого на вершины потребления, соответствовал уровню его потребления; и чтобы общая интенсивность поступления материала в каждый распределительный пункт была равна интенсивности, с которой материал покидает их.*

снабжение	каналы поставки	стоимость	пропускная способность канала
0: 4	0-3:	2	2
1: 4	0-7:	1	3
2: 6	1-4:	5	5
распределение	2-4:	3	4
3	2-9:	1	4
4	3-5:	3	2
5	3-6:	4	1
6	4-5:	2	5
потребность	4-6:	1	4
7: 7	5-7:	6	6
8: 3	5-8:	3	4
9: 4	6-9:	4	3

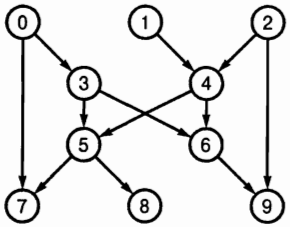
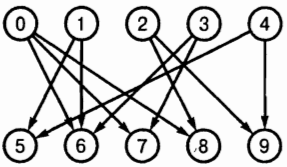


РИСУНОК 22.2. ТРАНСПОРТНАЯ ЗАДАЧА

*Транспортная задача во многом подобна задаче распределения, только в ней не учитываются пропускная способность каналов и отсутствуют распределительные пункты. В рассматриваемом примере мы имеем пять вершин снабжения (от **0** до **4**), пять вершин потребления (от 5 до 9) и двенадцать каналов. Задача заключается в том, чтобы найти способ распределения материала по каналам минимальной стоимости с таким расчетом, чтобы поставки материала везде соответствовали спросу на него. В частности, нам нужно присвоить каналам такие веса (интенсивность распределения), чтобы сумма весов исходящих ребер была равна притоку материала в каждую снабжающую вершину; сумма весов входящих ребер должна быть равна суммарной потребности каждой потребляющей вершины; общая стоимость (сумма произведений весов на стоимость ребра для всех ребер) должна быть минимальной по всем присвоенным весам.*

снабжение	каналы поставки	стоимость
0: 3	0-6:	2
1: 4	0-7:	1
2: 6	0-8:	5
3: 3	1-6:	3
4: 2	1-5:	1
потребность	2-8:	3
5: 6	2-9:	4
6: 6	3-6:	2
7: 7	3-7:	1
8: 3	4-9:	6
9: 4	4-5:	3



Транспортные потоки. Городское управление должно разработать план эвакуации людей из города в случае возникновения критических ситуаций. Какое минимальное время потребуется для эвакуации города в предположении, что мы можем регулировать транспортные потоки в такой степени, которая позволяет осуществить минимальную эвакуацию? Планировщики транспортных потоков могут ставить подобные вопросы, принимая решение, где строить новые дороги, мосты или туннели, которые способны смягчить проблему уличного движения в часы пик или в выходные дни.

В задаче *сочетания* сеть представляет собой возможные способы соединения двух пар вершин. Наша цель в этом случае заключается в таком выборе среди множества соединений (в соответствии со специальным критерием), при котором ни одна из вершин не выбирается дважды. Другими словами, избранное множество ребер определяет путь от одной вершины к другой. Мы можем выбирать колледжи для студентов, работу для соискателей, курсы для свободных часов в школе или членов Конгресса США (Государственной Думы России, Верховной Рады Украины) на посты в различных комитетах. В каждой их таких ситуаций мы можем воспользоваться широким разнообразием критериев, определяющих характер выбора.

Задача о трудоустройстве. Служба трудоустройства организует интервью для группы студентов с некоторым числом компаний; в результате этих интервью появляется ряд предложений работы. Исходя из предположения, что за интервью следует предложение работы, существует взаимная заинтересованность в том, чтобы студенты принимали предложения от компании, поэтому каждая из сторон стремится добиваться максимально возможного трудоустройства. Из примера, представленного на рис. 22.3, следует, что эта задача может оказаться достаточно сложной.

Alice	Adobe
Adobe	Alice
Apple	Bob
HP	Dave
Bob	Apple
Adobe	Alice
Apple	Bob
Yahoo	Dave
Carol	HP
HP	Alice
IBM	Carol
Sun	Frank
Dave	IBM
Adobe	Carol
Apple	Eliza
Eliza	Sun
IBM	Carol
Sun	Eliza
Yahoo	Frank
Frank	Yahoo
HP	Bob
Sun	Eliza
Yahoo	Frank

РИСУНОК 22.3. ВОПРОСЫ ТРУДОУСТРОЙСТВА

Предположим, что шесть студентов ищут работу, а шесть компаний намерены принять на работу по одному студенту. Два представленных на рисунке списка (один отсортирован по студентам, другой отсортирован по компаниям) образуют список предложений работы, который отражает взаимный интерес в сочетании студентов и предлагаемых работ. Существует ли способ подыскать каждому студенту работу таким образом, чтобы каждое вакантное место было заполнено, а каждый студент получил работу? Если это не удается, то каково число вакансий, которые могут быть заполнены?

Задача сочетания с минимальными расстояниями. Пусть даны два множества, содержащие по N точек; требуется найти множество линейных сегментов, при условии, что границы каждого сегмента совпадают с точками каждого из заданных множеств, суммарная длина которых минимальна. Одним из приложений этой чисто геометрической задачи является радарная система слежения. Каждый поворот радиолокатора дает множество точек, которые представляют самолеты. Мы полагаем, что самолеты достаточно далеко удалены друг от друга в пространстве, так что решение этой задачи позволяет связать положение каждого самолета на одном повороте радиолокатора с его позицией на следующем повороте, сле-

довательно, можно получить пути следования всех самолетов. К этой схеме можно свести и другие приложения, использующие выборку данных.

В задаче *сечения*, подобной той, что иллюстрируется рисунком 22.4, мы удаляем ребра с целью разбиения сети на две или большее число частей. Задачи о сечениях непосредственно связаны с фундаментальными вопросами связности графа, которые обсуждались в главе 18. В этой главе мы будем изучать центральную теорему, которая обнаруживает удивительную связь между задачей о сечении и задачей о потоках в сетях, существенно расширяя применимость алгоритмов вычисления потоков в сетях.

Надежность сети. Упрощенная модель представляет телефонную сеть как некоторое множество проводов, которые соединяют телефонные аппараты через коммутаторы так, что существует возможность посредством переключений установить через магистральные линии коммутируемую линию связи, соединяющую любые два заданных телефонных аппарата. Каким будет максимальное число магистральных линий, которые можно отключить без нарушения связи между любой парой телефонных коммутаторов?

Отсечение линий снабжения. Когда та или иная страна ведет военные действия, она осуществляет материальное снабжение войск через систему взаимосвязанных дорог. Противник может прервать снабжение войск, нанося по дорогам бомбовые удары, при этом предполагается, что число бомб пропорционально ширине дороги. Какое минимальное число бомб понадобится противнику, чтобы лишить войска снабжения?

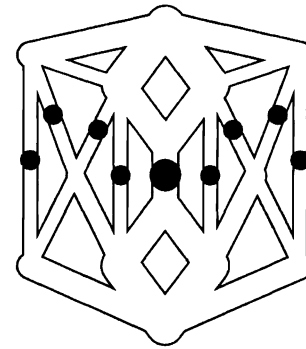


РИСУНОК 22.4. ОТСЕЧЕНИЕ ЛИНИЙ СНАБЖЕНИЯ

На этой диаграмме представлена система дорог, соединяющих базы снабжения армии в ее верхней части с войсками в ее нижней части. Черные точки изображают план бомбовых ударов противника, которые должны отделить войска от их баз снабжения. Задача противника заключается в том, чтобы минимизировать стоимость бомбардировок (возможно, исходя из предположения, что стоимость отсечения ребра пропорциональна его ширине), а целью армии является разработка такой сети дорог, чтобы существенно увеличить минимальную стоимость бомбардировок. Эта же модель полезна для повышения надежности сетей связи и во множестве других приложений.

Каждое из указанных выше приложений немедленно порождает многочисленные сопутствующие вопросы, кроме того, существуют и другие родственные модели, такие как задачи календарного планирования, которым была посвящена глава 21. На протяжении этой главы мы будем рассматривать и другие примеры, но все это составит лишь незначительную часть важных, непосредственно взаимосвязанных практических задач.

Изучаемая в настоящей главе модель потоков в сети важна не только потому, что она ставит перед нами две четко сформулированные задачи, к которым сводятся многие задачи, возникающие на практике, но еще и потому, что в нашем распоряжении имеются эффективные алгоритмы решения этих двух задач. Подобная широкая применимость стимулировала разработку многочисленных алгоритмов и реализаций. Решение, которое мы будем рассматривать, порождает некоторое противоречие между попыткой достичь универсальной применимости и поиском эффективного решения конкретных задач. Изучение алгоритмов вычисления потоков в сети само по себе является захватывающим заня-

тием, поскольку оно подводит нас ближе к компактной и элегантной реализации, которая достигает обеих целей.

Мы полагаем существование двух конкретных задач в рамках модели потоков в сетях, а именно, задачи о *максимальном потоке (maxflow)* и задачи *о потоке минимальной стоимости (mincost-flow)*. Несложно будет обнаружить их определенную связь с описанными выше моделями решения задач, с моделью кратчайшего пути из главы 21, с LP-моделью (linear-programming model — моделью линейного программирования) из части 8, а также со множеством специальных моделей задач, в том числе и только что обсужденных.

На первый взгляд может показаться, что многие из этих задач не имеют ничего общего с задачами о потоках в сетях. Выявление зависимости между заданной задачей и известными задачами часто является важным шагом при разработке подхода к решению этой задачи. Более того, этот шаг часто важен еще и потому, что мы должны иметь представление о той хрупкой границе, которая отделяет тривиальные задачи от труднорешаемых задач, прежде чем предпринимать попытки разработки реализаций. Внутренняя структура задач и различного вида зависимости между задачами, которые мы рассматриваем в этой главе, создают благоприятный контекст для выбора подхода к решению подобного рода задач.

В рамках приближенной классификации алгоритмов по категориям, которую мы начали в главе 17, алгоритмы, которые будут изучаться в этой главе, попадают в категорию "легко решаемых", поскольку у нас имеется простые реализации, которые гарантированно выполняются за время, пропорциональное полиному от размера сети. Другие реализации хотя и не гарантируют полиномиального времени выполнения в худшем случае, тем не менее, они компактны и элегантны и, как показала практика, пригодны для решения широкого круга практических задач, аналогичных обсуждаемым здесь. Мы рассмотрим их подробно, поскольку в дальнейшем они принесут нам несомненную пользу. Исследователи по-прежнему заняты поиском более быстродействующих алгоритмов с тем, чтобы стали возможными крупные приложения, а также чтобы снизить себестоимость критических приложений. Идеальные оптимальные алгоритмы решения задач о потоках в сети, обеспечивающие максимально возможное быстродействие, еще ждут своих первооткрывателей.

С другой стороны, известно, что некоторые задачи, которые сводятся к задачам о потоках в сетях, решаются эффективнее с помощью специальных алгоритмов. В принципе, мы можем заняться реализацией и совершенствованием таких специализированных алгоритмов. Однако, несмотря на то, что этот поход продуктивен в отдельных случаях, алгоритмы решения множества других задач (отличных от тех, что решаются методом сведения к потокам в сети) не существуют. Но даже в тех случаях, когда специализированные алгоритмы существуют, разработка реализаций, способных превзойти удачные программы, в основе которых лежит задача о потоках в сетях, связана с существенными сложностями. Более того, исследователи продолжают работы по совершенствованию алгоритмов решения задач о потоках в сетях, так что удачные алгоритмы этого типа вполне могут превзойти известные методы решения данной практической задачи.

С другой стороны, задачи о потоках в сетях представляют собой специальные случаи еще более общей LP-задачи, которые нам предстоит рассмотреть в части 8. И хотя мы могли воспользоваться (а многие так и делают) алгоритмом решения LP-задач для реше-

ния задач о потоках в сетях, рассматриваемые далее алгоритмы решения задач о потоках в сети намного проще и эффективнее, чем те, которые решают LP-задачи. Однако исследователи продолжают совершенствовать программы решения LP-задач, и все еще остается возможность того, что качественный алгоритм решения LP-задачи, будучи использованным для решения задач о потоках в сетях, может когда-нибудь превзойти по всем параметрам алгоритмы, которые мы рассматриваем в данной главе.

Классическое решение задач о потоках в сетях тесно связаны с другими, уже изученными, алгоритмами на графах, и мы можем писать удивительно компактные программы их решения, воспользовавшись разработанными ранее алгоритмическими инструментами. Как мы могли убедиться во множестве других ситуаций, качественные алгоритмы и структуры данных иногда позволяют существенно уменьшить время выполнения программ. Изучение методов разработки более совершенных реализаций классических алгоритмов, способных решать обширные классы задач, продолжается, и время от времени появляются новые подходы.

В разделе 22.1 мы проведем исследование свойств *транспортных сетей (flow networks)*, в рамках которого мы интерпретируем веса ребер как *пропускные способности (capacities)* и рассматриваем свойства *потоков (flows)*, представленные вторым множеством весов ребер, которые удовлетворяют некоторым естественным условиям. Далее мы рассмотрим задачу о *максимальном потоке*, которая заключается в вычислении лучшего в некотором специальном техническом смысле потока. В разделах 22.2 и 22.3 мы рассмотрим два подхода к решению задачи о максимальном потоке и изучим некоторое множество различных реализаций ее решения. Многие из алгоритмов и структур данных, которые мы рассматривали ранее, непосредственно зависят от разработки эффективного решения задачи о максимальном потоке. У нас еще нет наилучшего из возможных алгоритмов решения задачи о максимальном потоке, поэтому мы пока рассмотрим конкретные решения, используемые на практике. Чтобы продемонстрировать масштабность и разноплановость задачи поиска максимального потока, в разделе 22.4 мы рассмотрим другие формулировки этой задачи, равно как и возможность ее сведения к другим задачам.

Алгоритмы решения задачи о максимальном потоке и их реализации подготавливают нас к обсуждению более важной и более общей *задачи о потоке минимальной стоимости*, по условиям которой мы присваиваем ребрам *стоимости* (еще одно множество весов ребер) и определяем стоимости потоков, когда ищем решение задачи о максимальном потоке, обладающем минимальной стоимостью. Мы рассмотрим классическое общее решение задачи о потоке минимальной стоимости, известное как *алгоритм вычеркивания циклов (cycle-canceling algorithm)*, а затем, в разделе 22.6, дадим конкретную реализацию алгоритма вычеркивания циклов, известную как *сетевой симплексный* алгоритм. В разделе 22.3 мы обсудим все сведения к задаче о потоке минимальной стоимости, которые включают, помимо прочих, все приложения, которые отмечались выше.

Алгоритмы решения задачи о потоках в сети представляют собой тему, подходящую для завершения данной книги, сразу по нескольким причинам. Упомянутая тема является как бы вознаграждением за усилия, потраченные на изучение базовых алгоритмических инструментальных средств, таких как связные списки, очереди с приоритетами и общие методы поиска на графах. Классы, осуществляющие обработку графов, которые изучались в этой книге, непосредственно приводят к компактным и эффективным реализаци-

ям классов для задач о потоках в сетях. Эти реализации поднимают нас на новый уровень возможностей решения задач и могут быть непосредственно использованы в многочисленных практических приложениях. Более того, изучение возможностей их применения и понимание свойственных им ограничений позволяют установить контекст для исследований, направленных на получение более совершенных алгоритмов решения еще более сложных задач, – именно этим вопросам посвящается часть 8.

22.1. Транспортные сети

Чтобы дать описание алгоритмов решения задач о потоках в сети, начнем с изучения идеализированной физической модели, в которой используются несколько интуитивных фундаментальных понятий. В частности, представим себе некоторую систему взаимосвязанных нефтепроводных труб различных диаметров, в которой на каждом пересечении труб установлены вентили, управляющие движением потоков нефти на каждом пересечении, как показано в примере на рис. 22.5. Далее мы предполагаем, что в трубопроводе присутствует единственный исток (допустим, это нефтяное месторождение) и единственный сток (скажем, нефтеперерабатывающий завод), с которым в конечном итоге все трубы связывают исток. В каждой вершине потоки нефти находятся в равновесии, т.е., объем поступающей в вершину нефти равен объему вытекающей из нее нефти. Мы будем измерять потоки и пропускную способность труб в одних и тех же единицах (например, в галлонах в секунду).

Если каждый вентиль обладает свойством, что пропускная способность входящих труб равна пропускной способности исходящих труб, то задача, как таковая, отсутствует: мы просто-напросто заполняем все трубы до упора. В противном случае заполненными оказываются не все трубы, однако, нефть течет по трубам, управляемая настройками вентилей, установленных на пересечениях труб, и эти настройки подобраны таким образом, что объем нефти, поступающей на каждое пересечение, равен объему вытекающей из него нефти. Однако из описанного логического равновесия на пересечениях следует необходимость равновесия сети

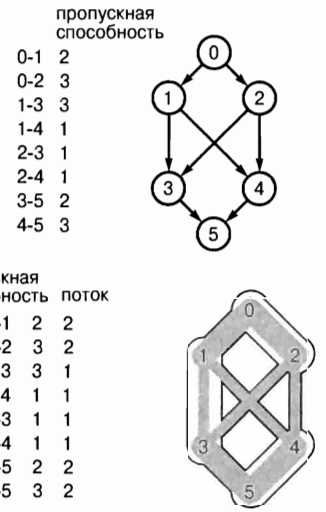


РИСУНОК 22.5. ПОТОКИ В СЕТИ

*Транспортная сеть есть взвешенная сеть, в которой мы интерпретируем веса ребер как пропускные способности (вверху). Наша цель заключается в том, чтобы вычислить второе множество весов ребер, ограниченное пропускными способностями, которые мы называем потоками. Нижняя диаграмма служит иллюстрацией наших соглашений, касающиеся вычерчивания сетей потоков. Ширина каждого ребра пропорциональна его пропускной способности; объем потока в каждом ребре показан в виде заштрихованной части; поток всегда направлен на странице сверху вниз из единственного истока вверху в единственный сток внизу; пересечения (такие как ребра **1-4** и **2-3** в рассматриваемом примере) не представляют вершин, если не обозначены таковыми. За исключением стока и истока, входной поток равен выходному потоку в каждой вершине· например, в вершине **2** имеются две единицы входного потока (из вершины **0**) и две единицы выходного потока (по одной единице в вершину **3** и вершину **4**).*

как единого целого: мы покажем при доказательстве свойства 22.1, что объем нефти, поступающей в сток, равен объему нефти, вытекающей из истока (в некоторых публикациях используется термин источник – *прим. перев.*). Более того, как следует из рис. 22.6, настройки вентилей, установленных на пересечениях, при таких объемах потоков из истока в сток оказывают нетривиальное влияние на поток через сеть. В свете указанных фактов мы заинтересованы в получении ответа на следующий вопрос: какие настройки вентилей обеспечат максимальный объем потоков нефти из истока в сток?

Мы можем прямо смоделировать эту ситуацию при помощи сети (взвешенный орграф, согласно определению из главы 21), которая обладает одним истоком и одним стоком. Ребра сети соответствуют трубам нефтепровода, вершины соответствуют пересечениям труб с установленными на них вентилями, которые регулируют, сколько нефти попадает в выходное ребро, а вес, присвоенный ребру, соответствует пропускной способности труб. Мы полагаем, что ребра ориентированы, и это обстоятельство отражает тот факт, что нефть может перемещаться только в одном направлении в каждой трубе. По каждой трубе проходит определенное число единиц потока, который меньше или равен ее пропускной способности, и каждая вершина удовлетворяет условию равновесия, суть которого заключается в том, что входной поток равен выходному потоку.

Абстракция транспортной сети представляет собой полезную модель решения задач, которая прямо применяется к широкому кругу приложений, а косвенно – к еще более широкому кругу приложений. Иногда мы обращаемся к представлению нефти, текущей по нефтепроводу, в стремлении отыскать интуитивную поддержку базовых идей, в то же время наши рассуждения одинаково применимы как к товарам, перемещающимся по распределительным каналам, так и к другим многочисленным ситуациям.

Модель потоков применяется непосредственно к сценарию распределения: мы интерпретируем размеры потоков как их интенсивность, так что транспортная сеть описывает потоки товаров точно так же, как и потоки нефти. Например, мы можем интерпретировать поток, представленный на рис. 22.5, как факт, определяющий необходимость переправки двух элементов в единицу времени из вершины **0** в **1** и из **0** в **2**, одного элемента в единицу времени из **0** в **2**, одного элемента в единицу времени из **1** в **3** и из **1** в **4** и т.д.

Другая интерпретация модели потоков для сценария распределения заключается в том, что потоки интерпретируются как объемы поставляемых товаров с той характерной особенностью, что сеть описывает только одноразовую поставку товара. Например, мы можем интерпретировать поток, представленный на рис. 22.5, как поставку четырех единиц товара из вершины **0** в **5** в рамках следующего трехэтапного процесса: сначала пересылаются две единицы товара из **0** в **1** и две единицы из **0** в **2**, так что в каждой из этих вершин оседают по две единицы товара. Далее производится пересылка по одной единице товара из **1** в **3**, из **1** в **4**, из **2** в **3** и из **2** в **4**, при этом в каждой из вершин **3** и **4** остается по две единицы товара. И, наконец, пересылка завершается доставкой двух единиц товара из вершины **3** в **5** и из **4** в **5**.

Как и в случае использования расстояния в алгоритмах поиска кратчайшего пути, мы вполне можем, когда это удобно, отказаться от любой физической интерпретации, поскольку все рассматриваемые нами определения, свойства и алгоритмы основаны исключительно на абстрактной модели, которая не обязательно подчиняется физическим зако-

нам. В самом деле, основная причина нашего интереса к модели потоков в сети заключается в том, что она позволяет решать множество других задач методом сведения, как в этом можно будет убедиться в разделах 22.4 и 22.6. В силу такой широкой применимости целесообразно дать точное определение терминов и понятий, которые были только что введены, однако без соблюдения надлежащих формальностей.

Определение 22.1. *Будем называть сеть с вершиной s, выбранной в качестве истока, и с вершиной t, выбранной в качестве стока*, **st-сетью**.

В этом определении мы используем понятие "выбранный", которое означает, что вершина *s* не обязательно должна быть истоком (вершина, у которой отсутствуют входящие ребра), а вершина *t* — стоком (вершина, у которой отсутствуют исходящие ребра), но, тем не менее, они рассматриваются именно в этом качестве, поскольку в наших рассуждениях (и в предлагаемых нами алгоритмах) игнорируются ребра, направленные в *s*, и ребра, исходящие из *t*. Во избежание путаницы, в примерах мы будем рассматривать сети с одним истоком и стоком; ситуацию более общего характера мы рассмотрим в разделе 22.4. Мы будем называть вершины *s* и *t*, соответственно, "истоком" и "стоком" *st*-сети, поскольку именно эти роли они исполняют в рассматриваемой сети. Остальные вершины сети мы будем называть *внутренними* вершинами.

Определение 22.2. Транспортная сеть (flow network) *есть st-сеть с положительными весами ребер, которые мы будем называть* **пропускными способностями (capacities)**. **Поток (flow)** *в транспортной сети есть множество ребер с неотрицательными весами, в дальнейшем* **реберными потоками (edge flows)**, *которые удовлетворяют требованию того, что ни один поток в ребре не может быть больше пропускной способности ребра и что суммарный поток, поступающий в каждую внутреннюю вершину, равен суммарному потоку, вытекающему из этой вершины.*

Мы будем называть суммарный поток, устремленный в некоторую вершину, *притоком (inflow)*

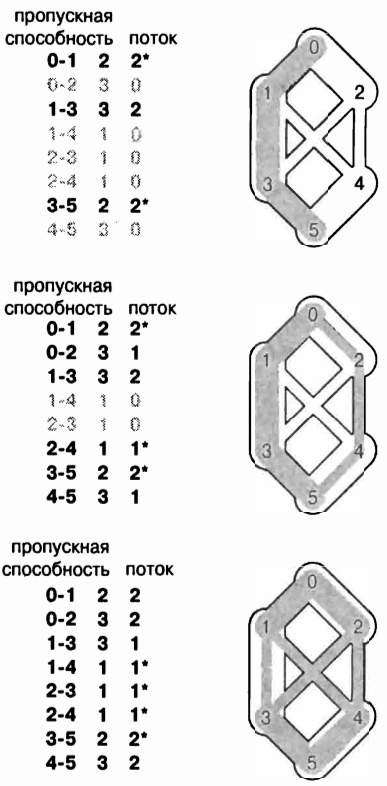


РИСУНОК 22.6. УПРАВЛЕНИЕ ПОТОКАМИ В СЕТИ

Открыв вентили вдоль пути 0-1-3-5, мы можем инициировать в этой сети поток, который может перемещать две единицы потока (диаграмма вверху); открыв вентили вдоль пути 0-2-4-5, мы получаем еще одну единицу потока в сети (диаграмма в центре). Звездочками отмечены заполненные ребра.

Поскольку ребра 0-1, 2-4 и 3-5 наполнены, прямого способа увеличить поток из 0 в 5 не существует, но если мы откроем вентиль в вершине 1, чтобы перенаправить достаточную часть потока с тем, чтобы заполнить ребро 1-4, мы увеличиваем пропускную способность ребра 3-5, которой достаточно для увеличения потока на пути 0-2-3-5, благодаря чему достигается максимальный поток рассматриваемой сети (диаграмма внизу).

этой вершины, и суммарный поток, вытекающий из той или иной вершины, — *истечением (outflow)* этой вершины. По соглашению мы устанавливаем потоки в ребрах, входящих в исток, и потоки в ребрах, исходящих из стока, равными нулю, а при доказательстве свойства 22.1 покажем, что истечение истока всегда равно притоку стока, этот поток мы будем называть *величиной (value)* сети. Имея в своем распоряжении эти определения, нетрудно дать формальное определение нашей главной задачи.

Максимальный поток. Пусть задана *st*-сеть, найти такой поток, что никакой другой поток из *s* в *t* не обладает большей мощностью. Будем называть такой поток *максимальным потоком (maxflow)*, а задачу построения такого потока в сети будем называть *задачей о максимальном потоке*. В некоторых приложениях нам достаточно знать только конкретную величину максимального потока, но общем случае мы хотим знать структуру потока (величины каждого реберного потока), обеспечивающих получение такой величины.

На ум немедленно приходят различные варианты этой задачи. Можем ли мы рассматривать сеть с несколькими истоками и стоками? Следует ли изучать сети без истоков и стоков? Можем ли мы допустить оба направления потока в ребрах? Можем ли мы накладывать ограничения на пропускную способность вершин вместо или в дополнение к ограничениям, наложенным на ребра? Для алгоритмов на графах характерно то, что отделение легко учитываемых ограничений от ограничений, влекущих далеко идущие последствия, само по себе может оказаться трудно решаемой задачей. Мы проведем исследование этой задачи и дадим примеры ее сведения к задаче о максимальном потоке множества различных задач, которые на первый взгляд имеют между собой мало общего, в разделах 22.2 и 22.3, после того, как мы рассмотрим алгоритмы решения фундаментальной задачи.

Характерным свойством потоков является условие локального равновесия, когда приток равен истечению в каждой внутренней вершине. На пропускные способности ребер такое ограничение не накладывается, в самом деле, именно нарушение баланса между суммарной пропускной способностью входящих ребер и суммарной пропускной способностью исходящих ребер является сутью задачи о максимальном потоке. Условие равновесия должно выполняться в каждой внутренней вершине, оказывается, что это локальное свойство определяет суммарное перемещение материала в сети. Однако это всего лишь гипотеза, которую нужно доказать.

Свойство 22.1. *Любой st-поток обладает тем свойством, что истечение вершины s равно притоку в вершину t.*

Доказательство: (В своих рассуждениях мы будем применять термин *st-поток (st-flow)*, который означает "поток в *st*-сети".) Добавим в сеть ребро, ведущее из фиктивной вершины в вершину *s*, с потоком и пропускной способностью, равными истечению вершины *s*, и ребро, ведущее из вершины *t* в другую фиктивную вершину, с потоком и пропускной способностью, равными втеканию вершины *t*. Теперь по индукции мы можем доказать более общее свойство: приток равен истечению для любого *множества* вершин (фиктивные вершины не учитываются).

По условию локального равновесия это свойство характерно для любой одиночной вершины. Предположим теперь, что это свойство выполняется для заданного набора вершин *S* и что мы добавили в это множество одиночную вершину *v*, так что теперь

имеем множество $S' = S \cup \{v\}$. При вычислении притока и истечения для S', обратите внимание на то обстоятельство, что каждое ребро, ведущее из вершины v в некоторую вершину множества S, уменьшает истечение (из вершины v) на ту же величину, на какую оно уменьшает приток (в S); каждое ребро, ведущее в вершину v из некоторой вершины множества S, уменьшает приток (в вершину v) на ту же величину, на какую уменьшает истечение (из S); все другие ребра обеспечивают приток или истечение из множества S' тогда и только тогда, когда они делают то же самое для S или для v. Таким образом, приток равен истечению для S', а величина потока равна сумме величин потоков S и v, минус сумма потоков в ребрах, соединяющих v с некоторой вершиной в S (в любом направлении).

Применяя это свойство к множеству всех вершин, мы находим, что приток в исток из связанной с ней фиктивной вершины (который равен истечению истока) равен истечению стока в связанную с ним фиктивную вершину (которое равно притоку стока). ■

Следствие. *Величина потока объединения двух множеств вершин равна сумме величин каждого из потоков этих двух множеств минус сумма весов ребер, соединяющих вершину одного множества с какой-либо вершиной другого множества.*

Доказательство: Приведенное выше доказательство для множества S и вершины v работает и в том случае, когда мы заменяем вершину v некоторым множеством T (которое не пересекается с S). Иллюстрацией этого свойства может служить пример, показанный на рис. 22.7. ■

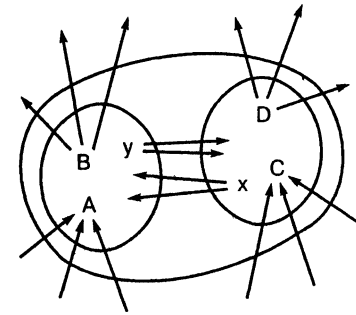


РИСУНОК 22.7. РАВНОВЕСИЕ ПОТОКОВ

Эта диаграмма служит иллюстрацией сохранения равновесия потоков при объединении множеств вершин. Две фигуры меньших размеров соответствуют двум произвольным непересекающимся множествам вершин, а буквы представляют множества ребер: A есть количество потока, втекавшего в множество слева извне относительно множества, изображенного справа, x есть количество потока, втекающего в множество слева из множества справа и так далее. Теперь, если установлено равновесие между двумя этими потоками, должно выполняться равенство

$$A + x = B + y$$

для множества слева и

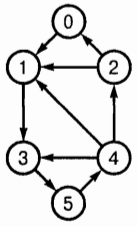
$$C + y = D + x$$

для множества справа. Суммируя эти два равенства и исключая сумму $x + y$, мы приходим к заключению, что

$$A + C = B + D,$$

или приток равен истечению для объединения двух множеств.

Мы можем обойтись и без фиктивных вершин при доказательстве свойства 22.1, расширить любую транспортную сеть за счет включения ребра из t в s с пропускной способностью, равной мощности сети, и знать, что приток равен истечению для любого множества узлов расширенной сети. Такой поток называется *циркуляцией (circulation)*, и эта конструкция показывает, что задача о максимальном потоке сводится к задаче отыскания такой циркуляции, которая обеспечивает максимальный поток в заданном ребре. Такая формулировка упрощает наши рассуждения в некоторых ситуациях. Например, как показывает рис. 22.8, она приводит к интересному альтернативному представлению потоков как множества циклов.



0-1	1		0-1	1			
1-3	3		1-3	1	1-3	1	
2-0	1		2-0	1			
2-1	1	1-3	2		2-1	1	
3-5	7			3-5	1	3-5	1
4-1	2	3-5	2				3-5 3
4-2	2	4-1	2	4-2	1	4-2	1
4-3	3						4-3 3
5-4	7	5-4	2	5-4	1	5-4	1 5-4 3
		1-3-5-4-1		0-1-3-5-4-2-0		1-3-5-4-2-1	3-5-4-3

РИСУНОК 22.8. ПРЕДСТАВЛЕНИЕ В ВИДЕ ЦИКЛИЧЕСКИХ ПОТОКОВ

*Эта диаграмма показывает, что циркуляция слева разбивается на четыре цикла **1-3-5-4-1**, **0-1-3-5-4-2-0**, **1-3-5-4-2-1**, **3-5-4-3** с весами, соответственно **2**, **1**, **1** и **3**. Каждое ребро каждого цикла появляется в соответствующем столбце, а суммирование весов каждого ребра в каждом цикле, в котором оно появляется (по соответствующей строке), дает его вес в циркуляции.*

Если задано множество циклов и величин потоков для каждого цикла, то легко вычислить соответствующую циркуляцию путем обхода каждого цикла и добавления указанной величины потока в каждое ребро. Обратное свойство более интересно: мы можем найти множество циклов (и величину потока для каждого из них), которое эквивалентно любой заданной циркуляции.

Свойство 22.2. (Теорема декомпозиции потоков). *Любая циркуляция может быть представлена как поток в некотором множестве из максимум E направленных циклов.*

Доказательство: Этот результат устанавливает простой алгоритм. Будем повторять следующий процесс до тех пор, пока имеются ребра, по которым проходят потоки: отправляясь от произвольного ребра, в котором протекает поток, проследуем вдоль любого ребра, исходящего из вершины назначения первого ребра и в свою очередь содержащего поток, и продолжаем эту процедуру до тех пор, пока не попадем в вершину, которую мы уже посетили (был обнаружен цикл). Выполним обратный обход обнаруженного цикла с тем, чтобы найти ребро с минимальным потоком; затем уменьшим потоки в каждом ребре цикла на это значение. Каждая итерация этого процесса уменьшает поток до 0, по меньшей мере, в одном ребре, следовательно, существуют, по меньшей мере, E циклов. ■

Рисунок 22.9 иллюстрирует процесс, описанный в доказательстве. В случае *st*-потоков, применение этого свойства к циркуляции, полученной путем добавления ребра из s в t, позволяет сделать вывод о том, что любой *st*-поток может быть представлен как поток вдоль некоторого множества, состоящего из максимум E ориентированных путей, каждый из которых есть либо путь из s в t, либо цикл.

Следствие. *Любая st-сеть обладает максимальным потоком, подграф которого, индуцированный потоками ненулевой величины, есть ациклический граф.*

Доказательство: Циклы, которые не содержат ребра **t-s**, не меняют величины потока в сети, и в силу этого обстоятельства мы можем присвоить потоку в любом таком цикле величину 0, не меняя величины потока в сети. ■

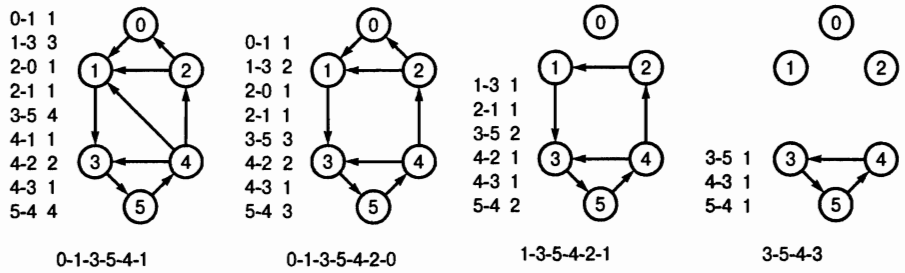


РИСУНОК 22.9. ПРОЦЕСС ДЕКОМПОЗИЦИИ ПОТОКОВ ЦИКЛА

*Чтобы разложить произвольную циркуляцию на множество циклов, мы многократно выполняем следующий процесс: следуем вдоль некоторого пути до тех пор, пока не встретим какой-либо узел второй раз, затем отыскиваем минимальный вес в обнаруженном цикле, после чего вычитаем минимальный вес из весов каждого ребра обнаруженного цикла и удаляем каждое ребро, вес которого получил значение 0. Например, на первой итерации мы производим обход вдоль пути **0-1-3-5-4-1**, на котором обнаруживаем цикл **1-3-5-4-1**, далее мы вычитаем 1 из весов всех ребер цикла, по результатам этого вычитания мы должны удалить из цикла ребро **4-1**, так как его вес принимает значение 0. На второй итерации мы удаляем ребра **1-3**, **4-2** и **2-1**, а на последней итерации удаляем **3-5**, **5-4** и **4-3**.*

Следствие. *Любая st-сеть обладает максимальным потоком, который может быть представлен как поток вдоль множества максимум E ориентированных путей из s в t.*

Доказательство: Непосредственно следует из выше сказанного. ■

Все сказанное позволяет получить представление о внутренней структуре потоков, которое будет весьма полезно при разработке и анализе алгоритмов отыскания максимальных потоков.

С одной стороны, мы можем получить более общую формулировку задачи отыскания максимального потока в сети со многими истоками и стоками. При таком допущении рассматриваемые нами алгоритмы могут быть использованы в более широком диапазоне приложений. С другой стороны, мы можем изучать специальные случаи, например, ограничиться изучением ациклических цепей. Такой подход может облегчить решение рассматриваемой задачи. В самом деле, как мы убедимся в разделе 22.4, по трудности решения эти варианты эквивалентны рассматриваемой нами версии. Поэтому в первом случае мы можем приспособить интересующие нас алгоритмы и приложения для их применения в более широкой области приложений, во втором случае мы едва ли можем надеяться найти более легкое решение. В приводимых нами иллюстрациях мы пользуемся ациклическими графами, поскольку такие примеры легче понять, когда направление потока подразумевается (сверху вниз на странице), но построенные нами реализации допускают существование сетей с циклами.

Чтобы реализовать алгоритм отыскания максимального потока, мы воспользуемся классом **GRAPH** из главы 20, организовав указатели на более сложный класс **EDGE**. Вместо одного веса, что делалось в главах 20 и 21, мы используем приватные элементы данных **pcap** и **pflow** (а также общедоступные функции-элементы **cap()** и **flow()**, которые возвращают значения этих элементов данных), соответственно, для пропускной способности и потока. Несмотря на то что сети являются ориентированными графами, рассматриваемые нами алгоритмы требуют обхода ребер в обоих направлениях, поэтому мы используем представление в виде *неориентированного* графа, которое применялось в главе 20, и функцию-элемент **from**, которая отличает ребро **u-v** от ребра **v-u**.

Такой подход позволяет отделить абстракцию, необходимую рассматриваемым нами алгоритмам (ребра идут в обоих направлениях), от конкретной структуры данных клиента, и ставит перед этими алгоритмами простую цель: присвоить элементам данных **flow** в ребрах клиента такие значения, которые обеспечивают максимизацию потока через сеть. В самом деле, критическая компонента в нашей реализации требует замены абстракции сети, которая зависит от значений потока и реализуется посредством функций-элементов класса **EDGE**. Мы будем рассматривать реализацию функций класса **EDGE** (программа 22.2) в разделе 22.2.

Поскольку транспортные сети, как правило, разрежены, мы воспользуемся классом **GRAPH**, в основу которого положено представление графа в виде списка смежных вершин, подобное реализации класса **SparceMultiGRAPH** (разреженный мультиграф) из программы 20.5. Что более важно, типовые транспортные сети могут иметь параллельные ребра (различной пропускной способности), соединяющие две вершины. Такая ситуация не требует специальных мер с применением класса **SparceMultiGRAPH**, тем не менее, в случае представления графа в виде матрицы смежности, клиенты должны объединить все параллельные ребра в одно ребро.

В сетях, представленных в главах 20 и 21, мы пришли к соглашению о том, что веса представляются вещественными числами в диапазоне от 0 до 1. В этой главе мы полагаем, что веса (пропускная способность и величина потоков) суть m-разрядные целые числа (в промежутке от 0 до 2^{m-1}). Это мы делаем, по меньшей мере, по двум причинам. Во-первых, нам довольно часто придется проверять на равенство различные комбинации весов, и делать это в представлении соответствующих величин в виде вещественных чисел с плавающей точкой может оказаться неудобным. Во-вторых, время выполнения рассматриваемых нами алгоритмов может зависеть от относительных значений весов, а параметр $M = 2^m$ дает удобный способ ограничения значений весов. Например, отношение наибольшего веса к наименьшему ненулевому весу принимает значение, меньшее M. Использование целочисленных весов есть одна из многочисленных возможных альтернатив (см., например, упражнение 20.8), которые мы можем выбрать при решении такого рода задач.

Иногда нам приходится рассматривать ребра *с неограниченной пропускной способностью.* Это может означать, что мы не соотносим поток с пропускной способностью такого ребра, либо мы можем воспользоваться сигнальным значением, которое принимает заведомо большее значение, чем величина любого потока.

Программа 22.1 представляет собой клиентскую функцию, которая проверяет, удовлетворяют ли потоки условию равновесия в каждом узле и возвращает значение величины потока, если поток удовлетворяет этому условию. В большинстве случаев мы включаем вызов этой функции в алгоритм вычисления максимального потока в качестве заключительной операции. Несмотря на все доверие, которое мы испытываем к свойству 22.1, характерная для всех программистов параноидная подозрительность нашептывает нам, что мы должны также проверить, равен ли поток, истекающий из истока, потоку, втекающему в сток. Возможно, мы благоразумно поступим, если проверим, что ни в одном ребре поток не превосходит по величине пропускную способность этого ребра и что структуры данных внутренне непротиворечивы (см. упражнение 22.12).

Программа 22.1. Проверка потока и вычисление мощности потока

За счет обращения к функции **flow(G, v)** вычисляется разность между втекающим и вытекающим потоками в вершине **v** сети **G**. Посредством вызова **flow(G, s, t)** проверяются величины потоков сети из истока **(s)** в сток **(t)**, при этом 0 возвращается в тех случаях, когда втекающий поток в некотором внутреннем узле не равен вытекающему потоку или если величина какого-либо потока принимает отрицательное значение; в противном случае возвращается величина потока.

```
template <class Graph, class Edge> class check
{
public:
  static int flow(Graph &G, int v)
    { int x = 0;
      typename Graph::adjIterator A(G, v);
      for (Edge* e = A.beg(); !A.end(); e = A.nxt())
        x += e->from(v) ? e->flow() : -e->flow();
      return x;
    }
  static bool flow(Graph &G, int s, int t)
    {
      for (int v = 0; v < G.V(); v++)
        if ((v != s) && (v != t))
          if (flow(G, v) != 0) return false;
      int sflow = flow(G, s);
      if (sflow < 0) return false;
      if (sflow + flow(G, t) != 0) return false;
      return true;
    }
};
```

Упражнения

▷ **22.1**. Найдите два различных максимальных потока в транспортной сети, изображенной на рис. 22.10.

22.2. В предположении, что все пропускные способности суть положительные целые числа, меньшие M, каким может быть максимально возможный поток для произвольной *st*-сети с V вершинами и E ребрами? Дайте два ответа, в зависимости от того, допускаются ли параллельные ребра.

▷ **22.3**. Предложите алгоритм решения задачи о максимальном потоке для случая, когда сеть образует дерево, если при этом убрать сток.

○ **22.4**. Найдите семейство сетей с E ребрами, обладающих циркуляцией, в котором процесс, описанный в доказательстве 22.2, порождает E циклов.

пропускная способность
0-1 2
0-2 3
0-3 2
1-2 1
1-3 1
1-4 1
2-4 1
2-5 2
3-4 2
3-5 3
4-5 2

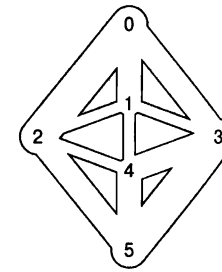


РИСУНОК 22.10. ТРАНСПОРТНАЯ СЕТЬ ДЛЯ УПРАЖНЕНИЙ

Представленная на этом рисунке транспортная сеть является предметом исследований в нескольких упражнениях в данной главе.

22.5. Разработайте класс **EDGE**, представляющий пропускные способности и потоки в виде вещественных чисел в промежутке от 0 до 1, выраженных d цифрами после десятичной точки, где d есть фиксированная константа.

▷ **22.6**. Напишите программу, которая строит транспортную сеть путем считывания из стандартного ввода ребер (пар целых чисел в интервале от 0 до $V-1$) с целочисленными значениями пропускных способностей. При этом предполагается, что верхняя граница пропускной способности M не превосходит значения 2^{20}.

22.7. Распространите полученное вами решение упражнения 22.6 на использование символических имен вместо чисел при обращении к вершинам (см. программу 17.10).

▷ **22.8**. Отыщите пример крупной транспортной сети, которую можно было бы использовать как среду для тестирования алгоритмов обработки потоков на реалистичных данных. Возможны варианты транспортных сетей (автомобильные перевозки, железнодорожный транспорт, авиация), сетей связи (телефонные и компьютерные сети передачи данных) или распределительных сетей. Если неизвестны пропускные способности, придумайте подходящую модель, назначающую пропускные способности каналов. Напишите программу, использующую интерфейс программы 22.2 для реализации транспортных сетей с вашими данными, возможно, с применением решения упражнения 22.7. При необходимости, разработайте дополнительные приватные функции для удаления данных, в соответствии с описанием упражнений 17.33–17.35.

22.9. Напишите программу генератора случайных сетей для разреженных сетей с пропускными способностями, принимающими значение в диапазоне от 0 до 2^{20}, на основании программы 17.7. Воспользуйтесь специальными классами для пропускных способностей и разработайте две реализации, одна из которых генерирует равномерно распределенные пропускные способности, а другая генерирует пропускные способности в соответствии с распределением Гаусса. Разработайте клиентские программы, которые генерируют случайные сети для обоих типов распределений весов с тщательно подобранным заданными значениями V и ребер E, чтобы можно было использовать их для прогона эмпирических тестов на графах, полученных на основе различных распределений весов ребер.

22.10. Напишите программу генератора случайных сетей для насыщенных сетей с пропускными способностями ребер в диапазоне от 0 до 2^{20} на базе программы 17.8 и генераторов пропускных способностей ребер, описанных в упражнении 22.9. Напишите клиентские программы генерации случайных сетей для обоих распределения весов с тщательно подобранными заданными значениями V и ребер E, чтобы можно было использовать их для прогона эмпирических тестов на графах, построенных на базе этих моделей.

• **22.11**. Напишите программу, которая генерирует на плоскости V случайных точек, затем постройте транспортную сеть с ребрами (в обоих направлениях), соединяющими все пары точек, расположенных на расстоянии, не превышающем d одна от другой (см. программу 3.20), устанавливая пропускную способность каждого ребра с помощью одной из случайных моделей, описанных в упражнении 22.9. Определите, как выбрать d таким, чтобы ожидаемое число ребер было равным E.

▷ **22.12**. Внесите изменения в программу 22.1, которые позволили бы ей для каждого ребра выполнять проверку, что величина потока меньше его пропускной способности.

▷ **22.13**. Найти все максимальные потоки в сети, показанной на рис. 22.11. Дайте представление каждого из них в виде цикла.

22.14. Разработайте функцию, которая считывает величины потоков и циклы (по одному в строке, в формате, изображенном на рис. 22.8) и строит сеть, имеющую соответствующий поток.

22.15. Разработайте клиентскую функцию, которая находит представление потока сети в виде циклов, используя метод, описанный в доказательстве свойства 22.2, и распечатывает величины потоков и циклы (по одному в строке, в формате, представленном на рис. 22.8).

○ **22.16**. Разработайте функцию, которая удаляет циклы из *st*-потока сети.

○ **22.17**. Напишите программу, которая назначает целочисленные потоки каждому ребру в любом заданном орграфе, который не содержит ни истоков, ни стоков, так, что этот орграф является транспортной сетью, представляющей собой циркуляцию.

пропускная способность	
0-1	2
0-2	3
0-3	2
1-3	1
1-4	1
2-1	1
2-5	2
3-4	2
3-5	3
4-2	1
4-5	2

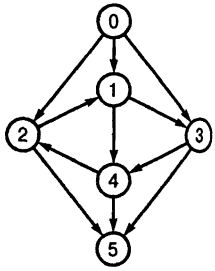


РИСУНОК 22.11. ТРАНСПОРТНАЯ СЕТЬ С ЦИКЛАМИ

Представленная на этом рисунке транспортная сеть подобна сети, изображенной на рис. 22.10, за исключением того, что направление двух ребер заменены на обратные, вследствие чего появляются два цикла. Эта транспортная сеть также является предметом исследований в нескольких упражнениях в данной главе.

○ **22.18**. Предположим, что поток представляет собой товары, которые перевозят на грузовых машинах между городами, при этом под потоком в ребре **u-v** понимается количество товара, которое надлежит доставить из города **u** в город **v** в течение дня. Напишите клиентскую функцию, которая распечатывает распорядок дня для водителей грузовиков, уведомляя их о том, где что взять, в каком количестве и в каком месте разгрузиться. Предполагается, что не существует предельных значений загрузки грузовиков, и ни одно транспортное средство не покинет заданный распределительный пункт, пока на него не поступит весь товар.

22.2. Алгоритм поиска максимального потока методом аугментального пути

Эффективный подход к решению задачи о максимальном потоке был разработан Л.Р. Фордом (L.R. Ford) и Д.Р. Фалкерсоном (D.R. Fulkerson) в 1962 г. Это обобщенный метод предусматривает инкрементное увеличения потоков вдоль пути от истока к стоку, который является основой для целого семейства алгоритмов. В классической литературе он известен под названием *метода Форда-Фалкерсона (Ford-Fulkerson method)*; широкое распространение также получил более образный термин, *метод аугментального пути (augmenting path method)*.

Рассмотрим произвольный ориентированный путь (не обязательно простой) из истока в сток в *st*-сети. Пусть x есть минимальное значение неиспользованной пропускной способности ребер на этом пути. Мы можем повысить величину потока в сети, по меньшей мере, на величину x, увеличив поток во всех ребрах, составляющих указанный путь, на эту величину. Неоднократно повторяя это действие, мы осуществляем первую попыт-

ку вычисления потока в сети: находим другой путь, наращиваем поток вдоль этого пути и продолжаем эту процедуру до тех пор, пока каждый из таких путей, ведущих из истока в сток, не получит, по меньшей мере, одно заполненное ребро (так что больше мы не сможем наращивать потоки подобным способом). В одних случаях такой алгоритм вычисляет максимальный поток, в других случаях это ему не удается. На рис. 22.6. показан случай, когда этому алгоритму не удается вычислить максимальный поток.

Чтобы усовершенствовать рассматриваемый алгоритм таким образом, чтобы он всегда находил максимальный поток, мы рассмотрим более общий способ наращивания потока вдоль произвольного пути, ведущего из истока в сток в неориентированном графе, положенном в основу сети. Ребра такого пути относятся либо к категории *прямых (forward)* ребер, направления которых совпадают с направлением потока (когда мы следуем по пути из истока в сток, мы идем по ребру из его начальной вершины в его конечную вершину), либо к категории *обратных (backward)* ребер, которые направлены против потока (при общем направлении из истока в сток, мы идем вдоль ребра из его конечной вершины в его начальную вершину). Теперь мы имеем возможность нарастить поток в сети по любому пути за счет увеличения потоков в прямых ребрах и *уменьшения* потоков в обратных ребрах. Величина, на которую поток может быть увеличен, ограничивается минимальной неиспользованной пропускной способностью и потоком в обратных ребрах. На рис. 22.12 показан соответствующий пример. В новом потоке, по меньшей мере, одно из прямых ребер, включенных в путь, заполняется, либо одно из обратных ребер, включенных в путь, становится пустым.

Описанный выше процесс служит основой классического алгоритма Форда-Фалкерсона вычисления максимального потока (алгоритм аугментального пути). Сформулируем его следующим образом:

> Начните выполнение алгоритма в любом месте сети. Наращивайте поток вдоль произвольного пути из истока в сток, содержащего незаполненные ребра или пустые обратные ребра, продолжая этот процедуру до тех пор, пока не останется в сети ни одного такого пути.

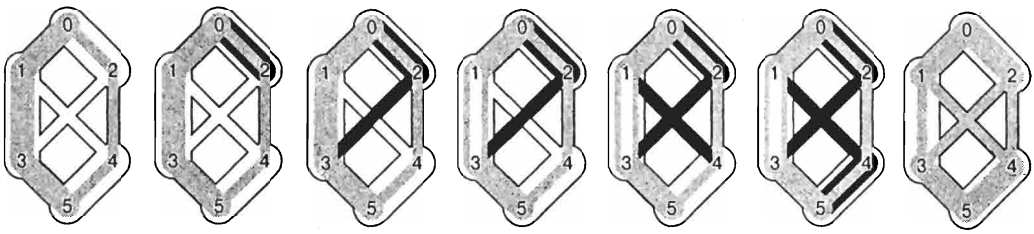


РИСУНОК 22.12. НАРАЩИВАНИЕ ПОТОКА ВДОЛЬ ПРОИЗВОЛЬНОГО ПУТИ

Представленная на рисунке последовательность диаграмм служит иллюстрацией наращивания потока в сети вдоль пути, включающего как прямые, так и обратные ребра. Отправляясь от потока, изображенного на левой диаграмме и выполняя операции слева направо, мы сначала наращиваем поток в ребре ***0-2****, а затем в ребре* ***2-3*** *(дополнительные потоки показаны черным цветом). Далее мы уменьшаем поток в ребре* ***1-3*** *(показан белым цветом) и отводим извлеченную часть этого потока в ребро* ***1-4****, а затем в* ***4-5****, в результате чего получаем поток, показанный на правой диаграмме.*

Как ни странно, но этот метод всегда находит максимальный поток независимо от способа выбора путей. Подобно методу отыскания дерева MST, обсуждавшегося в разделе 20.1, и методу Беллмана-Форда поиска кратчайшего пути, обсуждавшегося в разделе 21.7, это общий алгоритм, который полезен тем, что постулирует правильность всего семейства более специализированных алгоритмов. Мы свободны в выборе любого из этих методов, как бы мы не выбирали путь.

Рисунок 22.13 служит иллюстрацией нескольких различных последовательностей аугментальных путей, каждый из которых позволяет найти максимальный поток в сети, предложенной в качестве примера. Далее в этом разделе мы рассмотрим несколько алгоритмов, которые вычисляют последовательности аугментальных путей; каждый из которых приводит к нахождению максимального пути. Эти алгоритмы различаются по числу аугментальных путей, которые они вычисляют, по длине вычисляемых путей, однако все они реализуют алгоритм Форда-Фалкерсона и находят максимальный поток.

Чтобы показать, что любой поток, вычисленный с помощью любой реализации алгоритма Форда-Фалкерсона, и в самом деле имеет максимальную величину, мы покажем, что этот факт эквивалентен ключевому условию, известному как *теорема о максимальных потоках и минимальных сечениях (maxflow-mincut theorem)*. Понимание этой теоремы представляет собой очень важный шаг к пониманию алгоритмов на транспортной сети. Как следует из названия, теорема зиждется на непосредственной зависимости между потока-

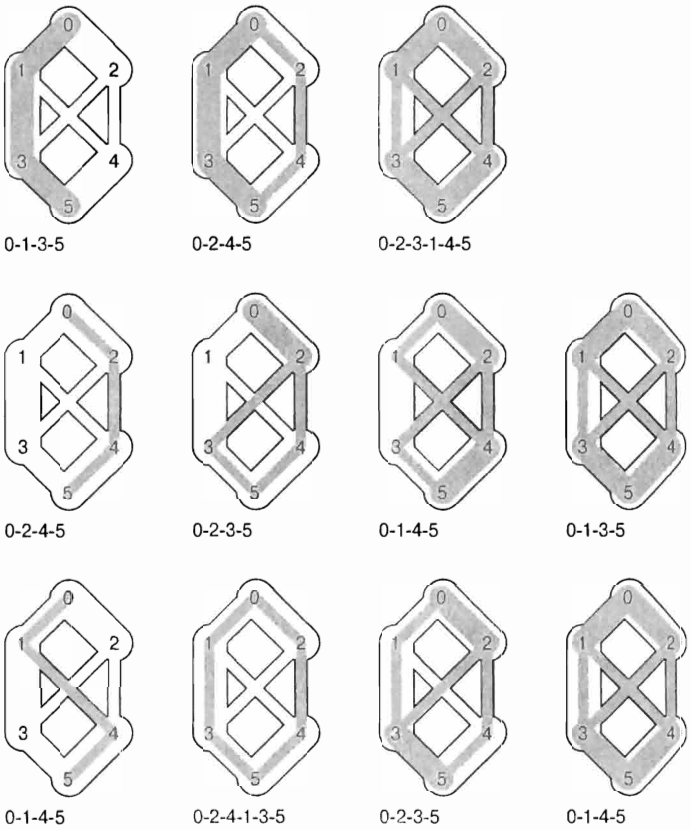


РИСУНОК 22.13. ПОСЛЕДОВАТЕЛЬНОСТИ АУГМЕНТАЛЬНЫХ ПУТЕЙ

В трех представленных на рисунке примерах мы производим наращивание потока вдоль различных последовательностей аугментальных путей до тех пор, пока станет невозможно найти новые аугментальные пути. Поток, который удается получить в каждом случае, есть максимальный поток. Классическая теорема из теории сетевых потоков утверждает, что мы получаем максимальный поток в любой сети, независимо от того, какую последовательность путей мы используем (см. свойство 22.5).

ми и *сечениями* в сетях, поэтому мы начнем с определения терминов, имеющих отношение к сечениям.

Напомним, что в соответствии с определением, данным в разделе 20.1, *сечение (cut)* графа есть разбиение множества вершин графа на два непересекающихся подмножества, а *пересекающее ребро (crossing edge)* есть ребро, соединяющее вершину некоторую одного подмножества с вершиной другого подмножества. Применительно к транспортным сетям мы уточним эти определения следующим образом (см. рис. 22.14).

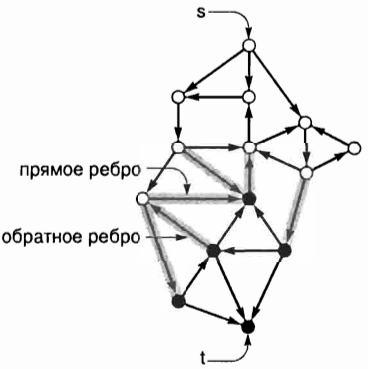


РИСУНОК 22.14. ТЕРМИНОЛОГИЯ, ИСПОЛЬЗУЕМАЯ ПРИ ОПИСАНИИ *st*-СЕЧЕНИЯ

В рассматриваемой st-сети имеется один исток s и один сток t. st-сечение есть разбиение вершин на множество, содержащее вершину s (белые кружки), и множество, содержащее вершину t (черные кружки). Ребра, связывающие вершины одного множества с вершинами другого множества (выделенные серым цветом), представляют собой разделяющее множество. Прямое ребро ведет из вершины множества, содержащего s, в вершину множества, содержащего t, обратное ребро ведет в другом направлении. В показанное на этой диаграмме сечение входят четыре прямых ребра и два обратных ребра.

Определение 22.3. *st-сечение есть сечение, которое помещает вершину s в одно из своих множества, а вершину t — в другое множество.*

Каждое пересекающее ребро, соответствующее *st*-сечению, есть либо *st*-ребро, ведущее из вершины, которая принадлежит множеству, содержащему вершину *s*, в вершину, входящую во множество, содержащую вершину *t*, либо *ts*-ребро, которое ведет в обратном направлении. Иногда мы называем множество пересекающих ребер **разделяющим множеством (cut set)**. **Пропускная способность** *st*-сечения в транспортной сети есть сумма пропускных способностей ребер этого *st*-сечения, а **поток через** *st*-сечение есть разность между суммой потоков в этих *st*-ребрах и суммой потоков в *ts*-ребрах этого сечения.

Удаление разделяющего множества приводит к делению связного графа на две связных компоненты, при этом отсутствуют пути, соединяющие какую-либо вершину одного множества с вершиной, принадлежащей другому множеству. С удалением всех ребер, входящих в *st*-сечение в сети не остается путей, соединяющих вершины *s* и *t* в неориентированном графе, составляющем основу сети, но если возвратить хотя бы одно такое ребро, то такой путь может возродиться.

Сечения являются удобным видом абстракций для приложений, упомянутых в начале текущей главы, где транспортная сеть служила описанием движения боеприпасов с базы в войска на передовой. Чтобы полностью прервать снабжение войск самым экономичным образом, противник должен решить следующую задачу.

Минимальное сечение. Пусть задана *st*-сеть, найти *st*-сечение, такое, что пропускная способность никакого другого сечения не меньше искомого. Для краткости мы будем называть такое сечение *минимальным сечением (mincut)*, а задачу вычисления такого сечения в той или иной сети — *задачей о минимальном сечении (mincut problem)*.

Задача о минимальном сечении есть обобщение задач о связности графов, краткий анализ которой был проведен в разделе 18.6. Мы подробно рассмотрим соответствующее отношение в разделе 22.4.

В формулировке задачи о минимальном сечении нет упоминания о потоках, и может показаться, что эти определения не находятся в русле наших обсуждений алгоритма вычисления аугментальных путей. На первый взгляд, вычисление минимального сечения (множества ребер) кажется более легким занятием, чем вычисление максимального пути (назначение весов всем ребрам). Ничего подобного, ключевым фактом этой главы является то, что задачи о максимальном потоке и задачи о минимальном сечении имеют тесную внутреннюю связь. Сам по себе метод построения аугментальных путей в сочетании с двумя фактами, касающимися потоков и сечений, служит тому доказательством.

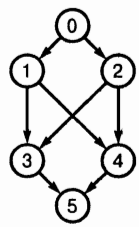
Свойство 22.3. *Для любого st-потока поток через каждый st-сечение равен величине этого потока.*

Доказательство: Это свойство является прямым следствием обобщения свойства 22.1, которое мы обсуждали в рамках соответствующего доказательства (см. рис. 22.7). Добавим ребро **t-s**, поток в котором равен такой величине, при которой приток равен истечению для любого множества вершин. Далее, для любого *st*-сечения, для которого C_s есть набор вершин, содержащий вершину *s*, а C_t есть набор вершин, содержащий вершину *t*, приток в C_s есть приток в вершину s (величина потока) плюс сумма потоков в обратных ребрах, пересекающих сечение; истечение из C_s равно сумме потоков в прямых ребрах, проходящих через сечение. Равенство этих двух количественных величин дает искомый результат. ■

Свойство 22.4. *Величина st-потока не может превышать пропускной способности никакого st-сечения.*

Доказательство: Вполне понятно, что поперечный поток через сечение не может превзойти пропускную способность этого сечения, так что результат непосредственно следует из свойства 22.3. ■

	пропускная способность	поток
0-1	2	2
0-2	3	2
1-3	3	1
1-4	1	1
2-3	1	1
2-4	1	1
3-5	2	2
4-5	3	2



0	1 2 3 4 5	0-1 0-2	1-3 2-3	5
0 1	2 3 4 5	0-2 1-3 1-4	1-4 2-4	7
0 2	1 3 4 5	0-1 2-3 2-4		4
0 3	1 2 4 5	0-1 0-2 3-5	2-3	6
0 4	1 2 3 5	0-1 0-2 4-5	2-4	8
0 1 2	3 4 5	1-3 1-4 2-3 2-4	1-3	6
0 1 3	2 4 5	0-2 1-4 3-5	1-4	6
0 1 4	2 3 5	0-2 1-3 4-5	1-3 1-4 2-3 2-4	9
0 2 3	1 4 5	0-1 2-4 3-5		5
0 2 4	1 3 5	0-1 2-3 4-5		6
0 3 4	1 2 5	0-1 0-2	1-3 1-4	5
0 1 2 3	4 5	1-4 2-4 3-5		4
0 1 2 4	3 5	1-3 4-5		6
0 2 3 4	1 5	0-1 3-5 4-5		7
0 1 2 3 4	5	3-5 4-5		5

РИСУНОК 22.15. ВСЕ *ST*-СЕЧЕНИЯ

В рассматриваемом списке для сети, изображенной слева, представлены все st-сечения, вершины множества, содержащего вершину s, вершины множества, содержащего вершину t, прямые ребра, обратные ребра и пропускная способность (сумма пропускных способностей прямых ребер). Для любого потока поток через все сечения (потоки в прямых ребрах минус потоки в обратных ребрах) один и тот же. Например, что касается потока в сети слева, то поток через сечение, разделяющее вершины 0 1 3 и 2 4 5 есть 2 + 1 + 2 (поток, соответственно, в ребрах 0-2, 1-4 и 3-5) минус 1 (поток в ребре 2-3) или 4. Эти подсчеты дают в результате значение 4 для каждого второго сечения в сети, а поток принимает максимальное значение, поскольку его величина равна пропускной способности минимального сечения (см. свойство 22.5). В этой рассматриваемой сети имеются два минимальных сечения.

Другими словами, сечения представляют собой узкие места сети. В рассматривавшемся выше военном приложении противник, который не способен полностью отрезать вражеские войска от их баз снабжения, должен учитывать, что снабжение вражеских войск будет ограничено максимум пропускной способностью любого заданного сечения. Разумеется, мы можем предполагать, что затраты на построение сечения в этом приложении пропорциональны его пропускной способности, заставляя тем самым наступающую армию искать решение задачи о минимальном потоке. Более того, из этих фактов, в частности, вытекает, что ни один поток не может иметь величины, превосходящей пропускную способность любого минимального сечения.

Свойство 22.5. (Теорема о максимальных потоках и минимальных сечениях). *Максимальный поток среди всех st-потоков в сети равен минимальной пропускной способности среди всех st-сечений.*

Доказательство: Достаточно найти такой поток и такое сечение, чтобы величина потока была равна пропускной способности этого сечения. Этот поток должен быть максимальным, поскольку величина никакого другого потока не может превысить пропускной способности сечения, а сечение должно быть минимальным, поскольку пропускная способность никакого другого сечения не может быть меньше величины потока, протекающего через этот сечение (свойство 22.4). Алгоритм Форда-Фалкерсона точно определяет такой поток и такое сечение: когда этот алгоритм завершает работу, выделите первое заполненное прямое или опорожненное обратное ребро для каждого пути из вершины s в вершину t графа. Пусть C_s есть множество всех вершин, достижимых из s через неориентированный путь, который не содержит наполненного прямого или порожнего обратного ребра, и пусть C_t есть множество всех остальных вершин. Тогда вершина t должна находиться в C_t, так что (C_s, C_t) есть *st*-сечение, разделяющее множество которого состоит исключительно из заполненных прямых или порожних обратных ребер. Поток через это сечение равен пропускной способности этого сечения (поскольку прямые ребра заполнены, а обратные ребра пусты), равной величине потока сети (свойство 22.3). ■

Это доказательство однозначно устанавливает, что алгоритм Форда-Фалкерсона способен отыскать максимальный поток. Независимо от того, какой метод мы выберем для того, чтобы найти аугментальный путь, а также независимо от того, какой путь мы найдем, он всегда заканчивается определением сечения, поток которого равен его пропускной способности, и в силу этого обстоятельства, равен величине потока сети, который по этой причине должен быть максимальным потоком.

Другое следствие правильности алгоритма Форда-Фалкерсона заключается в том, что для любой транспортной сети с целочисленными пропускными способностями ребер, существует решение задачи определения максимального потока, в котором все потоки также принимают целочисленные значения. Каждый аугментальный путь увеличивает величину потока сети на некоторое положительное целое число (минимальная из неиспользованных пропускных способностей в прямых ребрах и потоки в обратных ребрах, величины которых всегда принимают положительные значения). Этот факт оправдывает наше решение уделить основное внимание целочисленным значениям пропускных способностей и величин потоков. Можно применить максимальные потоки с величинами, выраженными нецелыми числами, даже в тех случаях, когда все пропускные способности суть целые числа (см. упражнение 22.23), однако в этом пока нет необходимости.

Это ограничение важно само по себе: обобщения, допускающие пропускные способности и потоки, выраженные вещественными числами, могут привести к неприятным аномальным ситуациям. Например, алгоритм Форда-Фалкерсона может привести к появлению бесконечной последовательности аугментальных путей, которые даже не сходятся к величине максимального потока.

Обобщенный алгоритм Форда-Фалкерсона не дает специального описания какого-либо конкретного метода определения аугментальных путей. Возможно, наиболее перспективное направление дальнейших исследований заключается в использовании обобщенной стратегии поиска на графе, рассмотренной в разделе 18.8. С этой целью мы начнем со следующего определения.

Определение 22.4. *Пусть задана транспортная сеть и поток в ней,* **остаточная сеть (residual network)** *для потока в транспортной сети содержит те же вершины, что и исходная сеть, и одно или два ребра в остаточной сети приходится на каждое ребро исходной сети, которые определены следующим образом: для каждого ребра* **v-w** *в исходной сети пусть* **f** *есть поток,* **c** *— пропускная способность. Если* **f** *положительно, следует включить* **w-v** *в остаточную сеть с пропускной способностью* **f**; *и если* **f** *меньше c, необходимо включить ребро* **v-w** *в остаточную сеть с пропускной способностью* **c-f**.

Если ребро **v-w** порожнее (**f** равно **0**), в остаточной сети существует одно ребро, соответствующее **v-w**, с пропускной способностью **c**; если ребро **v-w** заполнено (**f** равно **c**), то в остаточной сети существует единственное ребро с пропускной способностью **f**, соответствующее ребру **v-w**; и если **v-w** ни заполнено, ни пусто, то оба ребра **v-w** и **w-v** входят в остаточную сеть с соответствующими им пропускными способностями.

Программа 22.2 определяет класс **EDGE**, который мы используем для реализации абстракции остаточной сети, с функциями-элементами класса. При такой реализации мы будем работать исключительно с указателями на ребра клиентов. Наши алгоритмы работают с остаточной сетью, однако, по существу, они проверяют пропускные способности и изменения потоков (через указатели на ребра) ребер в клиентских программах. Функции-элементы **from** и **Plotherl** позволяют производить обработку ребер с любой ориентацией: функция **e.other(v)** возвращает конечную точку **e**, которая не есть **v**. Функции-элементы **capRto** и **addflowRto(v)** реализуют остаточную сеть: если **e** есть указатель на ребро **v-w** с пропускной способностью **c** и потоком **f**; то **e-> capRto(w)** есть **c-f**, а **e-> capRto(v)** есть **f**; **e-> addflowRto(w, d)** добавляет величину **d** к потоку в этом ребре, а **e-> addflowRto(v, d)** вычитает величину **d** из этого потока.

Программа 22.2. Ребра транспортной сети

Чтобы реализовать транспортную сеть, мы используем класс **GRAPH** *неориентированного* графа из главы 20, позволяющий манипулировать указателями на ребра, которые реализует этот интерфейс. Ребра *ориентированы*, тем не менее функции-элементы реализуют абстракцию остаточной сети, которая охватывает оба направления каждого ребра (см. текст).

```
class EDGE
{ int pv, pw, pcap, pflow;
public:
   EDGE(int v, int w, int cap) :
      pv(v), pw(w), pcap(cap), pflow(0) { }
   int v() const { return pv; }
   int w() const { return pw; }
```

```
   int cap() const { return pcap; }
   int flow() const { return pflow; }
   bool from (int v) const
      { return pv == v; }
   int other(int v) const
      { return from(v) ? pw : pv; }
   int capRto(int v) const
      { return from(v) ? pflow : pcap - pflow; }
   void addflowRto(int v, int d)
      { pflow += from(v) ? -d : d; }
};
```

Остаточные сети позволяют нам использовать любой обобщенный поиск на графе (см. раздел 18.8) для поиска аугментального пути, поскольку любой путь из истока в сток в остаточной сети непосредственно соответствует аугментальному пути в исходной сети. Увеличение потока вдоль этого пути приводит к изменениям в остаточной сети: например, по меньшей мере, одно ребро в остаточной сети меняет направление или исчезает (однако практикуемый нами способ абстрактной остаточной сети означает, что мы просто проводим проверку с целью отыскания положительной пропускной способности, и фактически нам не нужно вставлять и удалять ребра). На рис. 22.16 в качестве примера показана последовательность аугментальных путей и соответствующие остаточные сети.

Программа 22.3 представляет собой реализацию, построенную на базе очереди с приоритетами, которая охватывает все эти возможности, используя для этой цели слегка модифицированную версию полученной нами реализации поиска по приоритету на графах из программы 21.1, которая показана в программе 22.4. Эта реализация позволяет выбирать нужную реализацию алгоритма Форда-Фалкерсона из нескольких различных классических реализаций этого алгоритма за счет простого выбора приоритетов, который обеспечивает реализацию различных структур данных для бахромы.

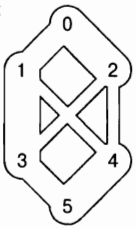
Как было показано в разделе 21.2, использование очереди с приоритетами для реализации стека, очереди или рандомизированной очереди для структуры данных типа бахрома влечет употребление дополнительного множителя $\lg V$, учитывающего затраты на операции с бахромой. Ввиду того, что мы можем избежать этих затрат, воспользовавшись АТД обобщенной очереди в реализациях, подобных программе 18.10 с прямыми реализациями, мы полагаем при анализе алгоритмов, что затраты на операции с бахромой в рассматриваемых случаях постоянны. Используя единственную реализацию программы 22.3, мы тем самым подчеркиваем наличие прямой связи между различными реализациями алгоритма Форда-Фалкерсона.

Несмотря на свою универсальность, программа 22.3 не охватывает всех реализаций алгоритма Форда-Фалкерсона (см. например, упражнения 22.36 и 22.38). Исследователи продолжают разрабатывать новые пути реализации этого алгоритма. Однако семейство алгоритмов, охватываемых программой 22.3, получило широкое распространение, и оно служит нам основой для понимания вычислений максимальных потоков и знакомит с реализациями, которые хорошо зарекомендовали себя в практических сетях.

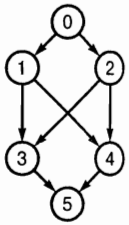
РИСУНОК 22.16. ОСТАТОЧНЫЕ СЕТИ (АУГМЕНТАЛЬНЫЕ ПУТИ)

*Вычисление аугментальных путей в транспортной сети эквивалентно поиску ориентированных путей в остаточной сети, определяемой потоком. Для каждого ребра в исходной транспортной сети мы создаем в остаточной сети ребро в каждом направлении: одно в направлении потока с весом, равным неиспользованной пропускной способности, а другое — в обратном направлении с весом, равным потоку. Ни в одном из случаев мы не включаем в сеть ребра, вес которых равен **0**. Первоначально (диаграмма вверху) остаточная сеть ничем не отличается от транспортной сети с весами ребер, равными их пропускным способностям. Когда мы производим наращивание потока вдоль пути **0-1-3-5** (вторая диаграмма сверху), мы заполняем ребра **0-1** и **3-5** до их пропускной способности с таким расчетом, чтобы они поменяли направление в остаточной сети, мы уменьшаем вес ребра **1-3** с тем, чтобы он соответствовал оставшемуся потоку, и добавляем ребро **3-1** с весом 2. Аналогично, когда мы производим наращивание вдоль пути **0-2-4-5**, мы заполняем ребро **2-4** до его пропускной способности, так что оно меняет направление на обратное, и получаем ребра, ведущие в разных направлениях между вершинами **0** и **2** и между вершинами **4** и **5**, представляющие поток и неиспользованную пропускной способность. После того, как будет проведено наращивание потока вдоль пути **0-2-3-1-4-5** (диаграмма внизу), в остаточной сети не остается ни одного ориентированного пути из истока в сток, в связи с чем отсутствуют аугментальные пути.*

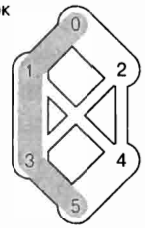
пропускная способность		поток
0-1	2	0
0-2	3	0
1-3	3	0
1-4	1	0
2-3	1	0
2-4	1	0
3-5	2	0
4-5	3	0



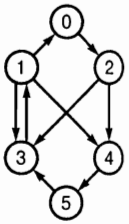
0-1	2
0-2	3
1-3	3
1-4	1
2-3	1
2-4	1
3-5	2
4-5	3



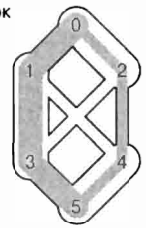
пропускная способность		поток
0-1	2	2
0-2	3	0
1-3	3	2
1-4	1	0
2-3	1	0
2-4	1	0
3-5	2	2
4-5	3	0



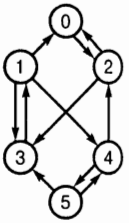
		1-0 2
0-2	3	
1-3	1	3-1 2
1-4	1	
2-3	1	
2-4	1	
		5-3 2
4-5	3	



пропускная способность		поток
0-1	2	2
0-2	3	1
1-3	3	2
1-4	1	0
2-3	1	0
2-4	1	1
3-5	2	2
4-5	3	1

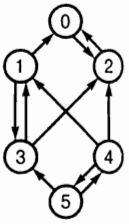


		1-0 2
0-2	2	2-0 1
1-3	1	3-1 2
1-4	1	
2-3	1	
		4-2 1
		5-3 2
4-5	2	5-4 1



пропускная способность		поток
0-1	2	2
0-2	3	2
1-3	3	1
1-4	1	1
2-3	1	1
2-4	1	1
3-5	2	2
4-5	3	2

		1-0 2
0-2	1	2-0 2
1-3	2	3-1 1
		4-1 1
		3-2 1
		4-2 1
		5-3 2
4-5	1	5-4 2



Программа 22.3. Реализация максимального потока в аугментальных путях.

Предлагаемый класс реализует общий алгоритм определения максимального потока посредством вычисления аугментальных путей (Форд-Фалькерсон). Он использует поиск по приоритету с целью обнаружения пути из истока в сток в остаточной сети (см. программу 22.4), затем добавляет в этот путь максимально возможную величину потока, многократно повторяя этот процесс до тех пор, пока не останется ни одного такого пути. Построение объекта этого класса устанавливает в заданных ребрах сети такие величины потоков, что поток из истока в сток становится максимальным.

Вектор **st** сохраняет остовное дерево с поиском по приоритету, при этом в **st[v]** содержится указатель на ребро, которое соединяет вершину **v** с ее родителем. Функция **ST** возвращает имя родителя вершины, заданной в качестве ее аргумента. Функция **augment** использует **ST** для обхода пути с целью определения его пропускной способности и последующего наращивания потока.

```
template <class Graph, class Edge> class MAXFLOW
{ const Graph &G;
   int s, t;
   vector<int> wt;
   vector<Edge *> st;
   int ST(int v) const { return st[v]->other(v); }
   void augment(int s, int t)
      { int d = st[t]->capRto(t);
        for (int v = ST(t); v != s; v = ST(v))
          if (st[v]->capRto(v) < d)
            d = st[v]->capRto(v);
        st[t]->addflowRto(t, d);
        for (int v = ST(t); v != s; v = ST(v))
           st[v]->addflowRto(v, d);
      }
   bool pfs();
public:
   MAXFLOW(const Graph &G, int s, int t) : G(G),
      s(s), t(t), st(G.V()), wt(G.V())
   { while (pfs()) augment(s, t); }
};
```

Как мы вскоре убедимся, эти базовые алгоритмические инструментальные средства дают нам простые (и полезные во многих приложениях) решения задачи о потоках в транспортных сетях. Однако всесторонние исследования, показывающие, какой из конкретных методов является наилучшим, сами по себе являются достаточно сложной задачей, поскольку время выполнения этих методов зависит от:

- Числа аугментальных путей, необходимых для отыскания максимального пути.
- Времени, необходимого для отыскания каждого аугментального пути.

Эти количественные величины могут изменяться в широких пределах, в зависимости от вида обрабатываемой сети и от стратегии поиска на графе (структура данных применительно к бахроме).

Программа 22.4. Реализация поиска по приоритету для определения аугментальных путей

Данная реализация поиска по приоритету была получена на основе реализации, которую мы использовали для алгоритма Дейкстры (программа 21.1) и в которую были внесены изменения, в соответствии чем веса приняли целочисленные значения, с целью обеспечения возможности обработки ребер остаточной сети и обеспечения останова при достижении стока или возврата значения **false**, если не существует пути из истока в сток. Заданное определение приоритета **P** позволяет получить аугментальный путь с максимальной пропускной способностью (отрицательные величины сохраняются для очередей по приоритетам с тем, чтобы были выполнены требования интерфейса программы 20.10); другие определения приоритета **P** приводят к различным алгоритмам вычисления максимальных алгоритмов.

```
template <class Graph, class Edge>
bool MAXFLOW<Graph, Edge>::pfs()
  { PQi<int> pQ(G.V(), wt);
    for (int v = 0; v < G.V(); v++)
      { wt[v] = 0; st[v] = 0; pQ.insert(v); }
    wt[s] = -M; pQ.lower(s);
    while (!pQ.empty())
      { int v = pQ.getmin(); wt[v] = -M;
        if (v == t || (v != s && st[v] == 0)) break;
        typename Graph::adjIterator A(G, v);
        for (Edge* e = A.beg(); !A.end(); e = A.nxt())
          { int w = e->other(v);
            int cap = e->capRto(w);
            int P = cap < -wt[v] ? cap : -wt[v];
            if (cap > 0 && -P < wt[w])
              { wt[w] = -P; pQ.lower(w); st[w] = e; }
          }
      }
    return st[t] != 0;
}
```

По-видимому, простейшая реализация алгоритма Форда-Фалкерсона использует кратчайший аугментальный путь (измеренный по числу ребер, образующих этот путь, а не по потоку или по пропускной способности). Этот метод был предложен Эдмондсоном (Edmondson) и Карпом (Karp) в 1972 г. Чтобы его реализовать, мы организуем очередь для бахромы либо путем использования возрастающего счетчика для **P**, либо путем использования АТД очереди вместо АТД очереди с приоритетами, которая применялась в программе 22.3. В этом случае поиск аугментального пути сводится к поиску в ширину (BFS) в остаточной сети, точно такому, какой был описан в разделах 18.8 и 21.2. На рис. 22.17 показана реализация этого метода Форда-Фалкерсона в действии на демонстрационном примере сети. Для краткости мы будем называть этот метод алгоритмом вычисления максимального потока с использованием *кратчайшего аугментального пути*. Из этого рисунка легко видеть, что длины аугментальных путей образуют неубывающую последовательность. Проведенный нами анализ этого метода во время доказательства свойства 22.7 показывает, что это свойство является для него характерным.

Другая реализация алгоритма Форда-Фалкерсона, предложенная Эдмондсоном и Карпом, может быть описана следующим образом: *выполняется наращивание вдоль пути, который увеличивает поток на наибольшую величину*. Значение приоритета **P**, которое было ис-

пользовано в программе 22.3, обеспечивает реализацию этого метода. Этот приоритет заставляет алгоритм выбирать ребра из бахромы таким образом, чтобы количество потока, который пропускается через прямое ребро или удаляется из обратного ребра, было максимальным. Этот метод мы будем называть *алгоритмом вычисления максимального потока путем наращивания потока до максимальной пропускной способности (maximum-capacity-augmenting-path maxflow algorithm)*. Рисунок 22.18 служит иллюстрацией работы этого алгоритма на той же транспортной сети из рис. 22.17.

Это всего лишь два примера (которые мы сейчас способны анализировать!) реализаций алгоритма Форда-Фалкерсона. В конце этого раздела мы изучим также и другие реализации. Однако прежде чем приступить к такому изучению, мы рассмотрим задачу анализа методов вычисления аугментальных путей с тем, чтобы исследовать их свойства и, в конечном итоге, определить, какой из методов обеспечивает наивысшую производительность.

Пытаясь сделать нужный нам выбор из семейства алгоритмов, представленных программой 22.3, мы попадаем в привычную ситуацию. Должны ли мы прежде всего принимать во внимание производительность, обеспечиваемую выбираемым алгоритмом в худшем случае, или это всего лишь математический домысел, не имеющий отношения к сетям, с которыми мы имеем дело на практике?

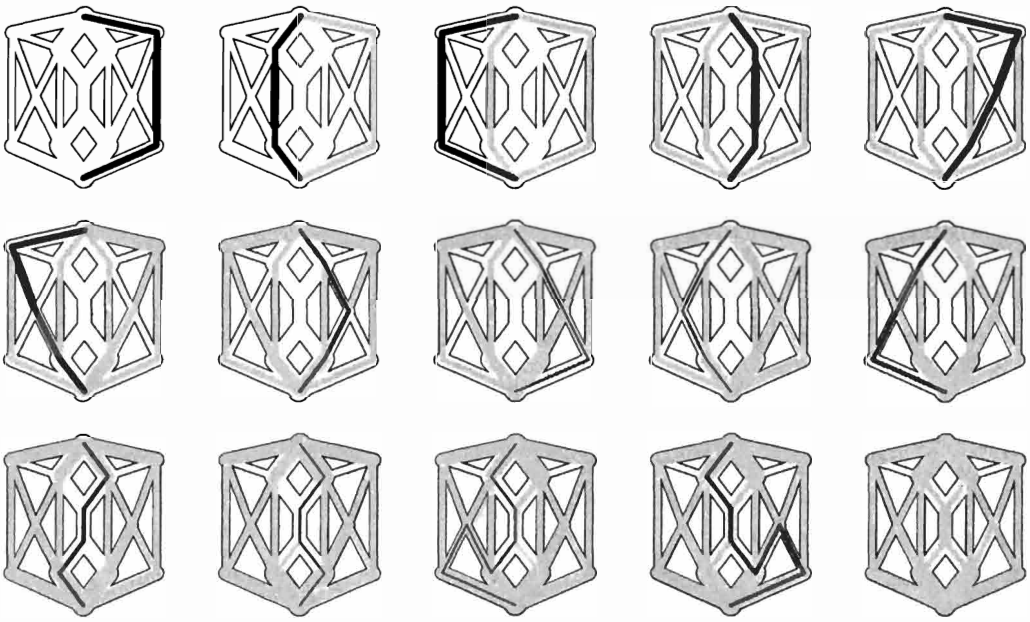


РИСУНОК 22.17. КРАТЧАЙШИЕ АУГМЕНТАЛЬНЫЕ ПУТИ

Рассматриваемая последовательность диаграмм показывает, как реализация метода Форда-Фалкерсона, выполняющая поиск кратчайшего аугментального пути, находит максимальный поток на демонстрационном примере транспортной сети. По мере продвижения алгоритма, длины путей увеличиваются: первые четыре пути в верхнем ряду имеют длину 3, последний путь в верхнем ряду и все пути во втором ряду имеют длину 4; два пути в нижнем ряду имеют длину 5, и процесс завершается отысканием двух путей длиной 7, при этом в каждом из них присутствует обратное ребро.

Этот вопрос особо актуален в рассматриваемом контексте, поскольку границы в классическом худшем случае, на которые мы можем ориентироваться, намного выше, чем фактическая производительность, которую мы получаем при работе с графами на практике.

Многие другие факторы еще больше осложняют ситуацию. Например, время выполнения в худшем случае для многих версий зависит не только от числа вершин V и числа ребер E, но также и от значений пропускных способностей ребер в сети. Разработка алгоритма максимального потока с высокой гарантированной производительностью в течение нескольких десятилетий привлекала к себе внимание многих исследователей, которые предложили множество методов решения. Оценка всех этих методов применительно к различным типам сетей, которые встречаются на практике, с достаточной степенью точности позволяет нам сделать правильный выбор, однако она более расплывчата, чем оценка той же задачи в других ситуациях, которые мы изучали ранее, например, в таких, как практические приложения алгоритмов сортировки или поиска.

Памятуя обо всех этих трудностях, рассмотрим теперь классические результаты, касающиеся производительности метода Форда-Фалкерсона в худшем случае: одну общую границу и две специальные границы, по одной на каждый из алгоритмов вычисления аугментального пути, которые мы уже освоили. Эти результаты в большей степени расширяют наше представление о внутренних характеристиках алгоритмов, чем обеспечивают возможность прогнозировать с достаточной степенью точности производительность этих алгоритмов с целью их сравнения и последующих выводов. Эмпирическое сравнение этих методов мы проведем в конце этого раздела.

Свойство 22.6. *Пусть M есть максимальная пропускная способность ребер в сети. Число аугментальных путей, необходимых для любой реализации алгоритма Форда-Фалкерсона, равно максимум VM.*

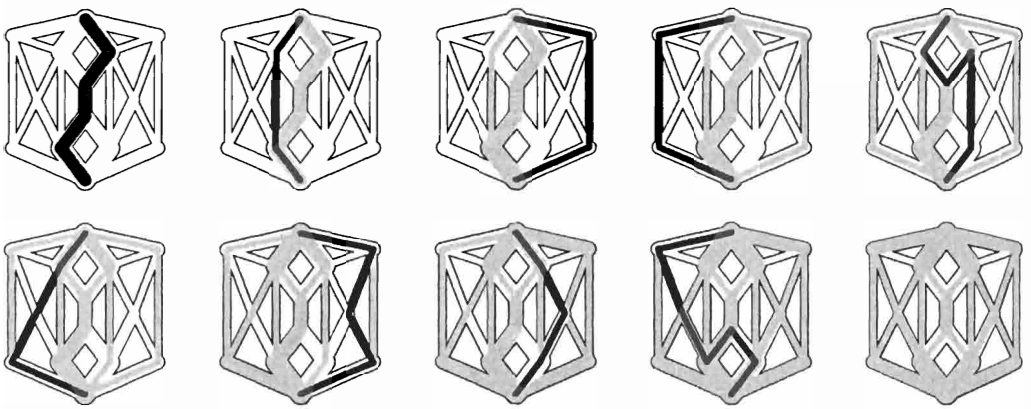


РИСУНОК 22.18. АУГМЕНТАЛЬНЫЕ ПУТИ С МАКСИМАЛЬНОЙ ПРОПУСКНОЙ СПОСОБНОСТЬЮ

Рассматриваемая последовательность диаграмм показывает, как реализация метода Форда-Фалкерсона, выполняющая поиск аугментального пути с максимальной пропускной способностью, находит максимальный поток на демонстрационном примере транспортной сети. По мере продвижения алгоритма, пропускная способность путей уменьшается, однако их длина может как увеличиваться, так и уменьшаться. Этому методу требуется найти всего лишь девять аугментальных путей, чтобы вычислить тот же максимальный поток, который представлен на рис. 22.17.

Доказательство: Любое сечение содержит максимум V ребер с пропускной способностью M, т.е., их максимальная пропускная способность есть VM. Каждый аугментальный путь увеличивает поток через сечение, по меньшей мере, на 1, следовательно, выполнение алгоритма прекратится после VM проходов, поскольку в результате выполнения множества процедур наращивания все сечения должны быть насыщены до их пропускной способности. ■

Как уже говорилось выше, в типовых ситуациях от такой границы мало проку, поскольку M может быть очень большим числом. Что еще хуже, легко описать ситуацию, когда число итераций пропорционально максимальной пропускной способности ребра. Например, предположим, что мы используем алгоритм определения *самого длинного* аугментального пути (возможно, основанного на интуитивном представлении, что чем больше путь, тем больший поток мы загружаем в ребра сети). Поскольку мы ведем подсчет итераций, мы на время игнорируем затраты на вычисление такого пути. Классический пример, представленный на рис. 22.19, показывает сеть, для которой число итераций алгоритма определения самого длинного аугментального пути равно максимальной пропускной способности ребер. Этот пример свидетельствует о том, что мы должны проводить более подробные исследования, чтобы выяснить, обеспечивают ли другие конкретные реализации существенное уменьшение итераций по сравнению с оценкой, данной свойством 22.6.

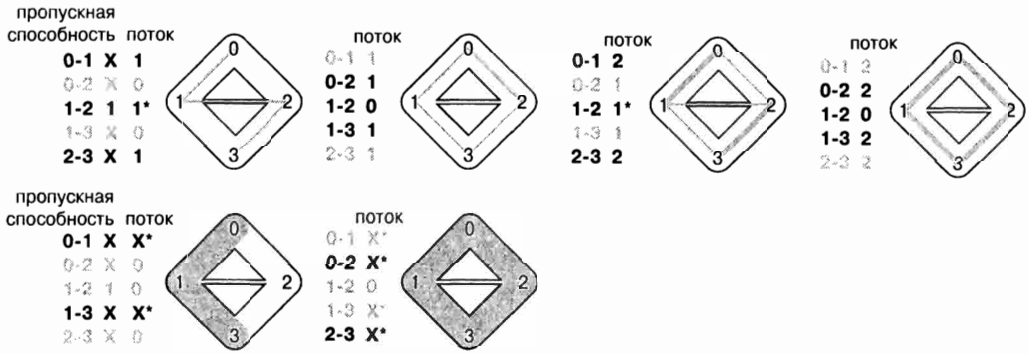


РИСУНОК 22.19. ДВА СЦЕНАРИЯ ДЛЯ АЛГОРИТМА ФОРДА-ФАЛКЕРСОНА

*Представленная на рисунке сеть служит иллюстрацией того, что число итераций, выполняемых алгоритмом Форда-Фалкерсона, зависит от пропускных способностей ребер в сети и от последовательности аугментальных путей, выбираемых реализацией рассматриваемого алгоритма. Она состоит из четырех ребер с пропускной способностью **X** и одного ребра с пропускной способностью 1. Сценарий, описание которого дано в верхней части рисунка, показывает, что реализация, которая попеременно использует цепочки **0-1-2-3** и **0-2-1-3** как аугментальные пути (например, та, что предпочитает длинные пути), потребует X пар итераций, подобных двум парам, показанным на рисунке, причем каждая такая пара увеличивает общих поток на 2. Сценарий в нижней части рисунка показывает, что реализация, которая выбирает цепочки **0-1-3**, а затем **0-2-3** в качестве аугментальных путей (например, та, что предпочитает короткие пути), определяет максимальный поток всего за две итерации.*

Если пропускные способности ребер выражены, скажем, 32-разрядными целыми числами, быстродействие сценария, описание которого помещено в верхней части рисунка, будет в миллиард раз меньшим быстродействия сценария в нижней части рисунка.

Для разреженных сетей и сетей с малыми целочисленными значениями пропускных способностей ребер, свойство 22.6 на самом деле дает верхнюю границу выполнения любой реализации алгоритма Форда-Фалкерсона, пригодную для практического применения.

Следствие. *Время, необходимое для нахождения максимального потока, равно $O(VEM)$, оно же для разреженных сетей равно $O(V^2M)$.*

Доказательство: Это утверждение непосредственно следует из фундаментального результата, утверждающего, что обобщенный поиск на графе линейно зависит от размера представления графа (свойство 18.12). Как уже говорилось выше, если мы используем реализацию бахромы в виде очереди с приоритетами, мы учитываем это обстоятельство, вводя дополнительный коэффициент $\lg V$. ■

Это доказательство фактически устанавливает тот факт, что сомножитель M можно заменить отношением наибольшей и наименьшей пропускной способности в сети (см. упражнение 22.35). Когда это отношение мало, граница подсказывает нам, что любая реализация алгоритма Форда-Фалкерсона обнаруживает максимальный поток за время, пропорциональное времени, затрачиваемому (например) в худшем случае на решение задачи нахождения всех кратчайших путей. Существует множество ситуаций, когда пропускные способности и в самом деле низкие, так что сомножитель M можно не учитывать. В разделе 22.4 мы рассмотрим соответствующий пример.

Когда M велико, граница VEM в худшем случае также велика; но это наиболее пессимистический вариант, поскольку мы получили его перемножением границ всех случаев, которые вытекают из придуманных нами примеров. Фактические затраты для сетей, встречающихся на практике, обычно намного ниже.

С теоретической точки зрения наша ближайшая цель состоит в том, чтобы определить, используя приблизительную субъективную классификацию, предложенную в разделе 17.8, можно ли найти решение задачи о максимальном потоке в сетях с большими целочисленными весами (решаемой при использовании алгоритма с полиномиальным временем выполнения). Только что найденные границы не решают этой проблемы, поскольку максимальный вес $M = 2^m$ может возрастать экспоненциально с увеличением V и E. С практических позиций мы нуждаемся в более надежных гарантиях производительности. Возьмем типовой пример из практики, предположим, что мы используем 32-разрядные целые числа ($m = 32$) для представления весов. В графе со многими сотнями вершин и тысячами ребер по свойству 22.6 мы должны будем выполнять сотни триллионов (10^{12}) операций при прогоне алгоритма вычисления аугментального пути. Если сеть содержит миллионы вершин, то этот вопрос требует дальнейших исследований, поскольку мы не только не можем иметь дело с весами, величина которых достигает порядка $2^{1000000}$, но еще и потому, что значения V^3 и VE настолько велики, что любые границы при этом теряют всякий смысл. Мы же заинтересованы в том, чтобы найти полиномиальные границы для ответа на вопрос о возможности использования соответствующей модели и нахождения более точных границ, соответствующих ситуации, с которой мы можем столкнуться на практике.

Свойство 22.6. носит общий характер: оно применимо к любой реализации алгоритма Форда-Фалкерсона. Универсальная природа этого алгоритма предоставляет нам достаточную гибкость, чтобы можно было рассматривать целый ряд простых реализаций, преследуя цель улучшить рабочие характеристики алгоритма. Мы надеемся, что конкрет-

ные реализации послужат объектом исследований с целью определения более точных границ для худших случаев. В самом деле, это одна из главных причин, чтобы провести их исследование в первую очередь! Теперь, как мы могли убедиться, реализация и использование большого класса таких реализаций тривиальны: мы просто выполняем подстановку различных реализаций обобщенной очереди или определений приоритетов в программу 22.3. Анализ различий поведения в условиях худшего случая представляет собой еще более трудную задачу, что подтверждается классическими результатами, к изучению которых применительно к двум базовым реализациям рассмотренного выше алгоритма вычисления аугментальных путей мы далее переходим.

Сначала мы проводим анализ алгоритма вычисления кратчайшего аугментального пути. Этот метод не является предметом задачи, представленной на рис. 22.19. В самом деле, мы не можем использовать ее для замены множителя M выражением $VE/2$ при определении времени выполнения, тем самым утверждая, что задача определения потоков в сетях относится к числу решаемых. Мы даже можем классифицировать ее как относящуюся к категории легко решаемых (решаемую за полиномиальное время в практических случаях посредством простой, но в то же время искусной реализации).

Свойство 22.7. *Число аугментальных путей, необходимых для реализации алгоритма Форда-Фалкерсона, использующей кратчайшие аугментальные пути, не превышает VE/2.*

Доказательство: Сначала, как это следует из примера на рис. 22.17, ни один из кратчайших путей не короче предыдущего. Чтобы установить этот факт, мы покажем методом доказательства от противного, что справедливо более сильное свойство: никакой аугментальный путь не может уменьшить длину кратчайшего пути из вершины s в любую другую вершину остаточной сети. Предположим, тем не менее, что некоторому аугментальному пути это удается, а вершина v есть первая вершина на этом пути. При этом мы рассмотрим два случая: либо никакая вершина на новом, более коротком пути из s в v не появляется в этом аугментальном пути, либо некоторая вершина w на новом кратчайшем пути из s в v появляется где-то между v и t в аугментальном пути. Обе ситуации противоречат условию минимальности аугментального пути.

Теперь, по построению, каждый аугментальный путь содержит, по меньшей мере, одно *критическое ребро*: ребро, которое удаляется из остаточной сети, поскольку оно соответствует либо прямому ребру, которое наполняется до пропускной способности, либо обратному ребру, которое опорожняется. Предположим, что ребро **u-v** есть критическое ребро для аугментального пути P длиной **d**. Следующий аугментальный путь должен иметь, по меньшей мере, длину **d + 2**, поскольку этот путь должен проходить из **s** в **v**, затем вдоль ребра **u-v**, затем из **u** в **t**. Первый сегмент имеет, по меньшей мере, длину, на 1 больше, чем расстояние от **s** в **u** в P, а заключительный сегмент длины, по меньшей мере, на 1 больше, чем расстояние из **v** в **t** в P, так что длина этого пути на 2 больше, чем P.

Поскольку длины аугментальных путей не могут быть больше V, из этого факта следует, что каждое ребро может оказаться критическим на максимум $V/2$ аугментальных путях, так что общее число аугментальных путей не может быть больше $EV/2$. ■

Следствие. *Время, необходимое для отыскания максимального потока в разреженной сети, равно $O(V^3)$.*

Доказательство: Время, необходимое для отыскания аугментального пути, есть $O(E)$, так что общее время есть $O(EV^2)$. Отсюда сразу же следует указанная выше граница. ■

Величина V^3достаточно велика, чтобы служить гарантией высокой производительности алгоритма на крупных сетях. Но этот факт не должен воспрепятствовать применению этого алгоритма на крупных сетях, поскольку это есть оценка его производительности в худшем случае, которая не всегда подходит для прогнозирования производительности алгоритма в практических ситуациях. Например, как только что было установлено, максимальная пропускная способность M (или максимальное отношение пропускных способностей) может быть намного меньше V, так что свойство 22.6 обеспечивает более точную границу. В самом деле, в лучшем случае число аугментальных путей, необходимых для метода Форда-Фалкерсона, меньше полустепени исхода вершины s или полустепени захода вершины t, которые, в свою очередь, могут быть намного меньше V. При наличии такого разброса производительности в лучшем и худшем случаях, сравнивать алгоритмы нахождения аугментальных путей, взяв за основу только границы худшего случая, по меньшей мере, неразумно.

Тем не менее, другие реализации, которые столь же просты, как и метод определения кратчайшего аугментального пути, могут иметь более точные границы или используют границы, хорошо зарекомендовавшие себя на практике (или оба вида границ). Например, в примере, представленном на рис. 22.17 и 22.18, алгоритм вычисления максимального аугментального пути использует намного меньше путей для отыскания максимального потока, чем метод определения кратчайшего аугментального пути. Теперь мы обратимся к анализу худшего случая рассматриваемого алгоритма.

Во-первых, так же, как и в случае алгоритма Прима и алгоритма Дейкстры (см. разделы 20.6 и 21.2), мы можем реализовать очередь с приоритетами таким образом, что для выполнения рассматриваемым алгоритмом одной итерации в худшем случае понадобится время, пропорциональное V^2 (для насыщенных графов), либо $(E + V) \log V$ (для разреженных графов), тем не менее, эти оценки пессимистичны, поскольку алгоритм сразу же остановится, как только достигнет стока. Мы также убедились, что можем немного улучшить рабочие характеристики, если воспользуемся более совершенными структурами данных. Однако более важной и более труднорешаемой задачей является определение необходимого числа аугментальных путей.

Свойство 22.8. *число аугментальных путей, необходимых для реализации алгоритма Форда-Фалкерсона с поиском максимальных аугментальных путей, не превышает* $2E \lg M$.

Доказательство: Пусть дана некоторая сеть, а F есть величина ее максимального потока. Пусть v есть значение потока в той точке выполнения алгоритма, когда мы начинаем поиск аугментального пути. Применяя свойство 22.2 к остаточной сети, мы можем разложить поток максимум на E ориентированных путей, что дает нам в сумме $F - v$, так что поток, по меньшей мере, в одном из путей есть, по меньшей мере, $(F - v)/E$. Теперь либо мы найдем максимальный поток перед тем, как выполнять построение следующих $2E$ аугментальных путей, либо величина такого аугментального пути после построения этой последовательности из $2E$ путей станет меньше, чем $(F - v)/2E$, что в свою очередь меньше, чем одна вторая от значения максимума, имевшего место перед тем, как была построена эта последовательность из $2E$ путей. То есть, в худшем случае нам нужна последовательность из $2E$ путей, чтобы уменьшить количество путей в два раза. Первое значение количества путей не может превышать M, которое мы должны уменьшать в два раза максимум $\lg M$ раз, так что в конечном итоге мы имеем максимум $\lg M$ последовательностей из $2E$ путей. ■

Следствие. *Время, необходимое для того, чтобы найти максимальный поток в разреженной сети, есть* $O(V^2 \lg M \lg V)$.

Доказательство: Это утверждение непосредственно следует из использования реализации очереди с приоритетами на основе сортирующего дерева, подобной использованной при доказательстве свойств 20.7 и 21.5. ■

Для значений M и V, которые обычно встречаются на практике, эта граница значительно ниже границы $O(V^3)$, определяемой свойством 22.7. Во многих практических ситуациях алгоритм поиска максимального аугментального пути использует значительно меньше итераций, чем алгоритм поиска кратчайшего аугментального пути, за счет несколько завышенной границы затрат на действия по поиску каждого пути.

Желательно изучить множество других вариантов, которые описаны в публикациях, посвященных алгоритмам определения максимальных потоков. Продолжают появляться новые алгоритмы с более точными границами для худшего случая, тем не менее, существование нетривиальной нижней границы до сих пор не доказано, так что все еще сохраняется возможность открытия алгоритма с простым линейным временем выполнения. Многие алгоритмы этого класса разрабатывались главным образом с целью понижения границ худшего случая для насыщенных графов, поэтому они не обеспечивают лучшей производительности, чем алгоритм поиска максимального аугментального пути для различных видов разреженных сетей, с которыми мы сталкиваемся на практике. Тем не менее, многие варианты ждут своих исследователей, поставивших своей целью обнаружение более эффективных практических алгоритмов определения максимальных потоков. Далее мы рассмотрим еще два алгоритма поиска аугментальных путей, а в разделе 22.3 подвергнем анализу другое семейство алгоритмов.

Один простой алгоритм поиска аугментальных путей предусматривает использование величины уменьшающегося счетчика P или стека для реализации обобщенной очереди из программы 22.3, благодаря чему поиск аугментальных путей уподобляется поиску в глубину. На рис. 22.20 показан поток, вычисленный для нашего небольшого примера по этому алгоритму. Интуиция подсказывает нам, что этот метод обладает подходящим быстродействием, легок в реализации и способен провести поток через сеть. Как мы увидим далее, его производительность меняется в широких пределах, от исключительно низкого уровня для одних видов сетей до вполне приемлемого для других.

Другой альтернативой является использование реализации рандомизированной очереди в качестве обобщенной очереди, вследствие чего поиск аугментального пути становится рандомизированным поиском. На рис. 22.21 показан поток, вычисленный для нашего небольшого примера с помощью рассматриваемого алгоритма. Этот метод также обладает приличным быстродействием и легок в реализации; кроме того, как мы уже отмечали в разделе 18.8, он может сочетать в себе достоинства как поиска в ширину, так и поиска в глубину. Рандомизация – это мощное инструментальное средства в разработке алгоритмов, и данная задача представляет собой ситуацию, в которой целесообразно ею воспользоваться.

Этот раздел мы завершим более подробным анализом методов, которые мы изучали с тем, чтобы прочувствовать все трудности, связанные с их сравнением между собой или с попытками прогнозирования рабочих характеристик для практических применений.

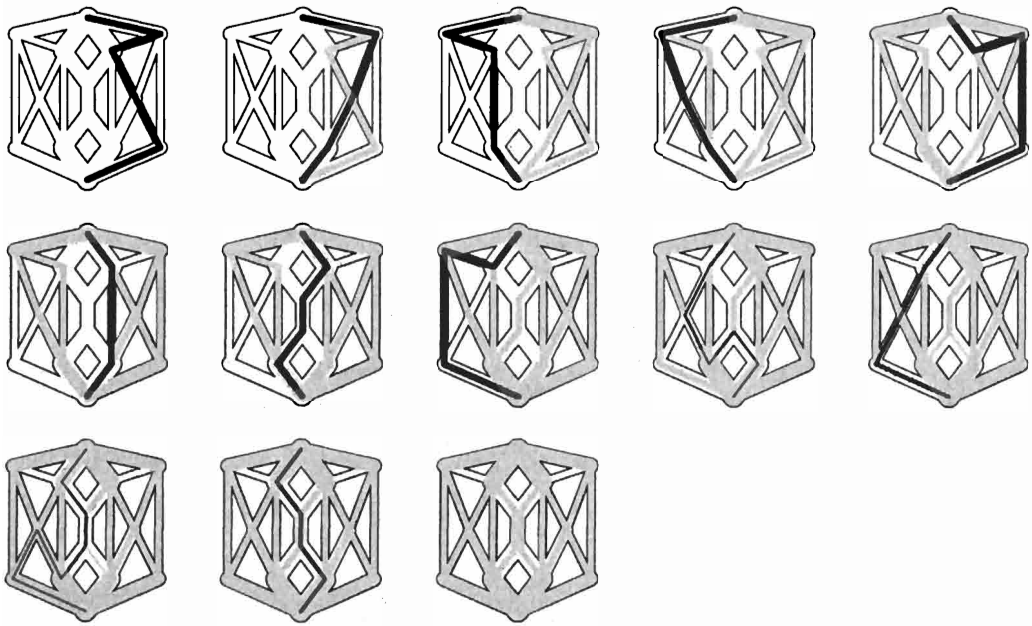


РИСУНОК 22.20. ПОИСК АУГМЕНТАЛЬНОГО ПУТИ С ИСПОЛЬЗОВАНИЕМ СТЕКА

Этот рисунок служит иллюстрацией использования стека для построения обобщенной очереди в нашей реализации метода Форда-Фалкерсона, благодаря чему поиск путей производится так же, как и поиск в глубину. В этом случае рассматриваемый метод работает почти так же успешно, как и поиск в ширину, но в то же время его несколько неустойчивое поведение в какой-то степени объясняется тем, что он довольно чувствителен от выбора представления сети и исследованию не подвергался.

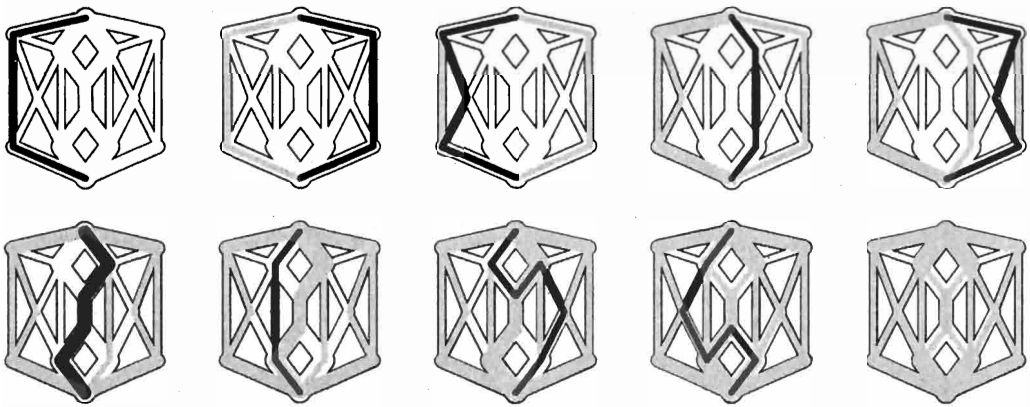


РИСУНОК 22.21. РАНДОМИЗИРОВАННЫЙ ПОИСК АУГМЕНТАЛЬНЫХ ПУТЕЙ

Эта последовательность есть результат использования рандомизированной очереди для структуры данных типа бахрома с целью поиска аугментальных путей в рамках метода Форда-Фалкерсона. В этом примере мы случайно натолкнулись на короткий путь с высокой пропускной способностью, благодаря чему нам потребовалось относительно небольшое число аугментальных путей. Несмотря на то что прогнозирование рабочих характеристик этого метода представляет собой достаточно трудную задачу, во многих ситуациях метод проявляет себя с лучшей стороны.

Чтобы получить начальное представление о возможных количественных различиях, мы воспользуемся двумя моделями транспортных сетей, построенных на основе эвклидовой модели графа, которые применялись раньше для сравнения с другими алгоритмами на графах. Обе модели используют граф, для построения которого использованы V точек на плоскости со случайными координатами, принимающими значения между 0 и 1, при этом ребрами соединяются точки, расположенные на фиксированном удалении друг от друга. Эти ребра отличаются друг от друга различными присвоенными им значениями пропускной способности.

Первая модель просто присваивает одно и то же постоянное значение каждой пропускной способности. В обсуждении, проводимом в разделе 22.4, утверждается, что этот вариант задачи о потоках в сети решается легче, чем эта же задача в общем виде. В рассматриваемых нами эвклидовых графах потоки ограничены полустепенью исхода истока и полустепенью захода стока, так что для каждого алгоритма требуется найти всего лишь несколько аугментальных путей. Однако эти пути существенно различаются для различных алгоритмов, и в этом мы вскоре убедимся.

Вторая модель присваивает случайные значения весов из некоторого фиксированного диапазона значений пропускных способностей. Эта модель генерирует типы сетей, которые исследователи обычно имеют в виду, обращаясь к данной задаче, и рабочие характеристики, которые различные алгоритмы демонстрируют на таких сетях, дают богатую пищу для размышлений.

Иллюстрацией обеих моделей служит рис. 22.22, на нем также показаны четыре потока, вычисленные четырьмя различными методами на двух сетях. Возможно, самой красноречивой характеристикой этих примеров является то, что сами потоки отличаются друг от друга по характеру. Все они имеют одну и ту же величину, но в рассматриваемых сетях имеется много максимальных потоков, в то же время различные алгоритмы выбирают различные ребра при их вычислении. На практике такая ситуация – обычное дело. Мы можем попытаться наложить дополнительные ограничения на потоки, которые мы намерены вычислить, однако такие изменения в условиях задачи существенно усложняют поиск ее решения. Задача отыскания потока минимальной стоимости, которую мы будем исследовать в разделах 22.5–22.7, представляет собой один из способов формализации подобных ситуаций.

В таблице 22.1 представлены более подробные количественные результаты, которые могут быть использованы всеми четырьмя методами для вычисления потоков в сетях, показанных на рис. 22.22. Производительность алгоритма вычисления аугментальных путей зависит не только от числа аугментальных путей, но также и от длины таких путей и затрат на их поиск. В частности, время выполнения алгоритма пропорционально числу ребер, проверенных во внутреннем цикле программы 22.3. Как обычно, это число может меняться в широких пределах, даже для одного и того же заданного графа, и зависит от свойств его представления; но основной нашей задачей является изучение характеристик различных алгоритмов. Например, на рис. 22.23 и 22.24 показаны деревья поиска, соответственно, алгоритма вычисления максимальной пропускной способности и алгоритма поиска кратчайшего пути. Эти примеры подтверждают общий вывод о том, что метод поиска кратчайшего пути затрачивает больше усилий на отыскание аугментальных путей с меньшим потоком, чем алгоритм вычисления максимальной пропускной способности, благодаря чему легче понять, почему предпочтение отдается последнему.

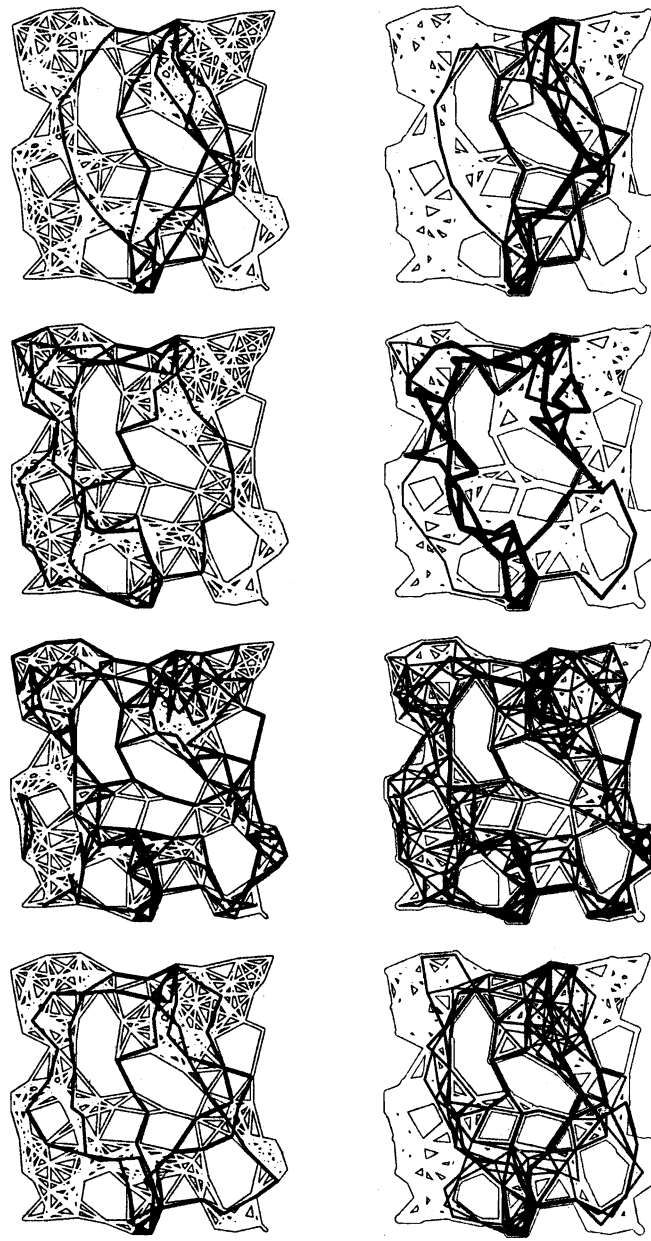


РИСУНОК 22.22. ТРАНСПОРТНАЯ СЕТЬ СО СЛУЧАЙНЫМИ ПОТОКАМИ

На этом рисунке представлены вычисления максимальных потоков в случайных эвклидовых графах на базе двух различных моделей пропускных способностей. В сетях слева всем ребрам присвоена единица в качестве пропускных способностей, в сетях справа ребрам присваиваются случайные значения пропускной способности. Исток находится в верхней части сети ближе к середине, а сток — в середине нижней части сети. Показанные на диаграммах потоки, направлены сверху вниз и вычислены с помощью, соответственно, алгоритма поиска кратчайшего пути, алгоритма вычисления максимальной пропускной способности, алгоритма, использующего стек, и рандомизирующего алгоритма. Поскольку вершины не имеют высоких степеней, а пропускные способности принимают малые целочисленные значения, существует много различных потоков, которые достигают максимума в условиях этих примеров.

Полустепень захода истока равна 6, так что все указанные выше алгоритмы вычисляет поток в рамках модели единичной пропускной способности, представленной в левой части рисунка, с помощью шести аугментальных путей.

Эти методы находят аугментальные пути, которые по своему характеру существенно отличаются для модели со случайными весами, представленной в правой части рисунка. В частности, методы, в основу которых положен стек, ищут длинные пути с малым весом и даже выполняют построение потока с разъединенным циклом.

Таблица 22.1. Эмпирический анализ алгоритмов вычисления аугментальных путей

В этой таблице представлены параметры производительности алгоритмов вычисления потоков в сетях путем построения аугментальных путей на примере эвклидовой сети с близкими связями (со случайными значениями пропускной способности и с максимальной величиной потока, равной 286) и с пропускными способностями, равными единице (при величине максимального потока, равной 1). Алгоритм вычисления максимальной пропускной способности превосходит все другие алгоритмы на обоих типах сетей. Алгоритм случайного поиска находит аугментальные пути, которые не намного длиннее, чем кратчайший путь, и проверяет меньшее число узлов. Алгоритм, использующий стек, очень хорошо зарекомендовал себя в условиях случайных весов, однако, несмотря на то что он находит очень длинные пути, он вполне может конкурировать с другими алгоритмами на сетях единичными весами.

	Число путей	Средняя длина	Общее число ребер
Случайное значение пропускной способности в пределах от 1 до 50			
Кратчайший путь	37	10.8	76394
Максимальная пропускная способность	7	19.3	15660
Поиск в глубину	286	123.5	631392
Случайный поиск	35	12.8	58016
Пропускная способность, равная 1			
Кратчайший путь	6	10.5	13877
Максимальная пропускная способность	6	14.7	10736
Поиск в глубину	6	110.8	12291
Случайный поиск	6	12.2	11223

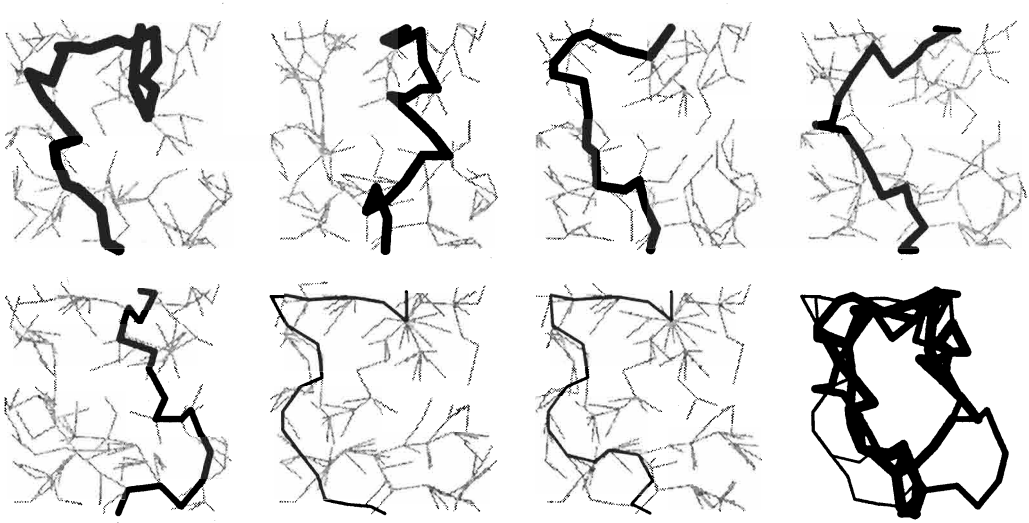


РИС. 22.23. АУГМЕНТАЛЬНЫЕ ПУТИ С МАКСИМАЛЬНОЙ ПРОПУСКНОЙ СПОСОБНОСТЬЮ (БОЛЬШОЙ ПРИМЕР)

На этом рисунке показаны аугментальные пути, вычисленные с посредством алгоритма определения максимальной пропускной способности для эвклидовой сети со случайными значениями весов, которая изображена на рис. 22.22, вместе с ребрами остовного дерева поиска на графе (изображены серым цветом). Полученный в результате поток показан на диаграмме внизу справа.

Возможно, основной урок, который можно вынести из подробного исследования конкретной сети, состоит в том, что различие в оценке верхней границы, полученной с применением свойств 22.6–22.8, и фактическим числом аугментальных путей, которые потребуются конкретному алгоритму в условиях заданного приложения, может оказаться огромным. Например, в транспортной сети, представленной на рис. 22.23, имеется 177 вершин и 2000 ребер с пропускной способностью, не превышающей 100 единиц, так что количественное значение $2E \lg M$ в соответствии со свойством 22.8 превышает 25000 путей, однако алгоритм вычисления максимальной пропускной способности находит максимальный поток, используя всего лишь семь аугментальных путей. Аналогично, количественная оценка $VE/2$ для этой сети, определяемая свойством 22.7, составляет порядка 177000 путей, однако алгоритму кратчайших путей для той же цели достаточно всего лишь 37 путей.

Как отмечалось выше, относительно низкие степени узлов и местоположение соединений частично объясняют причину таких расхождений между теоретической оценкой и производительностью, полученной на практике. Мы можем доказать, что существуют более точные границы производительности, учитывающие эти детали, однако подобные расхождения результатов, полученных на модели транспортной сети и в реальных сетях, являются скорее правилом, а не исключением. С другой стороны, на основании этих результатов мы можем утверждать, что эти сети имеют недостаточно общий характер, чтобы представлять сети, с которыми нам приходится иметь дело на практике; опять-таки, анализ худшего случая, возможно, еще больше далек от практики, чем все рассмотренные виды сетей.

Подобного рода огромные расхождения побуждают исследователей повышать интенсивность поисков с целью снижения границ для худшего случая. Существуют множество других возможных реализаций алгоритмов, использующих аугментальные пути, требующих подробных исследований, которые могут привести к более существенному улучшению их производительности в худшем случае или большему повышению производительности на практике, чем все те методы, которые мы изучали (см. упражнения 22.56–22.60). Многочисленные более сложные методы, которые, как мы показали, обладают более высокой производительностью, описаны в публикациях, посвященных результатам исследований (см. *раздел ссылок*).

При исследовании множества других задач, которые сводятся к задаче о максимальном потоке, возникают важные осложнения. Когда применяются подобного рода сведения, получаемая при этом транспортная сеть может обладать некоторой специальной структурой, особенностями которой могут воспользоваться специальные алгоритмы с целью повышения производительности. Например, в разделе 22.8 мы будем рассматривать сведение, которое дает транспортную сеть с единичной пропускной способностью всех ребер.

Даже когда мы ограничиваем внимание только алгоритмами вычисления аугментальных путей, мы убеждаемся в том, что изучение алгоритмов нахождения максимального потока можно рассматривать одновременно как науку и как искусство. Искусство проявляется в выборе стратегии, которая наиболее эффективна в данной практической ситуации, а наука лежит в основе понимания природы задачи. Помогут ли новые структуры данных решить задачу нахождения максимального потока за линейное время, либо мы сможем доказать, что такое решение не существует?

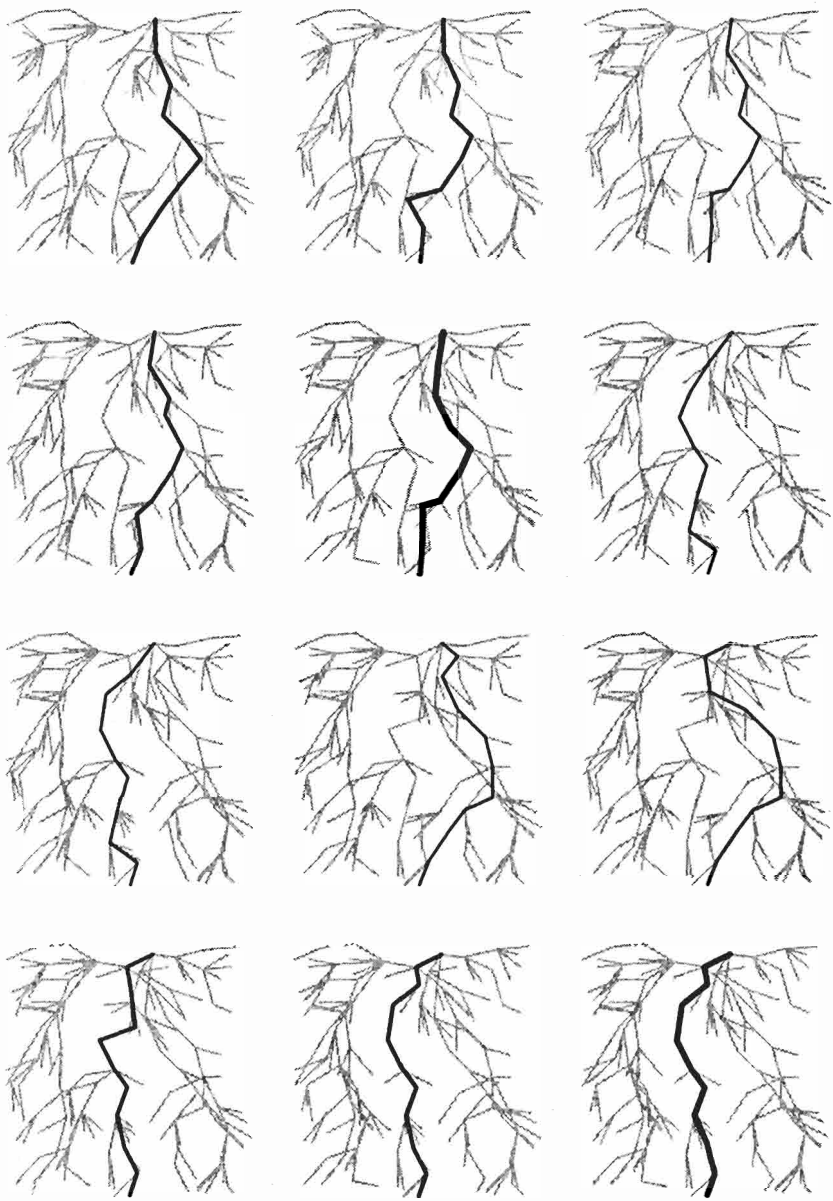


РИСУНОК 22.24. КРАТЧАЙШИЙ АУГМЕНТАЛЬНЫЙ ПУТЬ (БОЛЕЕ КРУПНЫЙ ПРИМЕР)

На этом рисунке изображены аугментальные пути, построенные с помощью алгоритма кратчайшего пути для эвклидовой сети со случайными весами, представленной на рис. 22.22, с ребрами остовного дерева поиска на графе (серого цвета). В этом случае рассматриваемый алгоритм обладает намного меньшим быстродействием, чем алгоритм вычисления максимальной пропускной способности, изображенный на рис. 22.23, в силу того, что он требует большее количество аугментальных путей (показаны первые 12 путей из общего числа 37 путей) и что остовные деревья оказываются больших размеров (обычно они содержат почти все вершины).

В разделе 22.3 будет показано, что никакой алгоритм вычисления аугментальных путей не может показывать линейную производительность в худшем случае, и будет представлено другое обобщенное семейство алгоритмов, которые обеспечивают упомянутую линейность.

Упражнения

22.19. Покажите в стиле рис. 22.13 столько различных последовательностей аугментальных путей, сколько вы их сможете найти в транспортной сети, показанной на рис. 22.10.

22.20. Покажите в стиле рис. 22.15 все сечения транспортной сети, показанной на рис. 22.10, их разделяющие множества и пропускные способности.

▷ **22.21**. Найдите минимальное сечение в транспортной сети, показанной на рис. 22.11.

○ **22.22**. Предположим, что в некоторой транспортной сети достигнут баланс пропускных способностей (для каждого внутреннего узла общая пропускная способность входящих ребер равна общей пропускной способности исходящих ребер). Использует ли когда-нибудь алгоритм Форда-Фалкерсона обратные ребра? Докажите, что использует, либо приведите встречный пример.

22.23. Определите максимальный поток для транспортной сети, представленной на рис. 22.5, в которой, по меньшей мере, величина одного потока, не является целым числом.

▷ **22.24**. Разработайте реализацию алгоритма Форда-Фалкерсона, который использует обобщенную очередь вместо очереди с приоритетами (см. раздел 18.8).

▷ **22.25**. Докажите, что число аугментальных путей, необходимых для реализации алгоритма Форда-Фалкерсона, есть не более чем в V раз уменьшенное целое число, которое превышает отношение пропускной способности наибольшего ребра к пропускной способности наименьшего ребра.

22.26. Докажите следующее свойство *нижней границы* линейного времени задачи поиска максимального потока: покажите, что для любых значений V и E любой алгоритм вычисления максимального потока должен проверить каждое ребро в некоторой сети с V вершинами и E ребрами.

▷ **22.27**. Представьте пример сети, подобной изображенной на рис. 22.19, для которой алгоритм поиска кратчайшего аугментального пути обладает поведением в худшем случае подобным представленному на рис. 22.19.

22.28. Постройте представление в виде списков смежных вершин сети, изображенной на рис. 22.19, для которой разработанная нами реализация поиска с применением стека (программа 22.3, использующая стек для обобщенной очереди), как показано на рис. 22.19, обеспечивает режим худшего случая.

22.29. Покажите в стиле рис. 22.16 транспортную и остаточную сеть после построения каждого аугментального пути в условиях, когда используется алгоритм поиска кратчайшего аугментального пути для определения максимального потока в транспортной сети, показанной на рис. 22.10. Включите также деревья поиска на графе для каждого аугментального пути. В тех случаях, когда возможен выбор более одного пути, покажите тот из них, который был выбран реализациями, рассмотренными в этом разделе.

22.30. Выполните упражнение 22.29, используя алгоритм построения аугментального пути с максимальной пропускной способностью.

22.31. Выполните упражнение 22.29, используя алгоритм поиска аугментального пути с применением стека.

○ **22.32**. Найдите семейство сетей, для которых алгоритм поиска максимального аугментального пути требует вычисления $2E \lg M$ аугментальных путей.

● **22.33**. Сможете ли вы упорядочить ребра таким образом, чтобы рассмотренные нами реализации тратили время, пропорциональное E, на поиск пути в условиях упражнения 22.32? При необходимости внесите изменения в свой пример с тем, чтобы достичь этой цели. Опишите представление графа в виде списков смежных вершин, построенное специально для предлагаемого вами примера.

22.34. Проведите эмпирические исследования с целью определения числа аугментальных путей и отношения времени выполнения к V для каждого из четырех алгоритмов, описанных в этом разделе, для различных видов сетей (см. упражнение 22.7–22.12).

22.35. Разработайте и протестируйте реализацию метода аугментальных путей, который использует эвристику кратчайшего пути исток-сток эвклидовой сети из раздела 21.5.

22.36. Разработайте и протестируйте реализацию метода аугментальных путей, основанного на альтернативно растущих деревьях поиска с корнями в истоке и стоке (см. упражнения 21.35 и 21.75).

● **22.37**. Реализация программы 22.3 останавливает поиск на графе, как только он находит первый аугментальный путь из истока в сток, производит наращивание потока, а затем инициирует поиск с самого начала. Другой способ заключается в том, что она продолжает поиск и находит другой путь. Этот процесс продолжается до тех пор, пока все вершины не будут помечены. Разработайте и протестируйте второй подход.

22.38. Разработайте и протестируйте метод аугментального пути, который использует пути, не относящиеся к категории простых.

▷ **22.39**. Предложите последовательность простых аугментальных путей, которые создают поток в цикле в сети, показанной на рис. 22.11.

○ **22.40**. Приведите пример, показывающий, что не все максимальные потоки могут быть получены методом наращивания потоков вдоль некоторой последовательности простых путей из истока в сток, начиная с пустой сети.

22.41. [Габов] Разработайте реализацию построения максимального потока, которая использует $m = \lg M$ этапов, при этом i-й этап решает задачу о максимальном потоке, используя первые i разрядов значений пропускных способностей. Начните с нулевого потока в любой точке сети; далее, по завершении первого этапа, выполните инициализацию потока удвоенной величиной потока, вычисленного на предыдущем этапе. Проведите эмпирические исследования на сетях различных видов (см. упражнения 22.7–22.12) и сравните этот метод с базовыми методами.

● **22.42**. Докажите, что время выполнения алгоритма, описанного в упражнении 22.41, есть $O(VE \lg M)$.

22.43. Проведите эксперименты с гибридным методом, который в начале использует один метод вычисления аугментальных путей, а затем переключается на другой метод вычисления аугментальных путей и использует его вплоть до завершения решения за-

дачи (частью которой является выбор подходящего критерия, чтобы определить момент переключения с одного метода на другой). Проведите эмпирические исследования на сетях различных видов (см. упражнения 22.7–22.12), выполняя более подробные исследования тех из них, которые показывают лучшие результаты.

22.44. Проведите эксперименты с гибридными методами, по условиям которых производится переключение между двумя или большим числом различных методов вычисления аугментальных путей. Проведите эмпирические исследования на сетях различных видов (см. упражнения 22.7–22.12), выполняя более подробные исследования тех из них, которые показывают лучшие результаты.

○ **22.45**. Проведите эксперименты с гибридными методами, по условиям которых производится случайный выбор из двух или большего числа различных методов вычисления аугментальных путей. Проведите эмпирические исследования на сетях различных видов (см. упражнения 22.7–22.12), и сравните эти гибридные методы с базовыми методами, выполняя более подробные исследования тех из них, которые показывают лучшие результаты.

○ **22.46**. Напишите клиентскую функцию для транспортной сети, которая, при заданном целом числе c, находит ребро, увеличение пропускной способности которого в c раз увеличивает максимальный поток до максимальной величины. Разрабатываемая функция может полагать, что клиентская программа уже вычислила максимальный поток с помощью функции **MAXFLOW**.

●● **22.47**. Предположим, что задан минимально загруженный сечение некоторой сети. Облегчает ли эта информация решение задачи вычисления максимального потока? Разработайте алгоритм, который использует заданный минимально загруженный сечение для существенного ускорения поиска аугментальных путей с максимальными пропускными способностями.

● **22.48**. Напишите клиентскую программу, которая выполняет анимацию динамики алгоритмов поиска аугментальных путей. Эта программа должна создавать динамические графические изображения, подобные рис. 22.17 и другим рисункам из этого раздела (см. упражнения 17.55 17.59). Протестируйте полученную реализацию на эвклидовых сетях из числа тех, описание которых даны в упражнениях 22.7–22.12.

22.3. Алгоритмы определения максимальных потоков методом выталкивания превосходящего потока

В этом разделе мы рассмотрим другой подход к решению задачи о максимальном потоке. Используя обобщенный метод, известный как метод *выталкивания превосходящего потока (preflow-push)*, мы будем постепенно наращивать поток по ребрам, исходящим из вершин, в которых приток превышает истечение. Метод выталкивания превосходящего потока был разработан Гольдбергом и Тарьяном в 1986 г. на основе ранее открытых алгоритмов. Алгоритм получил широкое распространение благодаря своей простоте, гибкости и эффективности.

В соответствии с соглашениями, принятыми в разделе 22.1, поток должен удовлетворять условию равновесия, согласно которому поток, вытекающий из источника, должен быть равен потоку, поступающему в сток, а приток в каждом внутреннем узле равен от-

току. Мы считаем такой поток *осуществимым (feasible)* потоком. Алгоритм вычисления аугментальных путей предполагает наличие осуществимых потоков: он наращивает поток вдоль аугментальных путей до тех пор, пока не будет достигнут максимальный поток. В отличие от него, алгоритмы выталкивания превосходящих потоков, которые мы будем изучать в этом разделе, допускает потоки, не относящиеся к категории осуществимых, поскольку в некоторых вершинах приток превышает отток: они проталкивают поток через эти вершины, пока поток не станет осуществимым (т.е. пока не установится баланс в каждой вершине).

Определение 22.5. *В транспортных сетях* **превосходящий поток (preflow)** *представляет собой множество положительных потоков в ребрах, удовлетворяющих условию, согласно которому поток в каждом ребре не превосходит пропускную способность этого ребра и что в каждой внутренней вершине приток не меньше оттока.* **Активная (active)** *вершина есть внутренняя вершина, приток в которой больше оттока (по соглашению исток и сток не могут быть активными).*

Назовем разницу между притоком активной вершины и ее оттоком *избытком (excess)* этой вершины. Чтобы изменить множество активных вершин, выбираем одну из них и выталкиваем его избыток в исходящее ребро, если же пропускная способность этого ребра не позволяет сделать это, то излишек выталкивается по входящему ребру. Если величина выталкиваемого потока выравнивает приток и истечение, вершина перестает быть активной; поток, вытолкнутый в другую вершину, может привести ее в активное состояние. Метод выталкивания превосходящего потока представляет собой систематический способ многократного выталкивания излишнего потока из активных вершин, при этом процесс выталкивания завершается получением максимального потока, когда ни одна из вершин не остается активной. Мы помещаем активные вершины в обобщенную очередь. Что касается метода вычисления аугментальных путей, то такое решение позволяет получить обобщенный алгоритм, который обхватывает целое семейство специальных алгоритмов.

Рисунок 22.25 представляет собой небольшой пример, иллюстрирующий базовые операции, используемые алгоритмами вытеснения избыточных потоков в рамках модельного представления, которое мы уже использовали и согласно которому мы полагаем, что поток может перемещаться вниз по странице. Мы либо выталкиваем избыточный поток из активной вершины вниз по исходящему ребру, либо воображаем, что активная вершина на время перемещается вверх, так что мы получаем возможность протолкнуть избыточный поток теперь уже вниз по входящему ребру.

Рисунок 22.26 являет собой пример, который показывает, что подход, предусматривающий вытеснение превосходящего потока, может оказаться более предпочтительным по сравнению с методом вычисления аугментальных путей. В рассматриваемой сети любой метод вычисления аугментальных путей последовательно многократно добавляет в длинный путь крошечные порции потока, медленно наполняя ребра этого пути до тех пор, пока не будет достигнут максимальный поток. В отличие от этого, метод вытеснения превосходящего потока наполняет ребра, составляющие этого длинного пути при первом проходе, затем распределяет этот поток непосредственно в сток без необходимости проходить этот длинный путь второй раз.

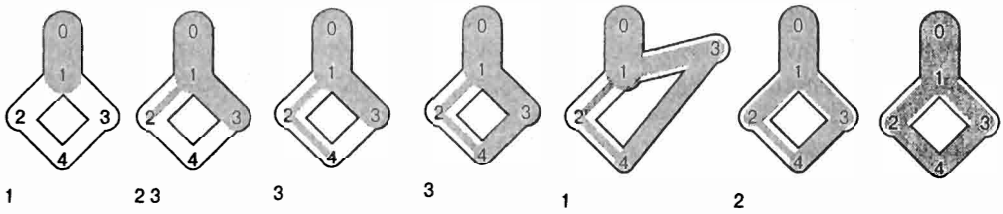


РИСУНОК 22.25. ПРИМЕР ВЫТЕСНЕНИЯ ПРЕВОСХОДЯЩЕГО ПОТОКА

В условиях алгоритма вытеснения превосходящего потока мы ведем список активных узлов, в которых входящие потоки превосходят исходящие (показаны под диаграммой каждой сети). Одной из версий этого алгоритма есть цикл, который активный узел из этого списка и проталкивает поток по исходящим ребрам до тех пор, пока узел не перестанет быть активным, возможно, создавая при этом другие активные узлы. В рассматриваемом примере мы проталкиваем поток вдоль ребра ***0-1****, в результате чего узел* ***1*** *становится активным. Затем мы проталкиваем поток по ребрам* ***1-2*** *и* ***1-3****, благодаря чему* ***1*** *перестает быть активным, зато активизируются узлы* ***2*** *и* ***3****. Далее мы проталкиваем поток вдоль ребра* ***2-4****, в результате чего* ***2*** *становится неактивным узлом. В то же время* ***3-4*** *не имеет достаточной пропускной способности, чтобы мы могли протолкнуть поток по ребру* ***3-1*** *обратно в вершину* ***1****, которая снова активизируется. Теперь мы можем отправить поток по ребру* ***1-2****, а затем по* ***2-4****, в результате чего все узлы становятся неактивными, и мы получаем максимальный поток.*

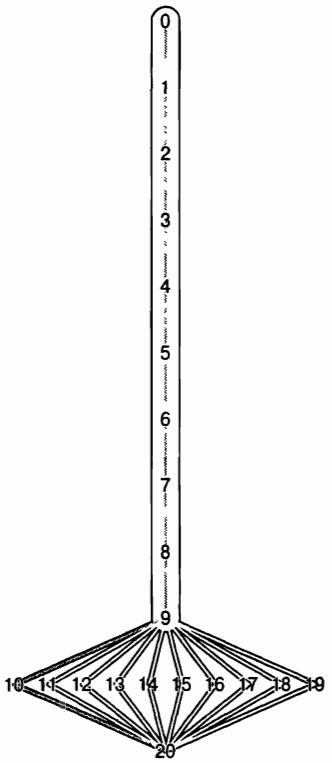


РИСУНОК 22.26. ХУДШИЙ СЛУЧАЙ АЛГОРИТМА ФОРДА-ФАЛКЕРСОНА

Эта сеть представляет семейство сетей с V вершинами, для которых любой алгоритм вычисления аугментальных путей требует V/2 длиной V/2 (поскольку каждый аугментальный путь должен включать длинный вертикальный путь), для чего потребуется время, пропорциональное V^2. Алгоритмы выталкивания превосходящего потока находят максимальный поток за линейное время.

Так же как и в случае алгоритма аугментальных путей, мы воспользуемся остаточными сетями (см. определение 22.4), чтобы отслеживать ребра, через которые можно проталкивать поток. Каждое ребро остаточной сети представляет собой место, в котором можно проталкивать поток. Если ребро остаточной сети ведет в том же направлении, что и соответствующее ему ребро транспортной сети, мы увеличиваем поток; если оно ведет в противоположном направлении, то мы уменьшаем поток. Если увеличение потока насыщает ребро или уменьшение потока опорожняет ребро, то соответствующее ребро в остаточной сети исчезает. В алгоритмах выталкивания превосходящего потока мы используем дополнительный механизм, который облегчают принятие решений относительно того, какие ребра в остаточной сети будут нам полезны в процессе устранения активных вершин.

Определение 22.6. Функция высоты (height function) *заданного потока в транспортной сети представляет собой множество неотрицательных весов вершин* $h(0)...h(V-1)$ *такое, что* $h(t) = 0$ *для стока* **t** *и* $h(u) \le h(v) + 1$ *для каждого ребра* **u-v** в *остаточной сети этого потока.* **Подходящее ребро (eligible edge)** *есть ребро* **u-v** *остаточной сети* с $h(u) = h(v) + 1$.

Тривиальная функция высоты, для которой не существует подходящих ребер, выглядит как $h(0) = h(1) = ...= h(V-1) = 0$. Если мы установим $h(s) = 1$, то любое ребро, исходящее из истока и пропускающее через себя поток, соответствует подходящему ребру в остаточной сети.

Мы дадим более интересное определение функции высоты, присваивая каждой вершине ее кратчайший путь до стока (ее расстояние до любого дерева BFS для обратной сети с корнем в **t**, как показано на рис. 22.27). Эта функция высоты верна, поскольку $h(t) = 0$, и для любой пары вершин **u** и **v**, соединенных ребром **u-v**, любой кратчайший путь, ведущий в **t** и начинающийся ребром **u-v**, имеет длину $h(v) + 1$; таким образом, длина кратчайшего пути из **u** в **t**, или $h(u)$, должна быть меньше или равной этому значению. Эта функция играет особую роль, поскольку она помещает каждую вершину на максимально возможную высоту. Возвращаясь назад, мы видим, что **t** должна находиться на высоте **0**; единственными вершинами, которые могут находиться на высоте **1** являются те вершины, которым соответствуют ребра, направленные в **t** в остаточной сети; на высоте **2** могут находиться только вершины, имеющие ребра, направленные в вершины, которые могут быть на высоте **1**, и т.д.

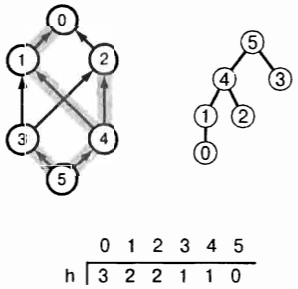


РИСУНОК 22.27. ИСХОДНАЯ ФУНКЦИЯ ВЫСОТЫ

*Изображенное справа дерево есть дерево BFS с корнем в вершине **5** обращенной сети, рассматриваемой в качестве примера (слева). Массив **h**, индексированный именами вершин, содержит расстояние каждой вершины сети до корня и представляет собой эффективную функцию высоты: для каждого ребра **u-v** в сети **h[u]** меньше или равно **h[v] + 1**.*

Свойство 22.9. *Для любого потока и ассоциированной с ним функцией высоты высота вершины не превосходит длины самого короткого пути из этой вершины в сток в остаточной сети.*

Доказательство: Для любой заданной вершины u пусть d есть длина кратчайшего пути из u в t, и пусть $u = u_1$, u_2,..., $u_d = t$ является кратчайшим путем. Тогда:

$$\begin{aligned} h(u) = h(u_1) &\le h(u_2) + 1 \\ &\le h(u_3) + 2 \\ &\cdot \\ &\cdot \\ &\cdot \\ &\le h(u_d) + d = h(t) + d = d \ \blacksquare \end{aligned}$$

Содержательно функции высоты означают следующее: если высота активного узла меньше, чем высота источника, возможно, что существует некоторый способ протолкнуть поток из этого узла в сток; когда высота активного узла превышает высоту источника, мы знаем, что избыток этого узла нужно вытолкнуть обратно в исток. Чтобы установить этот факт, мы пересмотрим наше толкование свойства 22.9, в котором длина кратчайшего пути рассматривается как верхняя граница высоты; вместо этого рассматриваем высоту как нижнюю границу длины кратчайшего пути.

Следствие. *Если высота вершины больше V, то не существует пути из этой вершины в сток остаточной сети.*

Доказательство: Если существует путь из этой вершины в сток, то из свойства 22.9 следует, что длина пути будет больше V, но этого быть не может, поскольку в сети имеется всего V вершин. ■

Теперь, когда мы понимаем эти базовые механизмы, нетрудно дать описание обобщенного алгоритма вытеснения избыточного потока. Мы начнем с выбора любой функции высоты и назначим нулевой поток всем ребрам, за исключением тех, которые присоединены к истоку, которые мы заполняем до пропускной способности. Затем мы повторяем следующее действие до тех пор, пока не останется ни одной активной вершины:

Выбираем активную вершину. Проталкиваем поток через некоторое подходящее ребро и покидаем эту вершину (если есть куда). Если нет таких ребер, увеличиваем высоту вершины.

Мы не конкретизируем, какую выбираем исходную функцию высоты, как производим выбор активной вершины, как выбираем подходящее ребро или какую величину потока выталкиваем. Мы будем называть этот обобщенный метод как *реберный (edge-based)* алгоритм выталкивания превосходящего потока.

Этот алгоритм зависит от выбора функции высоты, позволяющей определить подходящие ребра. Мы также используем функцию высоты как для доказательства того, что этот алгоритм вычисляет максимальный поток, так и для анализа производительности. Поэтому необходимо обеспечить, чтобы функция высоты оставалась пригодной на протяжении всего времени выполнения алгоритма.

Свойство 22.10. *Алгоритм выталкивания превосходящего потока, основанный на реберном представлении графа, сохраняет функциональную достоверность функции высоты.*

Доказательство: Мы увеличиваем значение $h(u)$ только в том случае, когда отсутствует ребро **u-v** с $h(u) = h(v) + 1$. То есть, $h(u) < h(v) + 1$ для всех ребер **u-v** перед увеличением $h(u)$ на 1, так что после него справедливо $h(u) \le h(v) + 1$. Для любого входящего ребра **w-u** увеличение $h(u)$ на 1 наверняка сохранит истинность неравенства $h(w) \le h(u) + 1$. Увеличение $h(u)$ не затрагивает неравенств, соответствующих любым другим ребрам, и мы никогда не увеличиваем $h(t)$ (или $h(s)$). Оба эти вывода помогают установить окончательный результат. ■

Весь излишек потока истекает из истока. С содержательной точки зрения, обобщенный алгоритм выталкивания превосходящего потока делает попытку вытолкнуть избыточный поток в сток, и если это ему не удается, он в конечном итоге проталкивает избыточный поток обратно в исток. Он принимает такую модель поведения, поскольку узлы с избытком потока всегда остаются связанными с истоком в остаточной сети.

Свойство 22.11. *Во время выполнения алгоритма выталкивания превосходящего потока на транспортной сети существует (ориентированный) путь в остаточной сети, соответствующей этой транспортной сети, существует (ориентированный) путь из каждой активной вершины в исток, и в то же время не существуют (ориентированные) пути из истока в сток в остаточной сети.*

Доказательство: Выполняется методом индукции. Вначале поток протекает только по ребрам, исходящим из истока, которые заполняются до пропускной способности, так что в этот момент активными вершинами в сети являются вершины назначения этих ребер и никакие другие. Поскольку указанные ребра насыщены до пропускной способности, в остаточной сети существует ребро, ведущее из каждой из этих вершин в исток, а в остаточной сети нет ребер, идущих из истока. Отсюда следует, что указанное свойство характерно для первоначального потока.

Исток достижим из каждой активной вершины, поскольку единственный способ увеличения множества активных вершин состоит в проталкивании потока из активной вершины по подходящему ребру. Этой операции в остаточной сети соответствует ребро, идущее из принимающей вершины в активную вершину, из которой, по индуктивному предположению, достижим исток.

В самом начале никакой узел остаточной сети недоступен из истока. Впервые некоторый узел **u** становится доступным из истока, когда поток выталкивается в обратном направлении по ребру **u-s** (что обусловливает добавление ребра **s-u** в остаточную сеть). Но это становится возможным только когда $h(u)$ становится больше $h(s)$, что в свою очередь может случиться только если значение $h(u)$ увеличится, поскольку в остаточной сети нет ребер, ведущих в вершины с меньшей высотой. Поскольку высота стока всегда равна 0, он не может быть достижим из истока. ■

Следствие. *Во время выполнения алгоритма выталкивания превосходящего потока высота вершин всегда меньше* $2V$.

Доказательство: Достаточно рассмотреть только активные вершины, поскольку высота неактивных вершин имеет то же значение либо на 1 больше значения, которое вершина имела, когда последний раз была активной. В соответствии с рассуждениями, какие мы использовали при доказательстве свойства 22.9, из наличия пути из заданной активной в исток следует, что высота этой вершины в максимум на $V-2$ больше высоты источника (этот путь не может проходить через **t**). Высота истока никогда не меняется и первоначально она не может быть больше V. Следовательно, высота активных вершин не может превышать значения $2V-2$, и ни одна из вершин не может иметь высоты $2V$ и больше. ■

Формулировка обобщенного алгоритма выталкивания превосходящего потока проста, его реализация не сопряжена с трудностями. Возможно, пока непонятно, как он вычисляет максимальный поток. Понятия функции высоты является ключевым фактором, чтобы установить, что он решает и эту задачу.

Свойство 22.12. *Алгоритм выталкивания превосходящего потока вычисляет максимальный поток.*

Доказательство: Прежде всего, мы должны доказать, что алгоритм останавливается. Должен наступить момент, когда не останется ни одной активной точки. Как только мы вытесним все избыточные потоки из некоторой вершины, эта вершина не может снова стать активной, пока какая-то часть этого потока не будет вытолкнута назад, и этот вытолкнутый поток может иметь место только тогда, когда высота этой вершины возрастет. Если имеется последовательность активных вершин неограниченной длины, одна из вершин должна появиться в этой последовательности неограниченное число раз, а это может случиться только в том случае, когда ее высота возрастает неограниченно, что противоречит свойству 22.9.

Если активных вершин нет, то поток осуществим. Так как согласно свойству 22.11, в остаточной сети путь от истока в сток отсутствует, то рассуждения, использованные при доказательстве свойства 22.5, приводят нас к заключению, что поток в сети представляет собой максимальный поток. ■

Можно уточнить доказательство этого алгоритма в части его останова и показать, что он остановится в худшем случае по истечении промежутка времени $O(V^2E)$. Все детали этого доказательства мы оставляем читателям на самостоятельную проработку (см. упражнения 22.66 и 22.67), а здесь отдадим предпочтение более простому доказательству 22.13, которое относится к более простой версии алгоритма. В частности, в основу рассматриваемых нами реализаций положим более конкретные инструкции, касающиеся итераций:

Выберите активную вершину. Увеличьте поток в подходящем ребре, исходящем из этой вершины (если это возможно), продолжая этот процесс до тех пор, пока или вершина не перестанет быть активной либо не останется ни одного подходящего ребра. В последнем случае увеличьте высоту вершины.

Другими словами, как только будет выбрана вершина, мы выталкиваем из нее как можно больший поток. Если мы выйдем в вершину, в которой все еще имеет место избыток потока, но в то же время не осталось ни одного подходящего ребра, мы увеличиваем высоту этой вершины. Мы будем называть этот метод *вершинным (vertex-based)* алгоритмом выталкивания превосходящего потока. Это специальный случай реберного обобщенного алгоритма, в условиях которого мы продолжаем выбирать одну и ту же вершину до тех пор, пока она не перестанет быть активной, либо пока не перепробуем все подходящие ребра, исходящие из нее. Доказательство свойства 22.12 сохраняет корректность применительно к любой реализации реберного обобщенного алгоритма, отсюда немедленно следует, что вершинный алгоритм вычисляет максимальный поток.

Программа 22.5 представляет собой реализацию обобщенного вершинного алгоритма, который использует обобщенную очередь активных вершин. Это непосредственная реализация только что описанного метода, она представляет собой семейство алгоритмов, которые отличаются только исходными функциями высоты (см. например, упражнение 22.52) и реализацией АДТ обобщенной очереди. Такая реализация построена в предположении, что обобщенная очередь блокирует хранение дубликатов вершин; в противном случае мы можем добавить в программу 22.5 код, запрещающий прием дубликатов в очередь (см. упражнения 22.61 и 22.62).

Возможно, наиболее простой структурой данных для хранения активных вершин является очередь FIFO. На рис. 22.28 показано, как работает этот алгоритм на демонстрационном примере сети. Из рисунка видно, что последовательность активных вершин удобно разбить на последовательность *этапов*, при этом этап составляют все вершины, которые находились в очереди после того, как все вершины, входившие в состав предыдущего этапа, были обработаны. Такого рода разбиение помогает нам ограничить общее время выполнения алгоритма.

Свойство 22.13. Время выполнения реализации алгоритма выталкивания превосходящего потока на базе очереди FIFO пропорционально V^2E.

Доказательство: Мы ограничим число этапов, воспользовавшись для этой цели потенциальной функцией. Рассуждения, используемые для доказательства, представляют собой простой пример применения мощных средств анализа алгоритмов, а используемые структуры данных будут более подробно изучаться в части 8.

Определим числовое значение величины ϕ равным 0, если в сети нет активных вершин, и равным максимальной высоте активных вершин в противном случае, затем рассмотрим, как отражается выполнение каждого этапа на величине ϕ. Пусть $h_0(s)$ есть начальная высота истока. Вначале $\phi = h_0(s)$; по завершении $\phi = 0$.

Прежде всего, отметим, что число этапов, на которых высота некоторых вершин возрастает, не превосходит значения $2V^2 - h_0(s)$, поскольку высота каждой из V вершин, согласно свойству 22.11, может быть увеличена максимум до значения $2V$. Поскольку ϕ может возрастать только тогда, когда увеличивается высота некоторых вершин, число этапов, когда ϕ возрастает, не может быть больше, чем $2V^2 - h_0(s)$.

Если, однако, на каком-либо этапе высота ни одной из вершин не будет увеличена, то значение ϕ должно быть уменьшено, по меньшей мере, на 1, поскольку результат этапа заключается в том, чтобы протолкнуть весь избыточный поток из каждой активной вершины в вершины с меньшей высотой.

Принимая во внимание все эти факты, приходим к заключению, что число этапов не должно быть больше $4V^2$: значение ϕ в начале есть $h_0(s)$ и может быть увеличено максимум $2V^2 - h_0(s)$ раз, т.е., оно может быть уменьшено максимум $2V^2$ раз. Худший случай каждого этапа возникает тогда, когда все вершины находятся в очереди и все их ребра подвергаются проверке, что соответствует граничному значению, установленному для общего времени выполнения.

Эта граница плотная. На рис. 22.29 изображено семейство транспортных сетей, для которых число этапов, использованных алгоритмом выталкивания избыточного потока пропорционально V^2. ■

Поскольку предлагаемые нами реализации поддерживают неявное представление остаточной сети, они проверяют ребра, исходящие из вершины, даже если эти ребра не содержатся в остаточной сети (чтобы проверить, находятся они там или нет). Можно показать, что границу V^2E, устанавливаемую свойством 22.13, можно уменьшить до V^3 для реализаций за счет поддержки явного представления остаточной сети. Несмотря на то что эта теоретическая граница есть наименьшая из тех, с которыми нам приходилось сталкиваться при решении задачи вычисления максимального потока, это достижение, по-видимому, не заслуживает нашего внимания, особенно в случае разреженных графов, с которыми мы часто имеем дело на практике (см. 22.63–22.65).

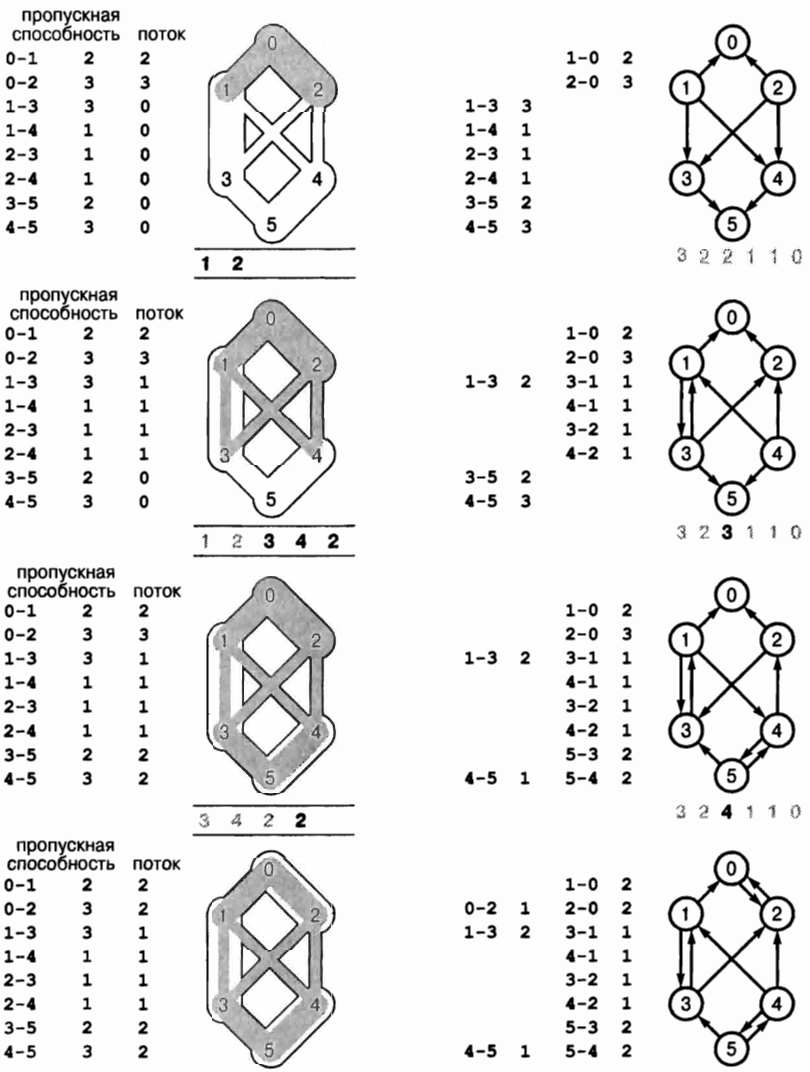


РИСУНОК 22.28. ОСТАТОЧНАЯ СЕТЬ (ОЧЕРЕДЬ FIFO ДЛЯ ВЫТАЛКИВАНИЯ ИЗБЫТОЧНОГО ПОТОКА)

На этом рисунке показаны транспортные сети (слева) и остаточные сети (справа) для каждого этапа алгоритма выталкивания превосходящего потока на базе очереди FIFO, работающего на демонстрационном примере. Содержимое очередей показано под диаграммами транспортных сетей, а метки расстояний — под диаграммами остаточных сетей. На начальном этапе мы проталкиваем поток по ребрам ***0-1*** *и* ***0-2****, переводя тем самым вершины* ***1*** *и* ***2*** *в категорию активных. На втором этапе мы проталкиваем поток через эти две вершины в* ***3*** *и* ***4****, благодаря чему они становятся активными, а вершина* ***1*** *перестает быть активной (вершина* ***2*** *остается активной, а метка ее расстояния увеличивается). На третьем этапе мы проталкиваем поток через вершины* ***3*** *и* ***4*** *в вершину* ***5****, и это делает их неактивными (****2*** *все еще остается активной, ее метка расстояния снова увеличивается). На четвертом этапе* ***2*** *остается единственным активным узлом, а ребро* ***2-0*** *становится приемлемым, поскольку метка расстояния возрастает и одна единица потока проталкивается обратно вдоль* ***2-0****, после чего вычисления завершаются.*

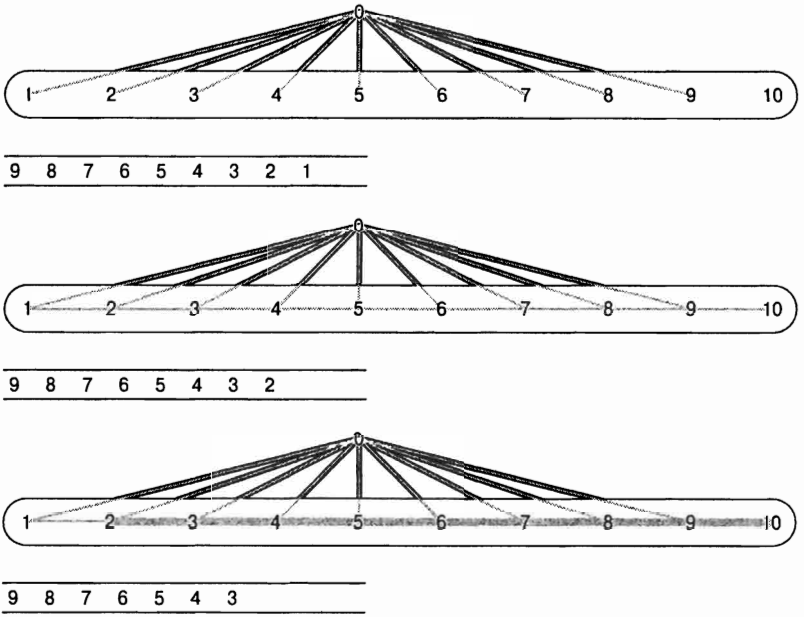


РИСУНОК 22.29. ХУДШИЙ СЛУЧАЙ ВЫПОЛНЕНИЯ АЛГОРИТМОМ ВЫТАЛКИВАНИЯ ИЗБЫТОЧНОГО ПОТОКА, ПОСТРОЕННОГО НА БАЗЕ ОЧЕРЕДИ FIFO

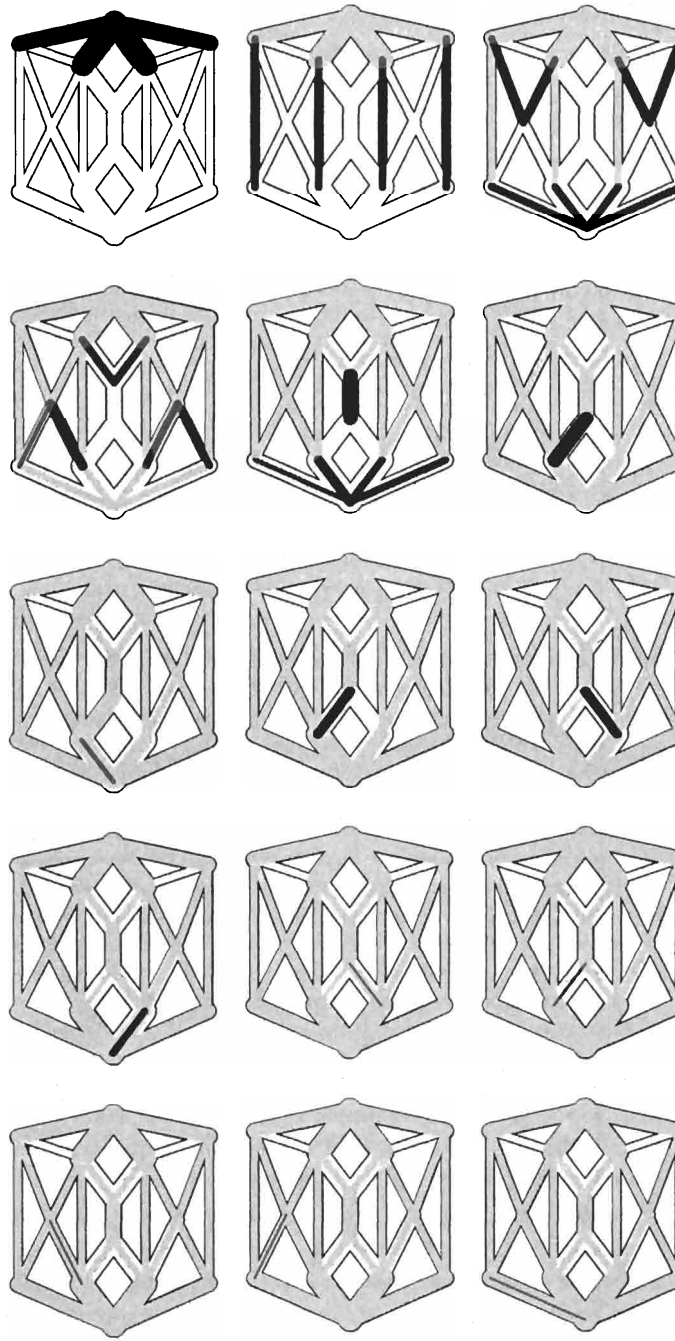
*Данная сеть представляет семейство сетей с V вершинами, таких, что общее время прогона алгоритма выталкивания избыточного потока пропорционально V^2. Она состоит ребра с пропускной способностью, равной единице, исходящих из истока (вершина **0**) и горизонтальных ребер с пропускной способностью v − 2, ведущих слева направо в сток (вершина **10**). На начальном этапе алгоритма выталкивания избыточного потока (сверху) мы выталкиваем одну единицу потока из каждого ребра, идущего из истока, в результате чего все вершины становятся активными, за исключением истока и стока. В стандартном представлении графа в виде списков смежных вершин они попадают в очередь FIFO активных вершин в обратном порядке, как показано в строке под диаграммой сети. На втором этапе (в центре) мы выталкиваем единицу потока из вершины **9** в **10**, переводя **9** в неактивное состояние (временно), затем выталкиваем единицу потока из вершины **7** в **8**, переводя **8** в активное состояние и т.д. Только вершина **1** остается неактивной. На третьем этапе (внизу) мы проходим через аналогичный процесс, чтобы сделать вершину **2** неактивной. Этот процесс продолжается на протяжении V − 2 этапов.*

Следует отметить, что границы худшего случая дают заниженную оценку, и в силу этого обстоятельства не всегда годятся для прогнозирования производительности алгоритма на реальных сетях (хотя расхождение между ними не настолько велико, какое мы находим в случае алгоритмов определения аугментальных путей). Например, алгоритм FIFO находит поток в сети, представленной на рис. 22.30 за 15 этапов, в то время как оценка, полученная на основании свойства 22.13, утверждает, что число этапов для его вычисления не превышает 182.

Чтобы повысить производительность, мы можем попытаться использовать в программе 22.5 стек, рандомизированную очередь или любой другой вид обобщенной очереди. Один из подходов, хорошо зарекомендовавших себя на практике, заключается в такой реализации обобщенной очереди, чтобы функция могла **GQGet** возвращать максимальную активную вершину.

РИСУНОК 22.30. АЛГОРИТМ ВЫТАЛКИВАНИЯ ПРЕВОСХОДЯЩЕГО ПОТОКА (С ОЧЕРЕДЬЮ FIFO)

Приведенная последовательность диаграмм показывает, как реализация метода выталкивания превосходящего потока находит максимальный поток на демонстрационном примере сети. Он выполняет обработку в несколько этапов. Сначала он проталкивает максимально возможную величину потока из истока по исходящим из него ребрам (диаграмма слева вверху). Затем выполняется проталкивание потока из каждого такого узла и продолжение этой процедуры до тех пор, пока все узлы не придут в равновесие.



Мы будем называть этот метод алгоритмом вычисления максимального потока *методом выталкивания превосходящего потока из вершины с максимальной высотой (highest-vertex-preflow-push)*. Мы можем реализовать данную стратегию за счет использования очереди по приоритету, однако существует возможность воспользоваться конкретными особенностями высот, чтобы обеспечить выполнение операций на обобщенной очереди за постоянное время. Было доказано, что временная граница для худшего случая выполнения этого алгоритма есть $V^2\sqrt{E}$ (для разреженных графов она принимает вид $V^{5/2}$) (см. *раздел ссылок*); по обыкновению, эта оценка носит пессимистичный характер. Было предложено и множество других вариантов выталкивания превосходящего потока, некоторые из них понижают временную границу для худшего случая до значения, близкого к VE (см. раздел ссылок).

Программа 22.5. Реализация вычислений максимального потока методом выталкивания превосходящего потока

Программная реализация обобщенного вершинного алгоритма выталкивания превосходящего потока для вычисления максимального потока использует обобщенную очередь, не принимающую дубликаты активных вершин. Вектор **wt** содержит избыточный поток в каждой вершине, благодаря чему неявно определяет множество активных вершин. Предполагается, что вершина **s** первоначально активна, но никогда не возвращается в очередь, а **t** никогда не бывает активной. Главный цикл выбирает активную вершину **v**, затем проталкивает поток через каждое инцидентное ей подходящее ребро (вносит при необходимости вершины, принимающие потоки, в список активных вершин) до тех пор, пока либо **v** перестает быть активной, либо будут просмотрены все подходящие ребра. В последнем случае высота **v** увеличивается, а сама она отправляется в очередь.

```
template <class Graph, class Edge> class MAXFLOW
{ const Graph &G;
   int s, t;
   vector<int> h, wt;
   void initheights();
public:
   MAXFLOW(const Graph &G, int s, int t) : G(G),
      s(s), t(t), h(G.V()), wt(G.V(), 0)
   { initheights();
      GQ gQ(G.V());
      gQ.put(s); wt[t] = -(wt[s] = M*G.V());
      while (!gQ.empty())
      { int v = gQ.get();
         typename Graph::adjIterator A(G, v);
         for (Edge* e = A.beg(); !A.end(); e = A.nxt())
          { int w = e->other(v);
             int cap = e->capRto(w);
             int P = cap < wt[v] ? cap : wt[v];
             if (P > 0 && v == s || h[v] == h[w]+1)
              { e->addflowRto(w, P);
                 wt[v] -= P; wt[w] += P;
                 if ((w != s) && (w != t)) gQ.put(w); }
          }
         if (v != s && v != t && wt[v] > 0)
          { h[v]++; gQ.put(v); }
      }
   }
};
```

В таблицу 22.2 сведены результаты, показанные алгоритмами выталкивания превосходящего потока, которые соответствуют приведенным в таблице 22.1 результатам алгоритмов, основанных на вычислении аугментальных путей, для двух моделей сетей из раздела 22.2. Эти эксперименты свидетельствуют о том, что различия в производительности различных алгоритмов выталкивания намного меньше, чем те, что мы наблюдали при использовании разных методов аугментальных путей.

Таблица 22.2. Эмпирические исследования алгоритмов выталкивания превосходящих потоков

В этой таблице приводятся значения параметров (число вершин, в которых величина потока увеличивалась, и число узлов из списков смежных с ними вершин, задействованных в этих операциях) для различных алгоритмов выталкивания превосходящих потоков в транспортной сети на примере эвклидовой сети с близкими связями. В первом случае пропускные способности ребер принимают случайные значения, а максимальный поток равен 286, во втором случае пропускные способности ребер принимают единичные значения, а максимальный поток равен 6. Различия в результатах применения этих методов к обоим типам сети минимальны. Для варианта случайных значений пропускных способностей число проверяемых ребер примерно то же, что и для варианта алгоритма вычисления аугментальных путей (см. таблицу 22.1). Для случая единичных пропускных способностей алгоритмы вычисления аугментальных путей исследуют в этих сетях значительно меньше ребер.

	Вершины	Ребра
Случайные значения пропускных способностей в диапазоне от 1 до 50		
Кратчайший путь	2450	57746
Поиск в глубину	2476	58258
Случайный поиск	2363	55470
Пропускная способность, равная 1		
Кратчайший путь	1192	28356
Поиск в глубину	1234	29040
Случайный поиск	1390	33018

Существует множество методов исследования в процессе разработки программных реализаций алгоритмов выталкивания превосходящих потоков. Мы уже обсуждали три основных варианта:

- Сравнение вершинного обобщенного алгоритма с реберным.
- Реализация обобщенной очереди.
- Первоначальная установка значений высоты.

Имеется несколько других возможностей исследований и множество различных вариантов каждой из них, достойных того, чтобы проверить их, и ведущих к появлению множества различных алгоритмов, заслуживающих подробного анализа (см., например, упражнения 22.56–22.60). Зависимость производительности алгоритмов от характеристик сети ввода приводит к дальнейшему увеличению таких возможностей.

Два обобщенных алгоритма, которые мы рассматривали выше (вычисления аугментальных путей и проталкивания превосходящих потоков), входят в число важнейших алгоритмов, если судить по тому вниманию, которое уделяется им в публикациях, посвящен-

ным методам определения максимальных потоков. Поиск более совершенных алгоритмов вычисления максимальных потоков все еще остается перспективной областью для дальнейших исследований. У исследователей появляется мотивация к разработке и исследованию новых алгоритмов и реализаций, обусловленная наличием алгоритмов решения различных практических проблем, обладающих высоким быстродействием, и возможностью существования простых линейных алгоритмов решения задачи вычисления максимального потока. До тех пор, пока такой алгоритм не появится, мы можем уверенно работать с рассмотренными выше алгоритмами и реализациями; многочисленные исследования показывают, что они обеспечивают эффективное решение широкого класса практических задач, требующих вычисления максимальных потоков.

Упражнения

▷ **22.49**. Опишите операции алгоритма выталкивания превосходящего потока в сети, пропускные способности которой находятся в равновесии.

22.50. Воспользуйтесь концепциями, приведенными в данном разделе (функции высоты, подходящие ребра, выталкивание потока через ребра), для описания алгоритмов вычисления максимального потока методом аугментальных путей.

22.51. Покажите в стиле рис. 22.28 потоки и остаточную сеть после каждой фазы, когда вы используете алгоритм выталкивания превосходящего потока с очередью FIFO для отыскания максимального потока в транспортной сети, показанной на рис. 22.10.

▷ **22.52**. Реализуйте функцию **initheights()** для программы 22.5, используя поиск в ширину, отправляясь от стока.

22.53. Выполните упражнение 22.51 для алгоритма выталкивания превосходящего потока из вершины с максимальной высотой.

○ **22.54**. Выполните модификацию программы 22.5 с целью реализации алгоритма выталкивания превосходящего потока из вершины с максимальной высотой путем построения обобщенной очереди как очереди с приоритетами. Проведите эмпирическое тестирование таким образом, чтобы у вас были основания включить в таблицу 22.2 строку для этого варианта алгоритма.

22.55. Начертите диаграмму, отображающую число активных вершин, а также число вершин и ребер в остаточной сети по мере выполнения реберного алгоритма выталкивания превосходящего потока с очередью FIFO для конкретных примеров различных видов сетей (см. упражнения 22.7–22.12).

○ **22.56**. Реализуйте обобщенный реберный алгоритм выталкивания превосходящего потока, который использует обобщенную очередь подходящих ребер. Выполните эмпирические исследования различных сетей (см. упражнения 22.7–22.12), сравнивая их с базовыми методами и проводя более подробный анализ реализаций обобщенных очередей, которые обеспечивают лучшие показатели, чем другие виды очередей.

22.57. Внесите изменения в программу 22.5 с целью периодического пересчета высот вершин, представляющие собой кратчайшие пути из соответствующей вершины в сток в остаточных сетях.

○ **22.58**. Дайте оценку идеи проталкивания избыточного потока из вершин путем его равномерного распределения по исходящим ребрам, но не путем наполнения одних до насыщения, оставляя пустыми другие.

22.59. Проведите эмпирические исследования, чтобы определить, оправданы ли для программы 22.5 вычисления кратчайших путей для определения исходных значений функции высоты, путем сравнения производительности программы при задании функции высоты этим способом для различных сетей (см. упражнения 22.7–22.12) с ее производительностью в случае, когда высоты вершины инициализированы нулями.

○ **22.60**. Проведите эксперименты с гибридными методами, предусматривающими использование различных комбинаций описанных выше идей. Проведите экспериментальные исследования на примере различных видов сетей (см. упражнения 22.7–22.12) с целью сравнения этих методов с базовыми методами, более подробно останавливаясь на вариантах, показывающих лучшие результаты.

22.61. Внести в реализацию программы 22.5 такие изменения, которые бы явно предупреждали появление дубликатов вершин в обобщенной очереди. Проведите эмпирические испытания на примере различных сетей (см. упражнения 22.7–22.12), чтобы определить, как отразятся эти изменения на значения фактического времени выполнения программы.

22.62. Как отразится разрешение принимать дубликаты вершин в обобщенную очередь на граничном значении времени прогона алгоритма в худшем случае, определяемом свойством 22.13?

22.63. Внести в реализацию программы 22.5 такие изменения, которые позволили бы поддерживать явное представление остаточной очереди.

○ **22.64**. Покажите, что граница, определяемая свойством 12.13, принимает значение $O(V^3)$ для реализации, задаваемой упражнением 22.63. *Указание*: Проведите проверку отдельных границ на число выталкиваний потока, которое соответствует удалению ребер из остаточной сети, и на число выталкиваний потока, в результате которых не появляются полные или пустые ребра.

22.65. Проведите эмпирические исследования на различных сетях (см. упражнения 22.7–22.12), чтобы определить, к каким последствиям приведет использование явных представлений остаточных сетей (см. упражнение 22.63) на фактических значениях времени прогона алгоритма.

22.66. Докажите, что число выталкиваний потоков в условиях применения обобщенного реберного алгоритма выталкивания превосходящего потока, которое соответствует удалению ребра из остаточной сети, не превышает $2VE$. Предполагается, что такая реализация поддерживает явное представление остаточной сети.

● **22.67**. Докажите, что число выталкиваний потоков в условиях применения обобщенного реберного алгоритма выталкивания превосходящего потока, которое не соответствует удалению ребра из остаточной сети, не превышает $4V^2(V+E)$. *Указание*: Воспользуйтесь суммой высот активных вершин как потенциальной функцией.

● **22.68**. Проведите эмпирические исследования для определения фактического числа проверяемых ребер и отношения времени прогона к V для различных версий алгоритма выталкивания превосходящего потока для различных сетей (см. упражнения 22.7–22.12). Рассмотрите различные алгоритмы, описанные в тексте и в предыдущих упражнениях, и остановите свое внимание на тех из них, которые зарекомендовали себя с лучшей стороны на крупных разреженных сетях. Сравните полученные вами результаты с результатами, полученными в упражнении 22.34.

- **22.69**. Напишите клиентскую программу, которая выполняет графическую анимацию динамики алгоритма выталкивания превосходящего потока. Ваша программа должна создавать изображения, подобные показанным на рис. 22.30 и на других рисунках этого раздела (см. упражнение 22.48). Протестируйте полученную реализацию на эвклидовых сетях из упражнений 22.7–22.12.

22.4. Сведение к максимальному потоку

В этом разделе мы рассмотрим некоторое число задач, решение которых сводится к решению задаче вычислении максимального потока с тем, чтобы показать, что алгоритмы, которые мы изучали в разделах 22.2 и 22.3, важны в более широком контексте. Мы можем снимать различные ограничения на задачи обработки сетей и графов, и, в конце концов, мы сможем решать задачи, которые выходят за рамки сетевых. В этом разделе будут рассмотрены примеры таких видов использования – определение максимального потока как общая модель решения задачи.

Мы также проведем исследование зависимости между задачей вычисления максимального потока и более сложными задачами с тем, чтобы установить контекст, позволяющий решать эти задачи в дальнейшем. В частности, отметим, что задача о максимальном потоке представляет собой специальный случай задачи отыскания потока минимальной стоимости, о которой пойдет речь в разделах 22.5 и 22.6, мы также покажем, как сформулировать задачи вычисления максимального потока как задачи линейного программирования, которые мы будем рассматривать в части 8. Задача отыскания потока минимальной стоимости и линейное программирование представляют собой более общие модели решения задач, нежели модели максимального потока. И хотя в общем случае мы можем решать задачи о максимальных потоках с применением специальных алгоритмов, описанных в разделах 22.2 и 22.3, с меньшими усилиями, чем с применением алгоритмов, предназначенных для решения более общих задач, очень важно иметь представление о существовании такого рода зависимости между моделями решения задач при переходе к более сложным моделям.

Мы будем употреблять термин *стандартная задача о максимальном потоке (standard maxflow problem)* в отношении версии задачи, которую мы изучали на протяжении последнего раздела (максимальный поток в *st*-сетях с ограниченной пропускной способностью ребер). Мы используем этот термин исключительно с целью облегчения ссылок в данном разделе. Итак, мы начнем с того, что покажем, что ограничения, обусловленные стандартной задачей о максимальном потоке, по существу не играют важной роли, поскольку несколько других задач о потоках сводятся стандартной задаче или эквивалентны ей. Мы можем рассматривать любую из эквивалентных задач как "стандартную" задачу. Простым примером таких задач, который уже упоминался как вытекающие из свойства 22.1, является задача отыскания в сетях циркуляции, позволяющая получить максимальный поток в заданном ребре. Далее, рассмотрим другие варианты постановки задачи, в каждом случае отмечая их связь со стандартной задачей.

Максимальный поток в сетях общего вида. Найти поток в сети, которая максимизирует суммарное истечение из ее источников (и, следовательно, приток в ее стоки). По соглашению, определяем поток как нулевой в сети, в которой нет истоков либо нет стоков.

Свойство 22.14. *Задача о максимальном потоке в сетях общего вида эквивалентна задаче вычисления максимального потока в st-сетях.*

Доказательство: Ясно, что алгоритм вычисления максимального потока в сетях общего вида будет работать на *st*-сетях, так что нам нужно установить, что задача общего вида сводится к задаче об *st*-сетях. Чтобы доказать это, найдем сначала истоки и стоки (с использованием, например, метода, который мы применяли для инициализации очереди в программе 19.8), и установим 0, если нет ни того, на другого. Далее, добавим фиктивную вершину-исток **s** и ребра, ведущие из **s** во все истоки сети (при этом устанавливаем пропускную способность такого ребра равной истечению вершины назначения этого ребра), а также фиктивное вершину-сток **t** и ребра, идущие из каждого стока сети в **t** (при этом устанавливаем пропускную способность такого ребра равной истечению начальной вершины назначения этого ребра). Рисунок 22.31 может послужить иллюстрацией такого сведения. Любой максимальный поток в *st*-сетях непосредственно соответствует максимальном потоку в исходной сети. ■

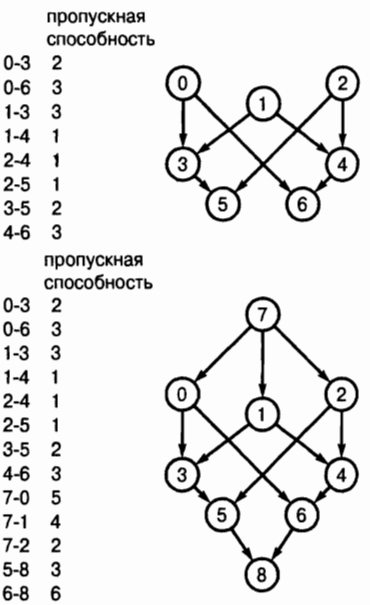


РИСУНОК 22.31. СВЕДЕНИЕ ЗАДАЧИ О МАКСИМАЛЬНОМ ПОТОКЕ В СЕТИ СО МНОГИМИ ИСТОКАМИ И СТОКАМИ К СТАНДАРТНАЯ ЗАДАЧЕ

*В сети, изображенной в верхней части рисунка, имеются три истока (вершины **1**, **2** и **3**) и два стока (вершины **5** и **6**). Чтобы найти поток, который обеспечивает максимальное истечение из истоков и максимальный приток в стоках, мы вычисляем максимальный поток в* st*-сети, представленной в нижней части рисунка. Эта сеть есть копия исходной сети, в которую добавлены исток **7** и сток **8**. В каждый исток исходной сети из вершины **7** проводим ребро, пропускная способность которого равна суммарной пропускной способности ребер, исходящих из соответствующего истока, и ребра, ведущие из каждого стока исходной сети в вершину **8**, пропускная способность которых равна суммарной пропускной способности ребер, входящих в стоки исходной сети.*

Ограничения, накладываемые на пропускную способность вершин. Пусть задана некоторая транспортная сеть, необходимо вычислить максимальный поток, удовлетворяющий ограничению, в соответствии с которым поток через каждую вершину не должен превышать некоторой фиксированной пропускные способности.

Свойства 22.15. *Задача вычисления максимального потока в транспортной сети с ограничениями на пропускную способность вершин эквивалентна стандартной задаче о максимальном потоке.*

Доказательство: И снова мы можем воспользоваться любым алгоритмом, который решает задачу об ограничениях на пропускную способность, для решения стандартной задачи (установив ограничения на пропускную способность в каждой вершине, согласно которому она больше, чем ее приток или ее истечение). В условиях заданного потока в сети с ограничениями по пропускной способности постройте стандартную транспортную сеть с двумя вершинами **u** и **u***, соответствующие каждой исходной вершине **u**, при этом все

ребра, входящие в исходную вершину, идут в **u**, а все исходящие ребра выходят из **u***, в то же время пропускная способность ребра **u-u*** равна пропускной способности исходной вершины. Эта конструкция показана на рис. 22.32. Потоки в ребрах типа **u*-v** при любом максимальном потоке в исходной сети должны удовлетворять ограничениям на пропускную способность вершин благодаря наличию ребер типа **u-u***. ■

Допуская наличие многих стоков и истоков или накладывая ограничения на пропускные способности, мы, на первый взгляд, обобщаем задачу о максимальных потоках; смысл свойств 22.14 и 22.15 сводится к тому, что решение этих задач становится нисколько не сложнее, чем решение стандартной задачи. Далее, мы рассмотрим одну версию этой задачи, которая поначалу кажется более легкой для решения, нежели стандартная задача.

Ациклические сети. Найти максимальный поток в ациклической сети. Осложняет ли наличие циклов в транспортной сети задачу вычисления максимального потока?

Мы уже сталкивались со многими примерами задач обработки орграфов, которые существенно усложняются при наличии в орграфах циклов. Возможно самыми известными из них следует считать задачи вычисления кратчайших путей во взвешенных орграфах, когда веса ребер могут принимать отрицательные значения (см. раздел 21.7), которые легко решаются за линейное время, когда циклы отсутствуют, и переходят в класс NP-трудных, если циклы допускаются. Как ни странно, задача о максимальном потоке в ациклических сетях ничуть не проще.

Свойство 22.16. *Задача вычисления максимального потока в ациклических сетях эквивалентна стандартной задаче о максимальном потоке.*

Доказательство: И снова нам достаточно показать, что стандартная задача сводится к ациклической задаче. Получив в свое распоряжение сеть с V вершинами и E ребрами, мы строим сеть с $2V + 2$ вершинами и $E + 3V$ ребрами, которая не только не принадлежит к категории ациклических, но и обладает простой структурой.

Пусть **u*** означает **u + V**, построим двудольный граф, в котором каждой вершине **u** исходной сети

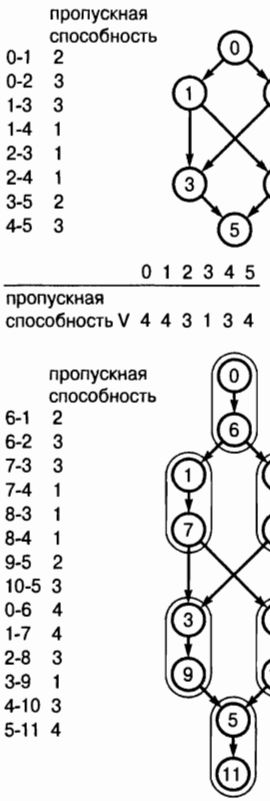


РИС. 22.32. УМЕНЬШЕНИЕ ПРОПУСКНЫХ СПОСОБНОСТЕЙ ВЕРШИН

*Чтобы решить задачу вычисления максимального потока в сети, показанной в верхней части рисунка, и чтобы этот поток не превосходил граничного значения пропускной способности, заданной массивом **capV**, индексированном именами вершин, построим стандартную сеть, которая приводится в нижней части рисунка. Для этого необходимо соединить новую вершину **u*** (где **u*** означает **v+V**) с каждой вершиной **u**, добавить ребро, пропускная способность которого равна пропускной способности вершины **u**, и включить ребро **u*-v** для каждого **u-v**. Каждая пара **u-u*** заключена в замкнутую линию. Любой поток в нижней сети напрямую соответствует потоку в верхней сети, которая удовлетворяет ограничениям, наложенным на пропускные способности вершин.*

соответствуют две вершины **u** и **u*** и каждому ребру **u-v** исходной сети соответствует ребро **u*-v** той же пропускной способности. Затем добавим в этот двудольный граф исток **s** и сток **t** и для каждой вершины **u** исходного графа, ребро **s-u** и ребро **u*-t**, пропускная способность каждого из них равна суммарной ребер, исходящих из вершины **u** в исходной сети. Наряду с этим, пусть X есть суммарная пропускная способность ребер исходной сети, добавим ребра из **u** в **u*** с пропускной способностью $X + 1$. Полученная конструкция представлена на рис. 22.33.

Чтобы показать, что любой максимальный поток в исходной сети соответствует максимальному потоку в преобразованной сети, рассмотрим вместо потоков сечения. Пусть задан любое *st*-сечение размера c в исходной сети, покажем, как построить *st*-сечение размера $c + X$ в преобразованной сети; кроме того, если задан любое минимальное сечение размера $c + X$ в преобразованной сети, мы покажем, как построить *st*-сечение размера c в исходной сети. Таким образом, если задано минимальное сечение в преобразованной сети, то соответствующее ему сечение в преобразованной сети также является минимальным. Более того, наше построение дает поток, величина которого равна пропускной способности минимального сечения, т.е. это максимальный поток.

В любом заданном сечении исходной сети, который отделяет исток от стока, пусть S представляет собой множество вершин истока, а T есть множество вершин стока. Постройте сечение преобразованной сети, помещая вершины из S в некоторое множество, содержащее вершину **s**, а вершины из T в некоторое множество, содержащее вершину **t**, и помещая вершины **u** и **u*** для всех **u** на одну и ту же сторону сечения, как показано на рис. 22.33. Для каждой вершины **u** во множестве связей сечения содержится либо **s-u**, либо **P|u*-t|**, а **u-v*** содержится во множестве связей сечения тогда и только тогда, когда ребро **u-v** содержится во множестве связей сечения исходной сети; следовательно, суммарная пропускная способность сечения равна пропускной способности исходной сети плюс X.

При любом заданном минимальном *st*-сечении преобразованной сети, пусть S^* есть множество вершины **s**, а T^* – множество вершины **t**. Наша цель заключается в том, чтобы построить сечение с той

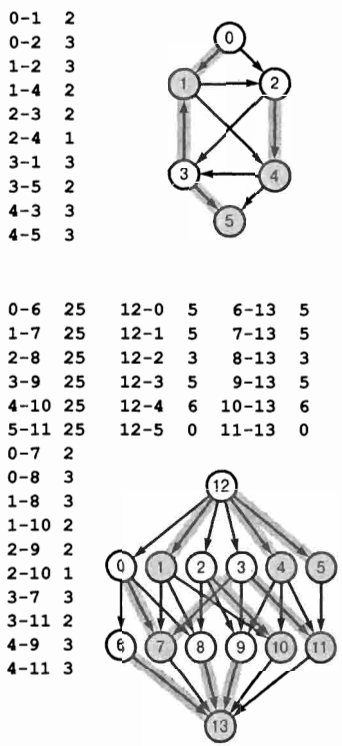


РИСУНОК 22.33. СВЕДЕНИЕ К АЦИКЛИЧЕСКОЙ СЕТИ

*Каждая вершина **u** в сети, изображенная в верхней части рисунка, соответствует двум вершинам **u** и **u*** (здесь **u*** означает **u** + V) в сети, показанной в нижней части рисунка, а каждое ребро **u-v** верхней сети соответствует ребру **u-v*** нижней сети. Кроме того, в нижней сети имеются ребра **u-u*** без пропускных способностей, источник **s** с ребрами, ведущими в каждую вершину, не отмеченную звездочкой, и сток **t**, в который ведет ребро из каждой вершины, помеченной звездочкой. Заштрихованные и незаштрихованные вершины (и ребра, соединяющие заштрихованные вершины с незаштрихованными) иллюстрируют прямую зависимость сечений в обеих сетях (см. рассуждения по тексту).*

же пропускной способностью и чтобы вершины **u** и **u*** входили в одно и то же множество для всех **u**, так что в соответствии с изложенным в предыдущем абзаце, мы получаем сечение в исходной сети, что и завершит наше доказательство. Во-первых, если **u** содержится в S^*, а **t** содержится в T^*, то **u-u*** должно быть пересекающим ребром, что приводит к противоречию: **u-u*** не может быть никаким минимальным сечением, поскольку сечение, состоящий из всех ребер, соответствующих ребрам исходного графа, имеет меньшую стоимость. Во-вторых, если **u** содержится в T^*, а **u*** содержится в S^*, то **s-u** должно находиться в этом сечении, поскольку оно является единственным ребром, соединяющим **s** и **u**. Но мы можем построить сечение той же стоимости путем подстановки всех ребер, направленных из **u**, на **s-u**, перемещая **u** в S^*.

Если в преобразованной сети задан какой-либо поток величины $c + X$, то мы просто мы назначаем каждому соответствующему ребру в исходной сети поток величины c. Преобразование сечения, описанное в предыдущем абзаце, не затрагивает это назначение, поскольку оно манипулирует ребрами, в которых поток равен 0. ■

В результате такого сведения мы не только получаем ациклическую сеть, мы получаем простую двудольную структуру. Это сведение говорит о том, что мы можем, если захотим, принять эти сети, обладающие более простой структурой, в качестве стандартных вместо сетей общего вида. Сначала кажется, что такая специальная структура позволяет создавать алгоритмы вычисления максимального потока, обладающие более высоким быстродействием. Однако описанное выше сведение показывает, что мы могли использовать любой алгоритм, разработанный для таких специальных видов сетей, какими являются ациклические сети, для решения задач о максимальном потоке в сетях общего вида ценой достаточно скромных дополнительных затрат. В самом деле, классические алгоритмы вычисления максимального потока используют гибкость модели сети общего вида: оба рассмотренных нами выше подхода, а именно, алгоритмы вычисления аугментальных путей и выталкивания избыточных потоков, используют принцип остаточной сети, предусматривающий введение циклов в сеть. Когда нам приходится решать задачу о максимальном потоке применительно к ациклической сети, для ее решения мы обычно используем стандартный алгоритм, ориентированный на сети общего вида.

Доказательство свойства 22.16 довольно сложное, оно показывает, что доказательства, использующие принцип сведения, требуют особого внимания, если не сказать – изобретательности. Такие доказательства важны уже потому, что не все версии задачи о максимальных потоках эквивалентны стандартной задаче, а мы желаем знать пределы, в каких мы можем применять наши алгоритмы. Математики продолжают исследование в этой области, поскольку сведение различных практических задач к другим задачам еще не установлено, как показывают следующий пример.

Максимальный поток в неориентированных сетях. Неориентированная транспортная сеть есть взвешенный граф с целочисленными весами ребер, которые мы интерпретируем как пропускные способности. Циркуляцией в таких сетях является назначение весов и направлений ребрам, удовлетворяющих условию, согласно которому поток в каждом ребре не может превышать его пропускной способности, а суммарный поток, поступающий в каждую вершину, равен суммарному потоку, покидающему эту вершину. Задача о неориентированном максимальном потоке состоит в том, чтобы найти циркуляцию, которая доводит поток в заданном направлении до максимально возможного значения в заданном ребре (т.е. из некоторой заданной вершины **s** в другую заданную вершину **t**). Эта

задача, по-видимому, более естественным образом, нежели стандартная задача, соответствует избранной нами модели трубопровода для транспортировки жидкостей: она позволяет случай, когда жидкость может протекать через трубы в обоих направлениях.

Свойство 22.17. *Задача о максимальном потоке для неориентированных st-сетей сводится к задаче о максимальном потоке для st-сетей.*

Доказательство: Пусть дана неориентированная сеть, построим ориентированную сеть с теми же вершинами, в которой каждому ребру исходной сети соответствуют два ребра, причем каждое из них обладает пропускной способностью, равной пропускной способности неориентированного ребра. Любой поток в исходной сети непосредственно соответствует потоку такой же величины, что и преобразованной сети. Обратное также верно: если в неориентированной сети через ребро **u-v** протекает поток f, через ребро **v-u** протекает поток g, то мы можем поместить поток $f - g$ в ребро **u-v** ориентированной сети, если $f \geq g$; в противном случае это будет поток $g - f$ в ребре **v-u**. Следовательно, любой максимальный поток в ориентированной сети есть максимальный поток в неориентированной сети: это построение дает поток, и любой поток в ориентированной сети большей величины будет соответствовать некоторому потоку большей величины в неориентированной сети; однако такого потока не существует. ■

Это доказательство не утверждает, что задача о максимальном потоке в неориентированной сети эквивалентна стандартной задаче. Иначе говоря, вполне может случиться, что вычисление максимальных потоков в неориентированных сетях легче, чем вычисление максимальных потоков в стандартных сетях (см. упражнение 22.81).

В целом мы можем применять к сетям со многими стоками и истоками, неориентированным сетям, к сетям с ограничениями на пропускные способности вершин и многим другим типам сетей (см., например, упражнение 22.79) алгоритмы вычисления максимального потока для *st*-сетей, рассмотренные в двух предыдущих разделах. Фактически свойство 22.16 утверждает, что мы можем решить все эти задачи даже при помощи алгоритма, который работает только на ациклических сетях.

Далее, мы рассматриваем задачу, которая не является чисто задачей о максимальном потоке, но которую мы можем свести к задаче о максимальном потоке и решить ее с помощью алгоритмов вычисления максимального потока. Это один из способов формализовать базовую версию задачи распределения товаров, описание которой дано в начале данной главы.

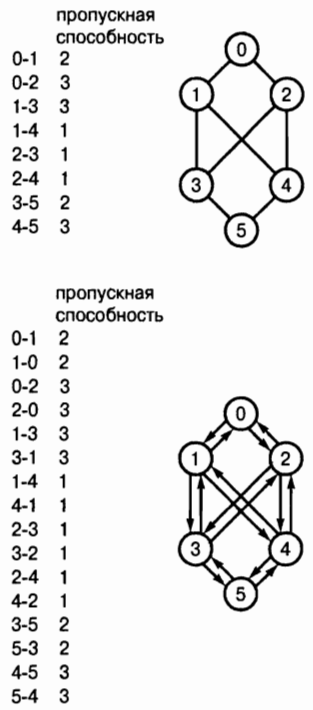


РИСУНОК 22.34. СВЕДЕНИЕ ЗАДАЧИ О ПОТОКАХ В НЕОРИЕНТИРОВАННЫХ СЕТЯХ К СТАНДАРТНОЙ ЗАДАЧЕ

Чтобы решить задачу о максимальном потоке в неориентированных сетях, мы можем рассматривать ее как ориентированную сеть с ребрами, направленными в обоих направлениях. Обратите внимание на тот факт, что каждому ребру неориентированной сети соответствует четыре ребра в соответствующей ей остаточной сети.

Допустимый поток. Предположим, что веса присвоены каждой вершине транспортной сети, и что вес можно рассматривать как запас (если он принимает положительное значение) или как спрос (если он принимает отрицательное значение), при этом сумма весов вершин сети равна нулю. Будем считать поток *допустимым*, если разность между притоком и истечением каждой вершины равна весу этой вершины (запас, если разность положительна, и спрос, если разность отрицательна). Пусть дана такая сеть, определить, существуют ли допустимые потоки. Рисунок 22.35 служит иллюстрацией задачи о допустимом потоке.

Вершины с запасом соответствуют складам в задаче о распределении запасов; вершины со спросом соответствуют розничным торговым точкам; ребра соответствуют дорогам на маршрутах грузовых машин. Задача о допустимых потоках отвечает на следующий основной вопрос: можно ли отыскать такой способ доставки товаров, при котором повсеместно обеспечивается соответствие запасов и спроса.

Свойство 22.18. *Задача о допустимом потоке сводится к задаче о максимальном потоке.*

Доказательство: Пусть поставлена задача о допустимом потоке, построить сеть с теми же вершинами и ребрами, но вершины при этом не имеют весов. Вместо этого добавьте вершину истока **s**, из которой исходят ребра в каждую вершину, имеющую запас с весом, равным запасу соответствующей вершины, и вершину стока **t**, в которую ведет ребро из каждой вершины со спросом (так что вес такого ребра положителен). Решим задачу о максимальном потоке на этой сети. В исходной сети содержится допустимое ребро тогда и только тогда, когда все ребра, исходящие из истока, и все ребра, ведущие в сток, заполнены при таком потоке до пропускной способности. Рисунок 22.36 показывает пример такого сведения. ■

Разработка классов, реализующих сведения рассмотренного типа, которые анализировались выше, может оказаться сложной задачей для программистов, главным образом в силу того обстоятельства, что объекты, которыми мы манипулируем, представлены сложными структурами данных. Должны ли мы строить новую сеть, чтобы свести еще одну задачу к стандартной задаче о максимальном потоке? Некоторые из задач требуют для своего решения дополнительных данных, таких как пропускная способность вершин,

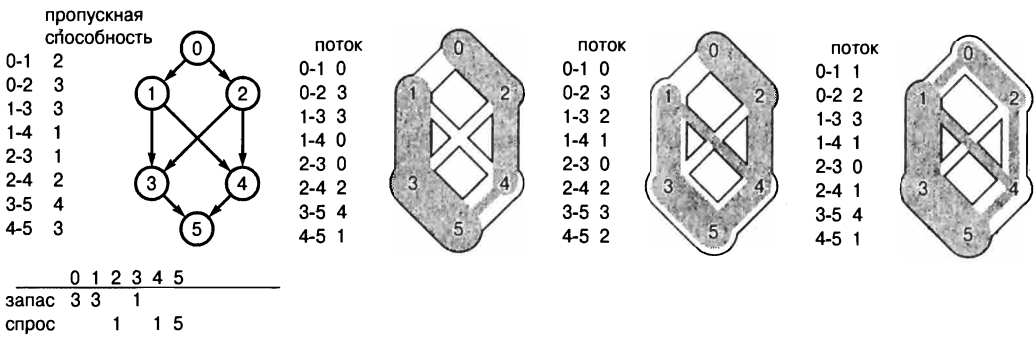


РИСУНОК 22.35. ДОПУСТИМЫЙ ПОТОК

В рамках задачи о допустимом потоке мы задаем величины запаса и спроса в дополнение к значениям пропускной способности ребер. Мы ищем такой поток, при котором истечение равно запасам плюс притоку в вершинах с запасом, а приток равен истечению плюс спрос в вершинах со спросом. Три решения задачи о допустимых потоках, представленных на диаграмме слева, показаны на диаграммах справа.

запас или спрос в каждой вершине, так что построение стандартной сети без этих данных может оказаться оправданным. Использование указателей на ребра играет здесь важную роль: если скопировать ребра сети, а затем вычислить максимальный поток, то что мы должны делать с результатом? Перенос вычисленного потока (вес каждого ребра) из одной сети в другую, когда обе сети представлены в виде списков смежных вершин, – это отнюдь не тривиальное вычисление. Когда используются указатели на ребра, новая сеть содержит копии указателей, а не ребер, так что мы можем передавать установки потоков непосредственно в сеть клиента. Программа 22.6 представляет собой реализацию, которая может служить иллюстрацией некоторых из этих проблем, которые приходится преодолевать в классе при решении задач поиска подходящих потоков с использованием свойства 22.16.

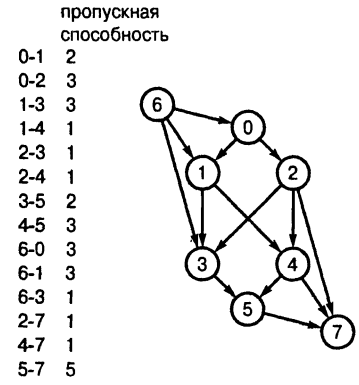


РИСУНОК 22.36. СВЕДЕНИЕ ЗАДАЧИ О ДОПУСТИМОМ ПОТОКЕ ДО СТАНДАРТНОЙ ЗАДАЧИ

Эта сеть есть стандартная сеть, построенная для задачи о допустимом потоке за счет добавления ребер, ведущих из новой вершины истока в вершины с запасами (пропускные способности каждого такого ребра равны величине запаса), и ребер, ведущих в новую вершину стока из вершин, обладающих спросом (пропускные способности каждого такого ребра равны величине спроса). В сети, показанной на рис. 22.35, допустимый поток имеется тогда и только тогда, когда в этой сети имеется поток (максимальный поток), который заполняет все ребра, исходящие из стока, и все ребра, входящие в исток.

Программа 22.6. Решение задачи о допустимых потоках путем сведения ее к задаче о максимальном потоке

Рассматриваемый класс решает задачу о допустимом потоке за счет сведения ее к задаче о максимальном потоке, используя построение, представленное на рис. 22.36. Конструктор принимает в качестве аргументов сеть и вектор **sd**, индексированный именами вершин, такой, что значение **sd[i]**, будучи положительным, представляет запас в вершине **i** и, будучи отрицательным, – спрос в вершине **i**.

Как было показано на рис. 22.36, конструктор строит новый граф с теми же ребрами, но с двумя дополнительными вершинами **s** и **t**, при этом ребра из **s** ведут во все вершины с запасами, а из всех вершин со спросом ребра ведут в вершину **t**. Затем она находит максимальный поток и проверяет, заполнены ли все дополнительные ребра до их пропускных способностей.

```
#include  "MAXFLOW.cc"
template <class Graph, class Edge> class FEASIBLE
{ const Graph &G;
   void freeedges(const Graph &F, int v)
    { typename Graph::adjIterator A(F, v);
      for (EDGE* e = A.beg(); !A.end(); e = A.nxt())
       delete e;
    }
public:
   FEASIBLE(const Graph &G, vector<int> sd) : G(G)
   {
    Graph F(G.V()+2);
```

```
    for (int v = 0; v < G.V(); v++)
    {
       typename Graph::adjIterator A(G, v);
       for (EDGE* e = A.beg(); !A.end(); e = A.nxt())
          F.insert(e);
    }
    int s = G.V(), t = G.V()+1;
    for (int i = 0; i < G.V(); i++)
     if (sd[i] >= 0)
        F.insert(new EDGE(s, i, sd[i]));
     else
        F.insert(new EDGE(i, t, -sd[i]));
    MAXFLOW<Graph, Edge>(F, s, t);
    freeedges(F, s); freeedges(F, t);
  }
};
```

Каноническим примером задачи о потоках, которую мы не можем решить в рамках модели максимального потока, и которая представляет собой темой исследований, проводимых в разделах 22.5 и 22.6, является расширение задачи о допустимых потоках. Мы вводим в употребление второе множество весов ребер и ставим перед собой цель отыскать допустимый поток с минимальной стоимостью. Эта модель формализует общую задачу распределения товаров. Мы хотим не только знать, можно ли транспортировать товары, но и какова минимальная стоимость такого транспортирования.

Все задачи, которые мы до сих пор рассматривали в данном разделе, имеют одну и ту же цель (вычисление потоков в транспортной сети), так что не следует удивляться тому, что мы можем решать их в рамках модели, характерной для решения задач транспортной сети. Как мы могли убедиться при доказательстве теоремы о максимальных потоках и минимальных сечениях, мы можем пользоваться алгоритмами вычисления максимального потока для решения задач обработки графов, которые, на первый взгляд, не имеют мало общего с потоками. Теперь мы обратимся к примерам такого рода.

Двудольное сочетание с максимальным кардинальным числом. Пусть задан двудольный граф, найти множество ребер с максимальным числом элементов, таких, что каждая вершина соединена максимум с одной другой вершиной.

Для краткости изложения будем далее называть эту задачу просто задачей *двудольного сочетания (bipartite matching)* за исключением случаев, в которых нужно отличить ее от других подобных задач. Она формализует задачу трудоустройства, которая кратко рассматривалась в начале данной главы. Вершины соответствуют претендентам и работодателям, а ребра – отношению "взаимной заинтересованности в трудоустройстве". Решение задачи двудольного сочетания приводит к максимально возможному трудоустройству. На рис. 22.37 показан двудольный граф, моделирующий демонстрационную задачу, представленную на рис. 22.3.

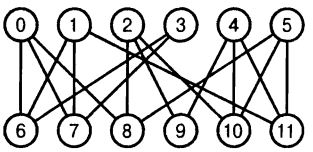


РИСУНОК 22.37. ДВУДОЛЬНОЕ СОЧЕТАНИЕ

Этот пример задачи двудольного сочетания служит формальным представлением задачи трудоустройства, отображенной на рис. 22.3. Отыскание наилучшего способа, позволяющего студентам получить рабочие места, на которые они претендуют, эквивалентно выявлению в этом двудольном графе максимального числа ребер с непересекающимися вершинами.

Это весьма поучительный пример, заставляющий задуматься над тем, как найти прямое решение задачи о двудольном сочетании без применения с этой целью модели, использующей графы. Например, эта задача сводится к следующей комбинаторной головоломке: "найти максимальное подмножество пар целых чисел (взятых из непересекающихся множеств), обладающих тем свойством, что ни одна из пар не содержит одинаковых целых чисел". Пример, приведенный на рис. 22.27, соответствует решению этой головоломки на парах **0-6**, **0-7**, **0-8**, **1-6** и т.д. Эта задача, на первый взгляд, кажется простой, однако, как показал пример задачи поиска гамильтонова пути, которую мы исследовали в разделе 17.7 (и многих других задач), наивный подход выбора пар каким-либо систематическим способом до тех пор, пока не возникнет противоречие, требует для своего выполнения экспоненциальных затрат времени. То есть, существует слишком большое подмножеств пар, чтобы можно было проверить все возможности; решение этой задачи должно быть достаточно разумным, чтобы мы использовали лишь немногими из них. Решение специальных головоломок сочетания, подобных описанной выше, или разработка алгоритмов, которые способны эффективно решать любую такую головоломку, суть нетривиальные случаи, позволяющие продемонстрировать большие возможности и применимость модели потоков в сети, которая также позволяет найти разумный способ построения двудольного сочетания.

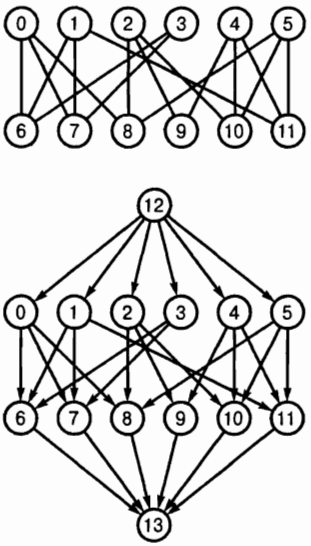


РИСУНОК 22.38. СВЕДЕНИЕ ЗАДАЧИ О ДВУДОЛЬНОМ СОЧЕТАНИИ ЗАДАЧЕ О МАКСИМАЛЬНОМ ПОТОКЕ

Чтобы найти максимальное сочетание в двудольном графе (сверху) мы строим st*-сеть (внизу), направляя все ребра из верхнего ряда в нижний ряд, добавляя новый исток, из которого ребра ведут в каждую вершину верхнего ряда, добавляя новый сток, в который ведут ребра из каждой вершины нижнего ряда, и назначая всем ребрам пропускную способность, равную 1. При любом потоке может быть заполнено, самое большее, одно ребро из всех ребер, исходящих из каждой вершины верхнего ряда, а также может быть заполнено, самое большее, одно входящее в ребро из всех ребер, входящих в каждую вершину нижнего ряда, так что решение задачи о максимальном потоке для этой сети дает максимальное сочетание для двудольных графов.*

Свойство 22.19. *Задача о двудольном сочетании сводится к задаче о максимальном потоке.*

Доказательство: Для заданной задачи двудольного сочетания построим пример задачи о максимальном потоке за счет проведения всех ребер из одного множества в другое, с добавлением истока, из которого направлены ребра во все элементы одного множества, и стока, в который направлены ребра из элементов другого множества. Чтобы преобразовать полученный орграф в сеть, назначим каждому ребру пропускную способность, равную 1. Означенное построение показано на рис. 22.38.

Теперь любое решение задачи о максимальном потоке применительно к этой сети дает решение соответствующей задачи о двудольном сочетании. Сочетание в точности соответствует тем ребрам, соединяющим вершины обоих множеств, которые заполнены до пропускной способности алгоритмом вычисления максимального потока. Во-пер-

вых, потоки в сети всегда дают допустимое сочетание: поскольку в каждую вершину входит (из истока) ребро с пропускной способностью, равной единице, либо выходит из нее (в сток), то через каждую вершину может пройти только единица потока, откуда, в свою очередь, следует, что каждая вершина может быть использована для сочетания только один раз. Во-вторых, ни одно сочетание не может использовать большее число ребер, поскольку такое сочетание может непосредственно привести к большей величине потока, чем та, которая получена по алгоритму вычисления максимального потока. ■

Например, на графе из рис. 22.38 алгоритм вычисления максимального потока методом аугментальных путей может использовать пути **s-0-6-t**, **s-1-7-t**, **s-2-7-t**, **s-4-9-t**, **s-5-10-t** и **s-3-6-0-7-1-11-t** для вычисления сочетания **0-7**, **1-11**, **2-8**, **3-6**, **4-9** и **5-10**. Следовательно, существует способ трудоустройства всех студентов в задаче, показанной на рис. 22.3.

Программа 22.7 является клиентской программой, которая считывает задачу о двудольном сочетании со стандартного ввода и использует сведение, задействованное при доказательстве существования ее решения. Каким будет время выполнения этой программы на крупных сетях? Разумеется, время выполнения зависит от выбора алгоритма вычисления максимального потока и используемой его реализации. Кроме того, мы должны принять во внимания тот факт, что для сетей, которые мы строим, характерна специальная структура (двудольные транспортные сети с единичной пропускной способностью ребер), благодаря чему время выполнения различных алгоритмов вычисления максимальных потоков не только не приближается к границам, определенным для худшего случая, но и существенно снижают эти границы. Например, первая граница, которую мы подвергли анализу, определяет быстрый отклик для случая выполнения обобщенного алгоритма вычисления аугментальных путей.

Программа 22.7. Двудольное сочетание через приведение к алгоритму вычисления максимального потока

Этот клиент считывает задачу о двудольном сочетании со стандартного ввода, затем строит транспортную сеть, соответствующую этой задаче двудольного сочетания, находит максимальный поток и использует полученное решение для распечатки максимального двудольного сочетания.

```
#include  "GRAPHbasic.cc"
#include  "MAXFLOW.cc"
int main(int argc, char *argv[])
{ int s, t, N = atoi(argv[1]);
   GRAPH<EDGE>  G(2*N+2);
   for (int i = 0; i < N; i++)
      G.insert(new EDGE(2*N, i, 1));
   while (cin >> s >> t)
      G.insert(new EDGE(s, t, 1));
   for (int i = N; i < 2*N; i++)
      G.insert(new EDGE(i, 2*N+1, 1));
   MAXFLOW<GRAPH<EDGE>, EDGE>(G, 2*N, 2*N+1);
   for (int i = 0; i < N; i++)
     {
        GRAPH<EDGE>::adjIterator A(G, i);
        for (EDGE* e = A.beg(); !A.end(); e = A.nxt())
         if (e->flow() == 1 && e->from(i))
         cout << e->v() << "-" << e->w() << endl;
     }
}
```

Следствие. *Время, которое требуется для вычисления сочетания на максимальном числе элементов некоторого двудольного графа, есть* $O(VE)$.

Доказательство: Непосредственно следует из свойства 22.6. ■

Работу алгоритмов аугментальных путей на двудольной сети с единичной пропускной способностью ребер описать нетрудно. Каждый аугментальный путь наполняет одно ребро, ведущее из истока, и одно ребро, ведущее в сток. Эта ребра никогда не используются как обратные, следовательно, существуют не более V аугментальных путей. Для каждого алгоритма, который находит аугментальные пути за время, пропорциональное E, действительна верхняя граница, пропорциональная VE.

В таблице 22.3 показаны значения производительности методов решения задач двудольного сочетания с использованием различных алгоритмов вычисления аугментальных путей. Из этой таблицы следует, что фактические значения времени решения этой задачи ближе к границе VE для худшего случая, чем к оптимальному (линейному) времени. Благодаря разумному выбору и соответствующей настройке реализации, вычисляющей максимальный поток, можно увеличить быстродействие этого метода в $\sqrt{V}$ раз (см. упражнения 22.91 и 22.92).

Таблица 22.3. Эмпирические исследования двудольного сочетания

В эту таблицу сведены показатели производительности (число добавленных вершин и число затронутых узлов списков смежных вершин) для случаев использования различных алгоритмов вычисления максимального потока методом аугментальных путей при вычислении двудольного сочетания для графов с 2000 парами вершин и 500 ребрами (вверху), а также 4000 ребер (внизу). Для решения этой задачей наиболее эффективным показал себя поиск в глубину.

	Вершины	Ребра
500 ребер, число сочетаний 347		
Кратчайший путь	1071	945599
Максимальная пропускная способность	1067	868407
Поиск в глубину	1073	477601
Случайный поиск	1073	644070
4000 ребер, число сочетаний 971		
Кратчайший путь	3483	8280585
Максимальная пропускная способность	6857	6573560
Поиск в глубину	34109	1266146
Случайный поиск	3569	4310656

Эта задача характерна для ситуаций, с которыми мы чаще сталкиваемся, когда изучаем новые задачи и более общие модели решения задач, демонстрирующие эффективность применения на практического сведения как инструментального средства решения задач. Если мы сможем найти сведение к более известной общей задаче, к такой как, например, задача о максимальном потоке, мы обычно рассматриваем это как существенное продвижение на пути к практическому решению, ибо оно, по меньшей мере, не только указывает на то, что задача решаемая, но и на то, что существуют многочисленные эффективные алгоритмы решения рассматриваемой задачи. Во многих ситуациях целесо-

образно использовать существующий класс максимального потока для решения такой задачи и переходить к решению следующей задачи. Если производительность остается критической проблемой, мы можем провести исследование различных алгоритмов вычисления максимальных потоков и их реализаций или использовать их поведение как отправную точку для разработки более совершенного алгоритма специального назначения. Общая модель решения задач устанавливает верхнюю границу, которой мы можем либо удовлетвориться, либо попытаться ее улучшить, и обеспечивает множество реализаций, которые продемонстрировали свою эффективность при решении других многочисленных задач.

Далее мы обсудим задачи, имеющие отношение к связности графов. Прежде чем принимать решение использовать алгоритмы вычисления максимального потока для решения задачи о связности графа, мы исследуем возможность применения теоремы о максимальных потоках и минимальных сечениях, чтобы завершить работу, начатую в главе 18, т.е. получить доказательства базовых теорем, имеющих отношение к путям и сечениям в неориентированных графах. Эти доказательства являются дальнейшим свидетельствованием, подчеркивающим фундаментальное значение теоремы о максимальных потоках и минимальных сечениях.

Свойство 22.20. (Теорема Менгера). *Минимальное число ребер, удаление которых разъединяет две вершины орграфа, равно максимальному числу не пересекающихся по ребрам (edge-disjoint) путей между этими двумя вершинами.*

Доказательство: Пусть задан граф, определить транспортную сеть с теми же самыми вершинами и ребрами, при этом всем ребрам присваиваются значения пропускных способностей, равные 1. По свойству 22.2 любую *st*-сеть мы можем представить как множество не пересекающихся по ребрам путей из s в t, при этом число таких путей равно величине потока. Пропускная способность любого *st*-сечения равна кардинальному числу сечения. С учетом всех этих фактов, теорема о максимальных потоках и минимальных сечениях дает результат, представленный в формулировке свойства. ■

Соответствующий результат для неориентированных графов, для связности вершин в орграфах и неориентированных графах, предусматривает применение концепции сведения, подобное рассмотренному в тексте и приложениях (упражнения 22.94–22.96).

Обратимся теперь к алгоритмическим проблемам, порождаемым зависимостью между потоками и связностью, которая устанавливается теоремой о максимальных потоках и минимальных сечениях. Свойство 22.5, по-видимому, представляет собой один из важнейших результатов (задача о минимальном сечении сводится к задаче о максимальном потоке), однако обратное утверждение не доказано (см. упражнение 22.47). Здравый смысл подсказывает нам, что располагая информацией о минимальном сечении, легче решать задачу определения максимального потока, однако пока никому еще не удалось показать, почему это так. Этот базовый пример подчеркивает необходимость действовать осторожно при сведении одних задач к другим.

В то же время, мы все еще можем использовать алгоритмы вычисления максимального потока для решения различных задач о связности. Например, они полезны при решении первых нетривиальных задач, с которыми мы сталкивались в главе 18.

Реберная связность. Каким должно быть минимальное число ребер, которые необходимо удалить с тем, чтобы разделить заданный граф на две части? Найти множество ребер с минимальным числом элементов, обеспечивающее такое разделение.

Вершинная связность. Каким должно быть минимальное число вершин, которые необходимо удалить с тем, чтобы разделить заданный граф на две части? Найти множество вершин с минимальным числом элементов, обеспечивающее такое разделение.

Эти задачи существуют также и в отношении орграфов, таким образом, в общем случае мы должны рассмотреть четыре задачи. Как и в случае теоремы Менгера, мы исследуем одну из них во всех подробностях (реберная связность в неориентированных графах), а остальные оставим читателям на самостоятельную проработку.

Свойство 22.21. *Время, необходимое для определения реберной связности в неориентированных графах, есть* $O(E^2)$.

Доказательство: Мы можем вычислить минимальный размер любого сечения, которое разделяет две заданных вершины, за счет вычисления максимального потока в *st*-сети, построенной на базе графа с назначением единичной пропускной способности каждому ребру. Реберная связность равна минимальному из этих значений на всех парах вершин.

Нет необходимости, однако, выполнять вычисления для всех пар вершин. Пусть на графе **s*** есть вершина с минимальной степенью. Обратите внимание на то, что степень вершины **s*** не может быть больше $2E/V$. Рассмотрим минимальное сечение этого графа. По определению, число ребер в сечении равно реберной связности графа. Вершина **s*** появляется в одном из множеств вершин сечения, а в другое множество должна входить некоторая вершина **t**, так что размер любого минимального сечения, разделяющего вершины **s*** и **t**, должен быть равен реберной связности графа. Следовательно, если мы решим $V-1$ задач о минимальном потоке (используя **s*** как исток и любую другую вершину как сток), полученная величина минимального потока будет являться связностью рассматриваемой сети.

Теперь любой алгоритм вычисления максимального потока с использованием аугментальных путей с вершиной **s*** в качестве истока требует вычисления максимум $2E/V$ путей; таким образом, если мы используем метод, требующий самое большее E шагов для определения аугментального пути, получаем самое большее $(V-1)(2E/V)E$ для определения реберной связности, откуда и следует искомый результат. ■

Этот метод, в отличие от всех других примеров из этого раздела, не есть сведение одной задачи к другой, но он дает практический алгоритм для вычисления реберной связности. И опять-таки, аккуратная настройка реализации вычислений максимального потока на эту конкретную задачу позволяет повысить производительность – мы можем решить эту задачу за время, пропорциональное VE (*см. раздел ссылок*). Доказательство свойства 22.21 служит примером более общего понятия эффективного (выполняемого за полиномиальное время) сведения, с которым мы впервые столкнулись в разделе 21.7 и которое играет существенную роль в теории алгоритмов, исследуемых в части 8. Такое сведение доказывает не только то, что задача принадлежит к числу решаемых, но и предлагает алгоритм для ее решения – т.е. важный первый шаг при решении новой комбинаторной задачи.

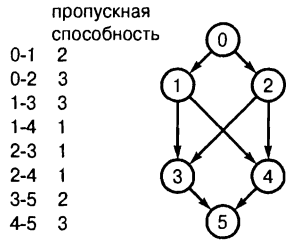
Мы завершим этот раздел анализом строгой математической формулировки задачи о максимальном потоке, используя средства линейного программирования (см. раздел 21.6). Это упражнение полезно тем, что оно помогает проследить связи с другими задачами, которые также могут быть четко сформулированы.

Формулировка этой задачи достаточно проста: мы рассматриваем систему неравенств, в которой каждому ребру соответствует одна переменная величина, каждому ребру соответствуют два неравенства и каждой вершине соответствует одно уравнение. Значение переменной есть величина потока в ребре, неравенства указывают на то, что величина потока в ребре должна принимать значение в пределах от 0 до пропускной способности этого ребра, а уравнение указывает на то, что суммарный поток в ребрах, которые ведут в некоторую конкретную вершину, должен быть равен суммарному потоку в ребрах, которые исходят из этой вершины.

Рисунок 22.39 служит примером такого построения. Любая задача о максимальном потоке таким способом может быть преобразована в задачу линейного программирования (в LP-задачу). Линейное программирование представляет собой гибкий подход к решению комбинаторных задач, и многочисленные задачи, которые мы изучаем, могут быть сформулированы как линейные программы. Тот факт, что задачи о максимальном потоке легче поддаются решению, чем LP-задача, можно объяснить тем фактом, что в формулировке задач о максимальном потоке как LP-задач употребляются ограничения, имеющие специфическую структуру, которая характерна не для всех LP-задач.

Даже если LP-задача в общем виде намного сложнее, чем задачи специального вида, такие как задача о максимальном потоке, существуют мощные алгоритмы, которые могут эффективно решать LP-задачи. Время выполнения этих алгоритмов в худшем случае почти наверняка превосходит время выполнения в худшем случае специальных алгоритмов, которые мы рассматривали выше, однако громадный опыт, накопленный за последние несколько десятилетий, показал их эффективность при решении задач этого типа, которые часто возникают на практике.

Построение, представленное на рис. 22.39, служит доказательством того, что задача о максимальном потоке сводится к соответствующей LP-задаче, если мы не настаиваем, что величины потоков должны быть целыми числами. Мы будем подробно рассматривать LP-задачи в части 8, где и опишем способ преодолеть затруднения, связанные с тем, что формулировка задачи о



Вычислить наибольшее значение x_{50} в условиях действия следующих ограничений

$$x_{01} \le 2$$
$$x_{02} \le 3$$
$$x_{13} \le 3$$
$$x_{14} \le 1$$
$$x_{23} \le 1$$
$$x_{24} \le 1$$
$$x_{35} \le 2$$
$$x_{45} \le 3$$
$$x_{50} = x_{01} + x_{02}$$
$$x_{01} = x_{13} + x_{14}$$
$$x_{02} = x_{23} + x_{24}$$
$$x_{13} + x_{23} = x_{35}$$
$$x_{14} + x_{24} = x_{45}$$
$$x_{35} + x_{45} = x_{50}$$

РИСУНОК 22.39. ФОРМУЛИРОВКА ЗАДАЧИ О МАКСИМАЛЬНОМ ПОТОКЕ В ТЕРМИНАХ ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ

Эта линейная программа эквивалентна задаче о максимальном потоке применительно к демонстрационному примеру, показанному на рис. 22.5. В рассматриваемом случае на каждое ребро приходится одно неравенство (которое показывает, что поток в некотором ребре не может превосходить его пропускной способности) и одно равенство на каждую вершину (которое показывает, что приток должен быть равен истечению из этой вершины). В этой сети мы используем фиктивное ребро, ведущее из стока в исток, в соответствии с изложенным при анализе свойства 22.2.

максимальном потоке в терминах линейного программирования не содержит условия, требующего, чтобы результаты ее решения были выражены в целых числах.

Этот контекст дает нам возможность воспользоваться жесткой математической основой, в рамках которой мы можем искать решения задач более общего вида и создавать более совершенные алгоритмы решения этих задач. Задача о максимальном потоке легко поддается решению, она обладает своего рода гибкостью, о чем говорят примеры, приведенные в данном разделе. Далее мы будем изучать более сложные задачи (но, тем не менее, они уступают по сложности LP-задачам), которые охватывают определенный класс практических задач. Мы обсудим разновидности решений в рамках этих моделей с возрастающим уровнем абстракции в конце этой главы, тем самым заложив основу для их полного анализа в части 8.

Упражнения

▷ **22.70**. Определите класс, выполняющий поиск циркуляции при максимальном потоке в заданном ребре. Разработайте реализацию, использующую функцию **MAXFLOW**.

22.71. Определите класс, вычисляющий максимальный поток без ограничений на число истоков и стоков. Разработайте реализацию, использующую функцию **MAXFLOW**.

22.72. Определите класс, вычисляющий максимальный поток в неориентированной сети. Разработайте реализацию, использующую функцию **MAXFLOW**.

22.73. Определите класс, вычисляющий максимальный поток в сети с ограничениями на пропускную способность вершин. Разработайте реализацию, использующую функцию **MAXFLOW**.

▷ **22.74**.Разработайте класс для решения задач о допустимых потоках, который включает функции-элементы, позволяющие клиентам устанавливать значения запаса и спроса с целью проверки правильности выбора значений потоков для каждой вершины.

22.75. Выполните упражнение 22.18 для случая, когда каждая точка распределения обладает ограниченной пропускной способностью (т.е. существует предел количества товаров, которые могут храниться в этой точке в любой заданный момент времени).

▷ **22.76**. Показать, что задача о максимальном потоке сводится к задаче о допустимом потоке (благодаря чему обе эти задачи можно рассматривать как эквивалентные).

22.77. Найти допустимый поток для транспортной сети, представленной на рис. 22.10 в условиях, когда даны дополнительные ограничения, по условиям которых вершины **0**, **2** и **3** суть вершины с запасами с весом 4, а вершины **1**, **4** и **5** суть вершины с запасами и весами, соответственно 1, 3 и 5.

○ **22.78**. Напишите программу, которая принимает в качестве ввода расписание соревнований некоторой спортивной лиги и текущее положение команд и определяет, выбыла ли заданная команда из турнира. Предполагается, что связи отсутствуют. *Указание*: Приведите данную задачу к задаче о допустимых потоках с одним истоком, который обладает величиной запаса, равной суммарному числу игр, которые осталось сыграть в текущем сезоне, с узлами стока, которые соответствуют каждой паре команд, имеющих величину спроса, равной числу оставшихся игр между этой парой, и с узлами распределения, соответствующими каждой команде. Ребра должны соединять узлы с запасами с узлами распределения каждой команды (пропускная способность которых равна числу игр, которые команда должна выиграть, чтобы выиграть у X при условии,

что X выиграет все свои оставшиеся игры), и должно существовать ребро (без пропускной способности), соединяющее узел распределения каждой команды с каждым узлом спроса, имеющим отношение к этой команде.

22.79. Покажите, что задача о максимальном потоке для сетей с относительно низкими границами на пропускные способности ребер сводится к стандартной задаче о максимальном потоке.

▷ **22.80**. Покажите, что для сетей с относительно низкими границами на пропускные способности ребер задача определения *минимального* потока (которая выдерживает границы) сводится к задаче о максимальном потоке (см. упражнение 22.79).

●●● **22.81**. Докажите, что задача о максимальном потоке в *st*-сетях сводится к задаче о максимальном потоке в неориентированных сетях, либо найдите алгоритм вычисления максимального потока в неориентированных сетях, время выполнения которого в худшем случае значительно лучше, чем аналогичный параметр алгоритмов, исследованных в разделах 22.2 и 22.3.

▷ **22.82**. Найдите все сочетания с пятью ребрами для двудольного графа, представленного на рис. 22.37.

22.83. Расширьте программу 22.7 таким образом, чтобы можно было пользоваться символическими именами вершин вместо цифровых обозначений (см. программу 17.10).

○ **22.84**. Докажите, что задача двудольного сочетания эквивалентна задаче нахождения максимального потока в сети, в которой все ребра наделены единичной пропускной способностью.

22.85. Мы можем интерпретировать пример на рис. 22.3 как предпочтение студента на выбор работы и предпочтение работодателя на выбор студентов, причем оба они могут и не быть взаимными. Применимо ли сведение, описанное в тексте, к *ориентированной* задаче двудольного сочетания, которая вытекает из этой интерпретации, где ребра двудольного графа направлены (в обоих направлениях) из одного множества в другое? Докажите, что это так, либо приведите встречный пример.

○ **22.86**. Постройте семейство задач двудольного сочетания, где средняя длина аугментальных путей, используемая любым алгоритмом построения аугментальных путей для решения соответствующей задачи о максимальном пути, оказывается пропорциональной E.

22.87. Покажите в стиле рис. 22.28, работу алгоритма выталкивания превосходящего сетевого потока, использующего очередь FIFO, на сети двудольного сочетания, показанной на рис. 22.38.

○ **22.88**. Расширьте таблицу 22.3 таким образом, чтобы она включала различные алгоритмы вытеснения превосходящего потока.

● **22.89**. Предположим, что в задаче двудольного сочетания имеются два множества размерами S и T, при этом $S << T$. Дайте как можно более четкую границу времени решения этой задачи в худшем случае для сведения, определяемого свойством 22.19, и реализации алгоритма Форда-Фалкерсона, использующей построение максимальных аугментальных путей (см. свойство 22.8).

● **22.90**. Выполните упражнение 22.89 для реализации алгоритма выталкивания превосходящего потока с использованием очереди FIFO (см. свойство 22.13).

22.88. Расширьте таблицу 22.3 таким образом, чтобы она включала реализации, которые используют подход построения всех аугментальных путей, описанные в упражнении 22.37.

●● **22.92**. Докажите, что время прогона метода, описанного в упражнении 22.91, есть $O(\sqrt{V}\,E)$ для поиска в ширину.

○ **22.93**. Выполните эмпирические исследования с целью построения кривой ожидаемого числа ребер в максимальном сочетании в случайных двудольных графах с $V + V$ вершинами и E ребрами для разумно выбранного множества значений V и подходящего числа значений E, достаточного для получения гладкой кривой, ведущей из нуля в V.

○ **22.94**. Докажите теорему Менгера (свойство 22.20) для неориентированных графов.

● **22.95**. Докажите, что минимальное число вершин, удаление которых нарушает связь двух вершин в орграфе, равно максимальному числу путей между этими двумя вершинами, не пересекающимися по вершинам. *Указание*: Воспользуйтесь преобразованием, предусматривающим разделение вершин, подобное показанному на рис. 22.32.

● **22.96**. Расширьте полученное вами доказательство упражнения 22.95 на неориентированные графы.

22.97. Постройте класс реберной связности для АТД графа из главы 17, конструктор которого использует алгоритм, описанный в этом разделе и предназначенный для поддержки общедоступной функции-элемента, возвращающей сведения о связности.

22.98. Расширьте полученное вами решение упражнения 22.97 с таким расчетом, чтобы в векторе, задаваемом пользователем, содержалось минимальное множество ребер, способных разрушить связность графа. На какой размер этого вектора должен рассчитывать пользователь?

● **22.99**. Разработайте алгоритм вычисления реберной связности орграфов (минимальное число ребер, удаление которых из орграфа лишает его свойства сильной связности). Постройте реализацию класса, основанного на предложенном вами алгоритме для АТД орграфа, описание которого приводится в главе 19.

● **22.100**. Разработайте алгоритм, в основе которых лежат решения, полученные в упражнениях 22.95 и 22.96, касающиеся вершинной связности орграфов и неориентированных графов. Постройте реализации классов, соответственно, на базе алгоритма для АТД орграфа, описание которого приводится в главе 19, и на базе алгоритма для АТД графа, описание которого приводится в главе 17 (см. упражнения 22.97 и 22.98).

22.101. Покажите, как можно выявить вершинную связность орграфа путем решения $V \lg V$-задач о максимальном потоке в сети с единичной пропускной способностью ребер. *Указание*: Воспользуйтесь теоремой Менгера и двоичным поиском.

○ **22.102**. Проведите эмпирические исследования на базе полученного вами решения упражнения 22.97 с целью получить сведения о связности различных графов (см. упражнения 17.63–17.76).

▷ **22.103**. Дайте формулировку задачи о максимальном потоке в транспортной сети, представленной на рис. 22.10, в терминах линейного программирования.

○ **22.112**. Сформулируйте в концепциях линейного программирования задачу двудольного сочетания, представленную на рис. 22.37.

22.5. Потоки минимальной стоимости

Нет ничего удивительного в том, что конкретная задача о максимальном потоке имеет несколько решений. Это обстоятельство дает нам право задать вопрос, можем ли мы ввести дополнительные критерии с целью выбора какого-то одного из них. Например, вполне понятно, что существует некоторое множество решений задач определения потоков в сети с единичной пропускной способностью ребер, представленной на рис. 22.22; возможно, по тем или иным причинам мы предпочитаем решение, использующее наименьшее число ребер, или решение, предусматривающее построение кратчайших путей, возможно, мы хотим знать, существует ли решение, использующее непересекающиеся пути. Эти задачи сложнее, чем стандартные задача о максимальном потоке, они охватываются более общей моделью, известной как *задача о потоке с минимальной стоимостью (minicost flow problem)*.

Как и в случае задачи о максимальном потоке, существуют различные эквивалентные способы постановки задачи о потоке с минимальной стоимостью. В этом разделе мы подробно рассмотрим стандартную постановку этой задачи, а различные сведения к другим задачам мы обсудим в разделе 22.7.

В частности, мы воспользуемся *моделью максимального потока с минимальной стоимостью (mincost-maxflow model)*: мы вводим понятие типа ребра, которое означает стоимость ребра, выраженную целым числом, используем стоимость ребра для определения стоимости потока естественным образом, а затем ставим задачу определения максимального потока с минимальной стоимостью. Как станет ясно из дальнейшего, мы не только получим эффективный алгоритм решения этой задачи, но и построим модель решения этой задачи, получившую широкое применение.

Определение 22.8. Стоимость потока (flow cost) *через ребро в транспортной сети со стоимостями ребер есть произведение потока в этом ребре и стоимости.* **Стоимость (cost)** *потока есть сумма стоимостей потоков в ребрах этого потока.*

И в этом случае мы полагаем, что в качестве пропускных способностей употребляются целые числа, меньшие M. Мы также полагаем, что стоимостью ребер являются отрицательные числа, меньшие C. (Отказ от использования отрицательных стоимостей мотивируется главным образом соображениями удобства, о чем подробнее будет сказано в разделе 22.7.) Как и ранее, мы присваиваем специальные обозначения этим значениям верхней границы, поскольку от них зависят времена прогона некоторых алгоритмов. После сделанных предположений формулировка задачи, которую мы намереваемся решить, не представляет трудностей.

Максимальный поток с минимальной стоимостью. Пусть задана транспортная сеть, ребра которой наделены стоимостями, найти такой максимальный поток, стоимость которого меньше стоимости любого другого максимального потока.

Рисунок 22.40 служит иллюстрацией различных максимальных потоков в транспортной сети со стоимостями, в том числе и максимальный поток с минимальной стоимостью. Разумеется, вычисление минимальной стоимости по трудоемкости не уступает вычислению максимального потока, с которым мы имели дело в разделах 22.2 и 22.3. И в самом деле, стоимость добавляет новое измерение, которое привносит новые трудноразрешаемые проблемы. Но даже если это так, то мы можем справиться с этими трудностями с

помощью обобщенного алгоритма, который подобен алгоритму построения аугментальных путей при решении задачи о максимальном потоке.

Другие многочисленные задачи сводятся к задаче о максимальном потоке минимальной стоимости. Например, следующая постановка задачи представляет интерес, поскольку она охватывает задачу о распределении товаров, которую мы изучали в начале данного раздела.

Допустимый поток минимальной стоимости. Напомним, что мы определяем поток в сети, содержащей вершины с весами (запас, если вес положительный, и спрос, если вес отрицательный) как *допустимый (feasible)*, если сумма весов вершин отрицательна и разность между притоком в каждую вершину и истечением из нее равна нулю. Задача заключается в том, чтобы в такой сети найти допустимый поток минимальной стоимости.

Для описания сетевой модели решение задачи о допустимом потоке минимальной стоимости мы для краткости используем термин *распределительная сеть (distribution network)* для обозначения "транспортной сети с заданными пропускными способностями и стоимостями ребер и весами запаса или спроса в вершинах".

В приложениях, осуществляющих распределение товаров, вершины с запасом соответствуют складам, вершины со спросом – розничным торговым точкам, ребра – маршрутам перевозок, величины запаса или спроса соответствуют количеству поставляемого и получаемого материала, а пропускные способности ребер – числу и грузоподъемности грузовых машин, курсирующих на том или ином маршруте. Естественной интерпретацией стоимости каждого ребра может служить стоимость перевозки единицы товара по этому ребру (стоимость пробега грузовой машины при перевозке единицы товара по соответствующему маршруту). Если задан поток, то стоимость потока через то или иное ребро составляет некоторую часть стоимости продвижения потока по всей сети, которую мы можем приписывать этому ребру. Если задано количество материала, который должен быть доставлен по некоторому заданному ребру, то мы можем вычислить стоимость доставки, умножив стоимость доставки единицы товара на его количество. Выполняя такое вычисление для каждого ребра и складывая полученные произведения, мы получим общую стоимость доставки товара, которую мы намерены минимизировать.

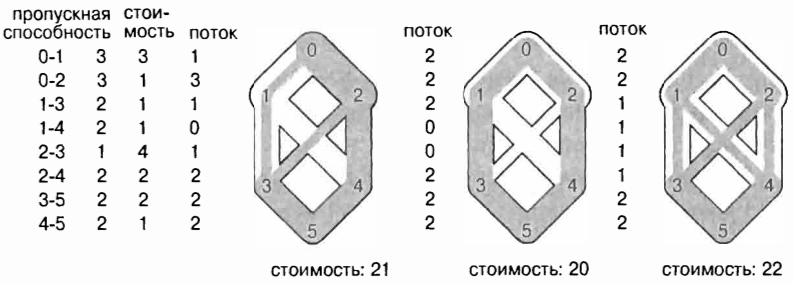


РИСУНОК 22.40. МАКСИМАЛЬНЫЕ ПОТОКИ В ТРАНСПОРТНЫХ СЕТЯХ С РЕБРАМИ, НАДЕЛЕННЫМИ СТОИМОСТЯМИ

Все эти потоки имеют одну и ту же (максимальную) величину, но их стоимости (сумма произведений потоков в ребрах на стоимость соответствующего ребра) различны. Максимальный поток на центральной диаграмме имеет минимальную стоимость (нет ни одного максимального потока с более низкой стоимостью).

Свойство 22.22. *Задача о допустимом потоке минимальной стоимости и задача о максимальном потоке с минимальной стоимостью эквивалентны.*

Доказательство: Непосредственно следует из соответствия, использованного при доказательстве свойства 22.18 (см. также упражнение 22.76). ■

Как следствие этой эквивалентности и в силу того, что задача о допустимом потоке с минимальной стоимостью непосредственно моделирует задачи о распределении товаров и многие другие приложения, мы употребляем термин *поток минимальной стоимости (minicost flow)* для обозначения обеих задач в контексте, в котором мы будем ссылаться на любую из них. Сведения к другим задачам мы рассмотрим в разделе 22.7.

Чтобы иметь возможность использовать стоимости ребер в транспортных сетях, введем целочисленный элемент данных **pcost** в класс **EDGE** из раздела 22.1 и функцию-элемент **cost()**, возвращающую значение стоимости клиенту. Программа 22.8 представляет собой клиентскую функцию, которая вычисляет стоимость потока в графе, в котором построены указатели на такие ребра. Как и в случаях, когда мы работаем с максимальными потоками, целесообразно реализовать функцию, проверяющую, что величины притоков и истечений соответствуют в каждой вершине, а структуры данных непротиворечивы (см. упражнение 22.12).

Программа 22.8. Вычисление стоимости потока

Эту функцию можно включить в программу 22.2. Она возвращает стоимость сетевого потока за счет суммирования произведения стоимости на величину потока для всех ребер, обладающих положительной пропускной способностью, позволяя при этом рассматривать ребра без пропускной способности как фиктивные.

```
static int cost(Graph &G)
{ int x = 0;
  for (int v = 0; v < G.V(); v++)
    {
      typename Graph::adjIterator A(G, v);
      for (Edge* e = A.beg(); !A.end(); e = A.nxt())
        if (e->from(v) && e->costRto(e->w()) < C)
          x += e->flow()*e->costRto(e->w());
    }
  return x;
}
```

Первый шаг в разработке алгоритмов решения задачи выявления потока с минимальной стоимостью состоит в таком расширении определения остаточных сетей, чтобы они включали стоимость ребер.

Определение 22.9. *Пусть задан поток в транспортной сети со стоимостями ребер,* **остаточная сеть (residual network)** *для этого потока содержит те же вершины, что и исходная сеть, и одно или два ребра этой остаточной сети соответствуют каждом ребру в исходной сети и определяются следующим образом: пусть для каждого ребра* **u-v** *в исходной сети* **f** *есть поток,* **c** *есть пропускная способность и* **x** *— стоимость. Если* **f** *положителен, ребро* **u-v** *включается в остаточную сеть, при этом ему присваивается пропускная способностью* **f**, *и стоимость* **-x**; *если* **f** *меньше, чем* **c**, *ребро* **u-v** *включается в остаточную сеть с пропускной способностью* **c-f**, *со стоимостью* **-x**.

Это определение почти идентично определению 22.4, однако отличия весьма существенны. Ребра в остаточной сети, представляющие обратные ребра, имеют *отрицательную* стоимость. Чтобы реализовать это соглашение, мы используем следующую функцию-элемент в классе ребра:

```
int costRto(int v)
  { return from(v) ? -pcost : pcost; }
```

Обход обратных ребер соответствует удалению потока из соответствующего ребра исходной сети, так что стоимость изменяется на соответствующую величину. В связи с наличием стоимостей ребер, эти сети могут иметь циклы с отрицательной стоимостью. Понятие отрицательных циклов, которое показалось нам несколько искусственным при первой встрече с ним, когда мы рассматривали его в контексте алгоритмов построения кратчайшего пути, играет фундаментальную роль в алгоритмах, как сейчас можно будет в этом убедиться. Мы рассмотрим два алгоритма, оба они основаны на следующем условии оптимальности.

Свойство 22.23. *Максимальный поток является* **максимальным потоком минимальной стоимости (mincost maxflow)** *тогда и только тогда, если его остаточная сеть не содержит (ориентированного) цикла с отрицательной стоимостью.*

Доказательство: Предположим, что задан максимальный поток минимальной стоимости, остаточная сеть которого содержит цикл с отрицательной стоимостью. Пусть x есть пропускная способность ребра с минимально пропускной способностью в цикле. Увеличим поток, добавляя x в ребра в потоке, соответствующем ребрам с положительной стоимостью в остаточной сети (прямые ребра) и вычитая x из ребер, соответствующих ребрам с отрицательной стоимостью в остаточной сети (обратные ребра). Эти изменения не оказывают влияния на разность между притоком и истечением любой вершины, однако они меняют стоимость сети на стоимость цикла, умноженную на x, которая принимает отрицательное значение, что противоречит условию, согласно которому стоимость исходного потока минимальна.

Чтобы доказать обратное, предположим, что существует максимальный поток F без циклов отрицательной стоимости, стоимость которых не является минимальной, и рассмотрим любой максимальный поток M с минимальной стоимостью. В силу рассуждений, аналогичных использованным при доказательстве теоремы о разложении потоков (свойство 22.2), мы можем найти самое большее E ориентированных циклов таких, что добавление этих циклов к потоку F дает поток M. Тем не менее, поскольку F не имеет отрицательных циклов, эта операция не может понизить стоимости потока F, т.е. получаем противоречие. Другими словами, мы должны быть способными преобразовать F в M за счет наращивания циклов, однако мы не можем это сделать, поскольку нет циклов отрицательной стоимости, которые можно было бы использовать для понижения стоимости потока. ■

Это свойство немедленно приводит к простому обобщенному алгоритму решения задачи о потоке минимальной стоимости, получившего название *алгоритма вычеркивания циклов (cycle-canceling algorithm)*:

Найти максимальный поток. Увеличить поток в произвольном цикле с отрицательной стоимости в остаточной сети, продолжая эту процедуру до тех пор, пока не останется ни одного такого цикла.

Этот метод объединяет механизмы, которые были разработаны на протяжении всей этой и предыдущих глав, обеспечивающие построение эффективных алгоритмов решения широкого класса задач, подпадающих под модель потока минимальной стоимости. Подобно нескольким другим обобщенным методам, с которыми нам приходилось сталкиваться, он допускает несколько различных реализаций, поскольку методы отыскания начального максимального потока и отыскания циклов с отрицательной стоимости не описаны. На рис. 22.41 показан пример вычисления максимального потока, который использует алгоритм вычеркивания циклов.

Поскольку мы уже разработали алгоритмы вычисления максимального потока и поиска отрицательных циклов, мы сразу же получаем реализацию алгоритма вычеркивания циклов, представленную программой 22.9. Мы используем любую реализацию максимального потока для вычисления начального максимального потока и алгоритм Беллмана-Форда поиска отрицательных циклов (см. упражнение 22.108). К двум этим реализациям в такую реализацию остается только добавить цикл, обеспечивающий наращивание потоков в циклах на графах.

Программа 22.9. Алгоритм вычеркивания циклов

Этот класс решает задачу вычисления максимального потока минимальной стоимости методом вычеркивания отрицательных циклов. Она использует класс **MAXFLOW** для поиска максимального потока и приватную функцию-элемент **negcyc** (см. упражнение 22.108) для нахождения отрицательных циклов. Если отрицательный цикл существует, этот программный код находит такой цикл, вычисляет максимальную величину потока, который нужно протолкнуть через этот цикл, и делает это. Функция **augment** аналогична используемой в программе 22.3, которая была разработана с расчетом (мы оказались предусмотрительными!) на работу в условиях, когда путь оказывается циклом.

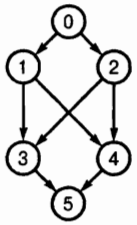
```
template <class Graph, class Edge> class MINCOST
{ const Graph &G;
   int s, t;
   vector<int> wt;
   vector <Edge *> st;
   int ST(int v) const;
   void augment(int, int);
   int negcyc(int);
   int negcyc();
public:
   MINCOST(const Graph &G, int s, int t) : G(G),
     s(s), t(t), st(G.V()), wt(G.V())
   { MAXFLOW<Graph, Edge>(G, s, t);
     for (int x = negcyc(); x != -1; x = negcyc())
       { augment(x, x); }
   }
};
```

Мы можем убрать вычисления максимального потока из алгоритма вычеркивания циклов, добавив фиктивное ребро, идущее из истока в сток, и назначив ему стоимость, превосходящую стоимость любого пути исток-сток в сети (например, *VC*) и поток, величина которого больше величины максимального потока (например, больше, чем истечение истока). По окончании этой настройки алгоритм вычеркивания циклов отводит максимально возможную величину потока из фиктивного ребра, в результате чего полученный поток становится максимальным потоком. Технология вычисления потока минимальной

РИСУНОК 22.41. ОСТАТОЧНЫЕ СЕТИ (ВЫЧЕРКИВАНИЕ ЦИКЛОВ) *Каждый из потоков, показанных на этом рисунке, представляет собой максимальный поток в транспортной сети в верхней части рисунка, но только поток, изображенный в нижней части является максимальным потоком минимальной стоимости. Чтобы найти его, мы начинаем вычисления, имея произвольный максимальный поток, и наращиваем поток, используя отрицательные циклы. Стоимость начального максимального потока (второй вверху) равна **22** единицам, но он не является максимальным потоком минимальной стоимости, поскольку остаточная сеть (показана справа) содержит три отрицательных цикла. В этом примере мы производим наращивание потока вдоль пути **4-1-0-2-4**, чтобы получит максимальный поток со стоимостью **21** (третий сверху), который все еще сохраняет один отрицательный цикл. Наращивание вдоль этого цикла дает поток минимальной стоимости (внизу). Обратите внимание на тот факт, что наращивание вдоль пути **3-2-4-1-3** дало бы нам максимальный поток минимальной стоимости всего за один шаг.*

	пропускная способность	стоимость	поток
0-1	3	3	0
0-2	3	1	0
1-3	2	1	0
1-4	2	1	0
2-3	1	4	0
2-4	2	2	0
3-5	2	2	0
4-5	2	1	0

0-1 3
0-2 3
1-3 2
1-4 2
2-3 1
2-4 2
3-5 2
4-5 2

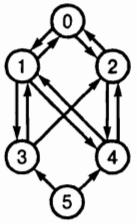


начальная величина максимального потока

	пропускная способность	стоимость	поток
0-1	3	3	2
0-2	3	1	2
1-3	2	1	1
1-4	2	1	1
2-3	1	4	1
2-4	2	2	1
3-5	2	2	2
4-5	2	1	2

суммарная стоимость: 22

0-1 1 1-0 2
0-2 1 2-0 2
1-3 1 3-1 1
1-4 1 4-1 1
3-2 1
2-4 1 4-2 1
5-3 2
5-4 2



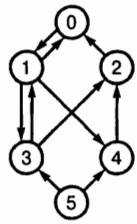
отрицательные циклы: 4-1-0-2-4
3-2-0-1-3
3-2-4-1-3

увеличение на +1 на пути 4-1-0-2-4 (стоимость -1)

	пропускная способность	стоимость	поток
0-1	3	3	1
0-2	3	1	3
1-3	2	1	1
1-4	2	1	0
2-3	1	4	1
2-4	2	2	2
3-5	2	2	2
4-5	2	1	2

суммарная стоимость: 21

0-1 2 1-0 1
2-0 3
1-3 1 3-1 1
1-4 2
3-2 1
4-2 2
5-3 2
5-4 2



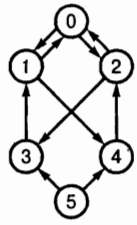
отрицательные циклы: 3-2-0-1-3

увеличение на +1 на пути 3-2-0-1-3 (стоимость -1)

	пропускная способность	стоимость	поток
0-1	3	3	2
0-2	3	1	2
1-3	2	1	2
1-4	2	1	0
2-3	1	4	0
2-4	2	2	2
3-5	2	2	2
4-5	2	1	2

суммарная стоимость: 20

0-1 1 1-0 2
0-2 1 2-0 2
3-1 2
1-4 2
2-3 1
4-2 2
5-3 2
5-4 2



стоимости показана на рис. 22.42. В примере, представленном на этом рисунке, мы используем начальный поток, равный максимальному потоку, для того, чтобы было ясно, что этот алгоритм просто вычисляет другой поток той же величины, но более низкой стоимости (в общем случае мы не знаем величины этого потока, поэтому некоторый поток все еще остается в фиктивном ребре на момент завершения работы алгоритма; этот поток мы игнорируем). Из рисунка понятно, что некоторые аугментальные циклы содержат фиктивное ребро и увеличивают поток в сети; другие циклы не содержат фиктивного ребра и уменьшают стоимость. В конечном итоге, мы получаем максимальный поток; на этом этапе все аугментальные циклы уменьшают стоимость, не меняя величины потока, в отличие от ситуации, когда мы начали вычисления максимального потока.

С технической точки зрения, для алгоритма вычеркивания циклов использование инициализации фиктивным потоком является ни более, и ни менее универсальным, нежели использование инициализации максимальным потоком. Первая затрагивает все алгоритмы вычисления максимального потока посредством построения аугментальных путей, однако с помощью алгоритма могут быть вычислены не все максимальные потоки (см. упражнение 22.40). С одной стороны, использование такой технологии может привести в утрате всех преимуществ, обеспечиваемых сложным алгоритмом вычисления максимального потока. С другой стороны, мы можем приобрести в плане снижения затрат в процессе построения максимального потока. На практике инициализация фиктивным потоком широко используется в силу простоты ее реализации.

Как и в случае вычисления максимальных потоков, существование этого обобщенного алгоритма служит гарантией того, что каждая задача о максимальном потоке минимальной стоимости (с заданными пропускными способностями ребер и целочисленными стоимостями) имеет решение, обеспечивающее целочисленные величины всех потоков, и этот алгоритм позволяет получить это решение (см. упражнение 22.107). Учитывая упомянутый факт, легко установить верхнюю границу времени, необходимого для выполнения произвольного алгоритма вычеркивания циклов.

Свойство 22.24. *Число аугментальных циклов, необходимых для выполнения обобщенного алгоритма вычеркивания циклов, не превосходит ECM.*

Доказательство: В худшем случае каждое ребро в начальном потоке имеет пропускную способность M, стоимость C и заполнено. Каждый цикл уменьшает эту стоимость, по меньшей мере, на 1. ■

Следствие. *Время, необходимое для решения задачи о максимальном потоке минимальной стоимости в разреженной сети, есть* $O(V^3CM)$.

Доказательство: Непосредственно следует из умножения числа аугментальных циклов в худшем случае на затраты на их поиск, которые определяет для худшего случая алгоритм Беллмана-Форда (см. свойство 21.20). ■

Подобно методам аугментальных путей, эта оценка времени выполнения алгоритма чрезвычайно пессимистична, поскольку она предполагает не только ситуацию худшего случая, в условиях которой мы должны выполнить громадное число циклов для минимизации стоимости, но и в силу того, что мы попадаем в другую ситуацию худшего случая, когда для нахождения каждого цикла требуется проверить огромное количество ребер.

РИС. 22.42. АЛГОРИТМ ВЫЧЕРКИВАНИЯ ЦИКЛОВ БЕЗ НАЧАЛЬНОГО МАКСИМАЛЬНОГО ПОТОКА

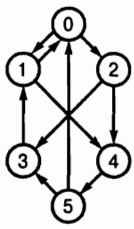
*Данная последовательность диаграмм служит иллюстрацией вычисления максимального потока минимальной стоимости, начинающегося с первоначально нулевого потока, с применением алгоритма вычеркивания циклов, который использует фиктивное ребро из истока в сток в остаточной сети с неограниченной пропускной способностью и неограниченными отрицательными стоимостями. Фиктивное ребро превращает любой аугментальный путь из **0** в **5** отрицательным циклом (однако, мы его игнорируем при наращивании потока и вычислении его стоимости). Наращивание вдоль этого пути ведет к увеличению потока, как это имеет место в рамках алгоритмов построения аугментальных путей (три верхних ряда). Когда нет циклов, в состав которых входит фиктивное ребро, то в остаточной сети отсутствуют пути из истока в сток, следовательно, мы имеем максимальный поток (третий ряд сверху). В этой точке наращивание потока вдоль отрицательного цикла уменьшает стоимость потока без изменения его величины (нижний ряд). В этом примере мы вычисляем максимальный поток, затем уменьшаем его стоимость; однако, не это главное. Например, алгоритм может увеличивать поток в отрицательном цикле **1-4-5-3-1** вместо цикла **0-1-4-5-0** на втором шаге. Поскольку каждое наращивание либо увеличивает поток, либо уменьшает стоимость, мы всегда в итоге получаем максимальный поток минимальной стоимости.*

увеличение на +2 на пути 0-1-3-5-0 (стоимость +6)

пропускная способность		стоимость	поток
0-1	3	3	**2**
0-2	3	1	**0**
1-3	2	1	**2**
1-4	2	1	**0**
2-3	1	4	**0**
2-4	2	2	**0**
3-5	2	2	**2**
4-5	2	1	**0**
5-0	*	*	

суммарная стоимость: 12

0-1 1 1-0 2
0-2 3
3-1 2
1-4 2
2-3 1
2-4 2
5-3 2
4-5 2
5-0 *

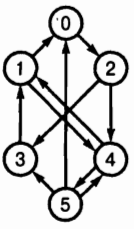


увеличение на +1 на пути 0-1-4-5-0 (стоимость +5)

пропускная способность		стоимость	поток
0-1	3	3	**3**
0-2	3	1	**0**
1-3	2	1	**2**
1-4	2	1	**1**
2-3	1	4	**0**
2-4	2	2	**0**
3-5	2	2	**2**
4-5	2	1	**1**
5-0	*	*	

суммарная стоимость: 17

1-0 3
0-2 3
3-1 2
1-4 1 4-1 1
2-3 1
2-4 2
5-3 2
4-5 1 5-4 1
5-0 *

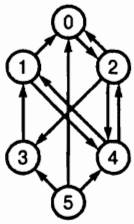


увеличение на +1 на пути 0-2-4-5-0 (стоимость +4)

пропускная способность		стоимость	поток
0-1	3	3	3
0-2	3	1	1
1-3	2	1	2
1-4	2	1	1
2-3	1	4	0
2-4	2	2	1
3-5	2	2	2
4-5	2	1	2
5-0	*	*	

суммарная стоимость: 21

1-0 3
0-2 2 2-0 1
3-1 2
1-4 1 4-1 1
2-3 1
2-4 1 4-2 1
5-3 2
5-4 2
5-0 *

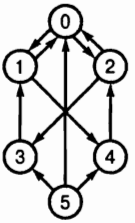


увеличение на +1 на пути 4-1-0-2-4 (стоимость -1)

пропускная способность		стоимость	поток
0-1	3	3	2
0-2	3	1	2
1-3	2	1	2
1-4	2	1	0
2-3	1	4	0
2-4	2	2	2
3-5	2	2	2
4-5	2	1	2
5-0	*	*	

суммарная стоимость: 20

0-1 1 1-0 2
0-2 1 2-0 2
3-1 2
1-4 2
2-3 1
4-2 2
5-3 2
5-4 2
5-0 *



Во многих ситуациях, имеющих место на практике, мы используем относительно небольшое число циклов, обнаружить которых относительно нетрудно, и алгоритм вычеркивания циклов оказывается достаточно эффективным.

Можно разработать стратегию выявления циклов отрицательной стоимости, гарантирующую, что число использованных циклов отрицательной стоимости меньше *VE* (см. *раздел ссылок*). Этот результат имеет важное значение, ибо он устанавливает тот факт, что задача о потоке минимальной стоимости имеет решение (равно как и все другие задачи, которые сводятся к ней). Тем не менее, на практике предпочтение обычно отдается реализациям, которые, признавая (теоретически) оценки для худшего случая, используют значительно меньше итераций при решении задач, возникающих на практике, чем их число, которое прогнозируется границами для худшего случая.

Задача о максимальном потоке минимальной стоимости представляет собой наиболее общую модель решения задач из числа изученных до сих пор, поэтому вызывает удивление тот факт, что мы можем решить ее, воспользовавшись столь простой реализацией. Ввиду особой важности этой модели, были разработаны и подробно исследованы другие многочисленные реализации метода вычеркивания циклов и многие другие методы. Программа 22.9 отличается удивительной простотой и представляет собой эффективную отправную точку для вычислений, однако в ней имеются два слабых места, которые могут послужить причиной ее низкой производительности. Во-первых, каждый раз, когда мы ищем отрицательный цикл, мы начинаем с самого начала. Можно ли сохранить промежуточную информацию, полученную во время поиска одного отрицательного цикла, которая может оказаться полезной при поиске следующего цикла? Во-вторых, программа 22.9 использует только первый отрицательный цикл, который находит алгоритм Беллмана-Форда. Сможем ли мы направить поиск на выявление отрицательных циклов с особыми свойствами? В разделе 22.6 мы рассмотрим более совершенную реализацию, которая, будучи обобщенной, дает ответы на оба эти вопроса.

Упражнения

22.105. Расширьте класс допустимых потоков из упражнения 22.44, обеспечив манипулирование стоимостями. Воспользуйтесь классом **MINCOST** для решения задачи о допустимых потоках минимальной стоимости.

▷ **22.106**. Пусть дана некоторая транспортная сеть, ребра которой не обладают максимальной пропускной способностью и максимальной стоимостью. Дайте более точную верхнюю границу, чем *ECM*, для стоимости максимального потока.

22.107. Докажите, что если все пропускные способности и стоимости суть целые числа, то задача о минимальной стоимости имеет решение, в рамках которого все потоки имеют целочисленные значения.

22.108. Реализуйте функцию **negcyc()** для программы 22.9, воспользовавшись для этой цели алгоритмом Беллмана-Форда (см. упражнение 21.134).

▷ **22.109**. Внесите изменения в программу 22.9, допускающие инициализацию потоком в фиктивном ребре вместо вычисления потока.

○ **22.110**. Дайте все возможные последовательности аугментальных циклов, которые могли бы быть показаны на рис. 22.41.

○ **22.111**. Дайте все возможные последовательности аугментальных циклов, которые могли бы быть показаны на рис. 22.42.

○ **22.112**. Покажите в стиле рис. 22.41 поток и остаточные сети после каждого наращивания в случае использования реализации программы 22.9 с алгоритмом вычеркивания циклов для отыскания потока минимальной стоимости в транспортной сети, изображенной на рис. 22.10, в которой ребрам **0-2** и **0-3** назначена стоимость **2**, ребрам **2-5** и **3-5** – стоимость **3**, ребру **1-4** – стоимость **4**, а всем остальным ребрам – стоимость **1**. Предполагается, что максимальный поток вычисляется с помощью алгоритма построения кратчайшего аугментального пути.

22.113. Выполните упражнение 22.112 в предположении, что в программу внесены изменения, позволяющие начинать вычисления в ситуации, когда максимальный поток протекает в фиктивном ребре из истока в сток, как показано на рис. 22.42.

22.114. Дополните полученные вами решения упражнений 22.6 и 22.7 вычислениями стоимостей в транспортных сетях.

22.115. Дополните полученные вами решения упражнений 22.9–22.11 вычислениями стоимостей в сетях. Назначьте каждому ребру стоимость, приблизительно пропорциональную эвклидовому расстоянию между вершинами, которые это ребро соединяет.

22.6. Сетевой симплексный алгоритм

Основу длительности выполнения алгоритма вычеркивания циклов составляют не только количество циклов отрицательной стоимости, которые этот алгоритм использует для снижения стоимости потока, но также и время, которое этот алгоритм использует для нахождения каждого такого цикла. В этом разделе мы рассмотрим базовый подход, который не только резко снижает затраты на выявление отрицательных циклов, но и использует методы уменьшения числа итераций. Такая реализация алгоритма вычеркивания циклов известна под названием *сетевого симплексного (network simplex)* алгоритма. Он основан на поддержке древовидной структуры данных и на присвоении стоимостям таких значений, которые обеспечили быстрое выявление отрицательных циклов.

Чтобы дать описание сетевого симплексного алгоритма, мы начнем с того, что отметим, что в отношении к любому потоку каждое ребро **u-v** сети может пребывать в одном из трех состояний (см. рис. 22.43):

- *Пустом (empty)*, так что поток можно протолкнуть только из **u** в **v**.
- *Полном (full)*, так что поток можно протолкнуть только из **v** в **u**.
- *Частично заполненном (partial)* (т.е. ни пустое, ни заполненное), так что поток можно проталкивать в обоих направлениях.

С такой классификацией мы уже сталкивались при использовании остаточных сетей на протяжении текущей главы. Если ребро **u-v** пустое, то **u-v** включено в остаточную сеть, однако **v-u** не включено; если ребро **u-v** заполнено, то **v-u** включено в остаточную сеть, а **u-v** не включено; если ребро **u-v** заполнено частично, то в остаточную сеть включены и **u-v**, и **v-u**.

Определение 22.10. *Пусть дан максимальный поток без циклов частично заполненных ребер*, тогда **допустимое остовное дерево (feasible spanning tree)** *этого максимального потока представляет собой любое остовное дерево сети, которое содержит все частично заполненные ребра.*

В этом контексте мы игнорируем направление ребра в остовном дереве. Иначе говоря, любое множество $V-1$ ориентированных ребер, которые соединяют V вершин между собой (при этом направление ребер игнорируется), составляет остовное дерево, а остовное дерево допустимо, если все ребра, не содержащиеся в дереве, являются полными или пустыми.

Первый шаг сетевого симплексного алгоритма заключается в построении остовного дерева. Один способ его построения предусматривает вычисление максимального потока, разбиение циклов частично заполненных ребер за счет наращивания потока, чтобы заполнить или опорожнить одно из его ребер, и последующее добавление пустых или полных ребер к остальным частично заполненным ребрам с целью построения остовного дерева. Пример такого процесса представлен на рис. 22.44. В другом варианте первый шаг предусматривает вычисление максимального потока в фиктивном ребре, соединяющем исток в сток. Далее, это ребро является единственным возможным частично заполненным ребром, и мы можем построить остовное дерево для данного потока в процессе любого просмотра графа. Пример такого остовного дерева приводится на рис. 22.45.

На данном этапе добавление в остовное дерево ребра, не принадлежащего ему, приводит к образованию цикла. Механизмом, составляющим основу сетевого симплексного алгоритма, является множество весов вершин, которое обеспечивает немедленную идентификацию тех ребер, которые, будучи включенными в остовное дерево, создают циклы отрицательной стоимости в остаточной сети. Мы будем называть эти веса вершин *потенциалами (potentials)* и использовать обозначение $\phi(v)$ в отношении потенциала, ассоциированного с вершиной **v**. В некоторых контекстах мы будем называть потенциалы функциями, определенными на вершинах, либо множеством целочисленных весов, предполагая при этом, что каждой вершине присвоен такой вес, либо вектором, проиндексированным именами вершин (поскольку обычно в реализациях мы храним их именно в таком виде).

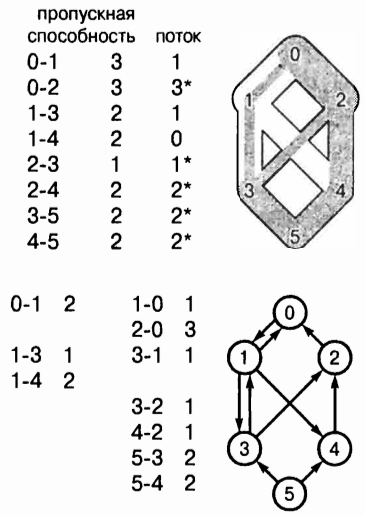


РИСУНОК 22.43. КЛАССИФИКАЦИЯ РЕБЕР

*По отношению к любому потоку ребро может быть пустым, полным или частично заполненным (ни пустое, ни заполненное). В этом потоке ребро **1-4** пустое; ребра **0-2**, **2-3**, **2-4**, **3-5** и **4-5** заполнены, ребра **0-1** и **1-3** частично заполнены. Наши соглашения дают два способа идентификации состояний ребер: в столбце потока вхождения 0 означают пустые ребра; вхождения, обозначенные звездочками, суть заполненные ребра; ребра, которые не обозначены ни 0, ни звездочками — это частично заполненные ребра. В остаточной сети (внизу) пустые ребра появляются только в правом столбце, а частично заполненные ребра — в обоих столбцах.*

Определение 22.11. *Пусть задан поток в транспортной сети, ребрам которой присвоены стоимости, причем $c(u,v)$ означает стоимость ребра* **u-v** *остаточной сети. Для любой потенциальной функции ϕ* **приведенная стоимость (reduced cost)** *ребра* **u-v** *остаточной сети относительно ϕ, которую мы обозначим как $c^*(u,v)$, по определению есть значение $c(u,v) - (\phi(u) - \phi(v))$.*

Иначе говоря, приведенная стоимость каждого ребра есть разность между фактической стоимостью ребра и разностью потенциалов вершин этого ребра. В реализации, связанной с распределением товаров, за потенциалом узла можно разглядеть конкретный смысл: если интерпретировать потенциал $\phi(u)$ как затраты на приобретение единицы товара в узле u, то полная стоимость $c(u,v) + \phi(u) - \phi(v)$ суть затраты на закупку товара в узле u, доставка товара в v и реализация товара в v.

Мы поддерживаем вектор **phi**, индексированный именами вершин, для хранения потенциалов вершин и вычисления приведенной стоимости ребра **v-w**, вычитая из стоимости ребра значения (**phi[v] - phi[w]**). То есть, нет необходимости где-то хранить приведенную стоимость ребра, поскольку ее очень легко вычислить.

При выполнении сетевого симплексного алгоритма мы используем допустимые остовные деревья с тем, чтобы определить потенциалы вершин таким образом, чтобы приведенные стоимости ребер относительно этих потенциалов давали прямую информацию о циклах с отрицательной стоимостью. По существу, мы поддерживаем допустимое остовное дерево во время выполнения алгоритма и устанавливаем такие значения потенциалов вершин, чтобы все ребра деревьев обладали приведенной стоимостью, равной нулю.

Свойство 22.25. *Мы говорим, что потенциалы вершин* **имеют силу, или допустимы (valid)**, *в отношении остовного дерева, если все древесные ребра имеют приведенную стоимость, равную нулю. Для всех имеющих силу потенциалов вершин любого остовного дерева величина приведенной стоимости каждого ребра сети является одной и той же.*

Доказательство: Пусть даны две различные потенциальные функции ϕ и ϕ', которые имеют силу в отношении заданного остовного дерева, покажем, что они различаются на аддитивную постоянную: т.е., что $\phi(u) = \phi'(u) + \Delta$ для всех u и некоторой постоянной Δ. Таким образом, $\phi(u) - \phi(v) = \phi'(u) - \phi'(v)$ для всех u и v. Отсюда следует, что все приведенные стоимости одинаковы для двух потенциальных функций.

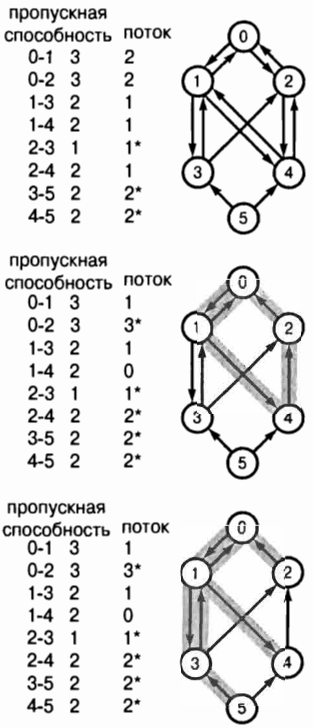


РИСУНОК 22.44. ОСТОВНОЕ ДЕРЕВО МАКСИМАЛЬНОГО ПОТОКА

Пусть задан максимальный поток (диаграмма вверху), мы можем построить некоторый максимальный поток, обладающий остовным деревом, таким, что ни одно из ребер, не включенных в остовное дерево, не является частично заполненным ребром, с помощью двухэтапного процесса, представленного в данном примере. Сначала мы разрушаем цикл частично заполненных ребер — мы разрушаем цикл ***0-2-4-1-0****, проталкивая через него поток величиной в одну единицу. Мы всегда можем заполнить или опорожнить таким способом, по меньшей мере, одно ребро; в рассматриваемом случае мы опорожняем ребро* ***1-4*** *и заполняем ребра* ***0-2*** *и* ***2-4*** *(диаграмма в центре). Затем мы включаем пустые и полные ребра во множество частично заполненных ребер, образующих остовное дерево; в рассматриваемом случае мы добавляем ребра* ***0-2***, ***1-4*** *и* ***3-5*** *(диаграмма внизу).*

Для любых двух вершин u и v, которые связаны между собой древесным ребром, должно иметь место равенство $\phi(v) = \phi(u) - c(u,v)$, исходя из следующих соображений. Если **u-v** есть древесное дерево, то $\phi(v)$ должна быть равно $\phi(u) - c(u,v)$, чтобы приведенная стоимость $c(u,v) - \phi(u) + \phi(v)$ была равна нулю; если **v-u** есть древесное ребро, то $\phi(v)$ должна быть равна $\phi(u) - c(v,u)$, чтобы приведенная стоимость $c(v,u) - \phi(v) + \phi(u)$ была равна нулю. Те же рассуждения справедливы и для ϕ', так что мы имеем $\phi'(v) = \phi'(u) - c(u,v)$.

Выполняя вычитание, находим, что $\phi(v) - \phi'(v) = \phi(u) - \phi'(u)$ для любых u и v, соединенных древесным ребром. Обозначив эту разность для любой вершины через Δ, и применив это равенство к ребрам любого дерева поиска, получаем $\phi(u) = \phi'(u) + \Delta$ для всех u. ■

пропускная способность	стои-мость	поток
0-1	3	3	0
0-2	3	1	0
1-3	2	1	0
1-4	2	1	0
2-3	1	4	0
2-4	2	2	0
3-5	2	2	0
4-5	2	1	0
0-5	6	9	4

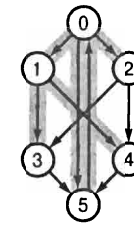


РИСУНОК 22.45. ОСТОВНОЕ ДЕРЕВО ДЛЯ ФИКТИВНОГО МАКСИМАЛЬНОГО ПОТОКА

*Если мы начинаем с потока фальшивого ребра из истока в сток, то это единственное возможное частично заполненное ребро, благодаря чему мы можем использовать любое остовное дерево из остальных ребер для построения остовного дерева для потока. В рассматриваемом примере ребра **0-5**, **0-1**, **0-2**, **1-3** и **1-4** составляют остовное дерево для исходного максимального потока. Все недревесные ребра остаются пустыми.*

Другой способ представления процесса определения некоторого множества имеющих силу потенциалов вершин заключается в том, что мы начинаем с того, что фиксируем одно значение, затем вычисляем эти значения для всех вершин, соединенных с заданной вершиной древесными деревьями, затем вычисляем эти значения для всех вершин, соединенных с этими вершинами, и т.д. Не имеет значения, в какой вершине мы начнем этот процесс, разность потенциалов между любыми двумя вершинами та же самая, и она определяется только структурой дерева. Мы подробно рассмотрим задачу вычисления потенциалов после того, как проведем исследования зависимости между приведенными стоимостями недревесных ребер и циклами отрицательной стоимости.

Свойство 22.26. *Мы говорим, что недревесное дерево есть* **подходящее (eligible)** *дерево, если цикл, который оно формирует с древесными деревьями, представляет собой цикл отрицательной стоимости. Ребро относится к категории подходящих тогда и только тогда, когда это ребро есть полное ребро положительной приведенной стоимости или пустое ребро отрицательной приведенной стоимости.*

Доказательство: Предположим, что ребро **u-v** создает цикл $\mathbf{t_1}$-$\mathbf{t_2}$-$\mathbf{t_3}$-...-$\mathbf{t_d}$-$\mathbf{t_1}$ с древесными ребрами $\mathbf{t_1}$-$\mathbf{t_2}$, $\mathbf{t_2}$-$\mathbf{t_3}$, ..., где **v** есть $\mathbf{t_1}$ и **u** есть $\mathbf{t_d}$. Из определения приведенной стоимости каждого ребра вытекает следующее:

$$c(u,v) = c^*(u,v) + \phi(u) - \phi(t_1)$$
$$c(t_1,t_2) = \phi(t_1) - \phi(t_2)$$
$$c(t_2,t_3) = \phi(t_2) - \phi(t_3)$$
$$\ldots$$
$$c(t_{d-1},u) = \phi(t_{d-1}) - \phi(u).$$

Сумма левых частей этих уравнений дает итоговую сумму цикла, а сумма в правой части сводится к $c^*(u,v)$. Другими словами, приведенная стоимость ребра дает стоимость цикла, таким образом, только описанные ребра могут дать циклы с отрицательной стоимостью. ■

Свойство 22.27. *Если даны поток и допустимое остовное дерево, в котором нет подходящих ребер, поток представляет собой поток минимальной стоимости.*

Доказательство: Если подходящие ребра отсутствуют, то в остаточной сети отсутствуют также циклы с отрицательной стоимостью, так что из условия оптимальности, определяемого свойством 22.23, следует, что поток является потоком минимальной стоимости. ■

Эквивалентная формулировка утверждает, что если задан поток и множество потенциалов вершин, таких, что все приведенные стоимости древесных ребер равны нулю, все полные недревесные ребра неотрицательны, а пустые недревесные ребра неположительны, то поток есть поток минимальной стоимости.

Если мы имеем подходящие ребра, мы можем выбрать одно из них и наращивать дерево вдоль цикла, которое оно образует с древесными деревьями с целью получения потока более низкой стоимости. Аналогично реализации из раздела 22.5, в которой мы применили метод отбрасывания циклов, выполняем обход соответствующего цикла с тем, чтобы определить величину потока, который мы можем протолкнуть, затем пройти еще раз по циклу, чтобы протолкнуть такую величину потока, которая обеспечивает заполнение или опорожнение, по меньшей мере, одного ребра.

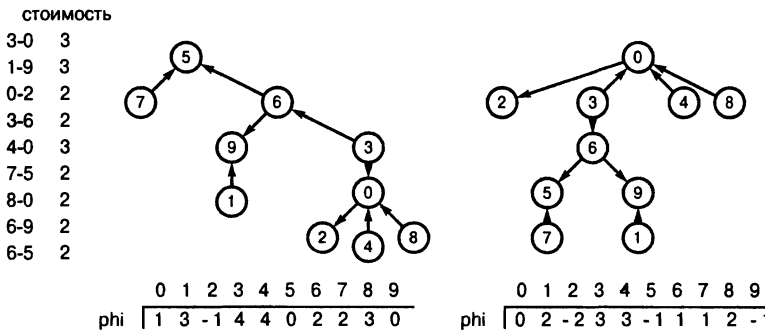


РИСУНОК 22.46. ПОТЕНЦИАЛЫ ВЕРШИН

Потенциалы вершин определяются структурой остовного дерева и исходными значениями потенциалов вершин, присваиваемых каждой вершине. В таблице слева показано множество ребер, образующих остовное дерево, состоящее из десяти вершин от ***0*** *до* ***9****. На диаграмме в центре изображено представление этого дерева, в котором в качестве корня выбрана вершина* ***5****, вершины, соединенные с вершиной* ***5*** *находятся на уровень ниже и т.д. Когда мы присваиваем корню значение потенциала, равное нулю, существует уникальное сочетание потенциалов, назначаемых другим узлам, при котором разность потенциалов вершин каждого ребра равна его стоимости. На диаграмме справа дано представление того же дерева, в котором в качестве корня выбрана вершина* ***0****. Потенциалы, которые мы получим, присваивая вершине* ***0*** *значение 0, отличаются от потенциалов, показанных на диаграмме в центре, на некоторое постоянное значение. Эта разность потенциалов используется во всех наших вычислениях: упомянутая разность остается одной и той же для любой пары потенциалов независимо от того, с какой вершины мы начнем вычисления (и независимо от того, какое значение мы присвоим вершине), следовательно, выбор исходной вершины и начального значения не играет роли.*

Если это подходящее ребро, которое мы использовали для построения цикла, оно перестает быть подходящим (его приведенная стоимость остается той же, однако из полного оно становится пустым или из пустого превращается в полное). Во всех других случаях оно становится частично заполненным. Присоединив его к дереву и удалив полное или пустое ребро из цикла, мы выполняем инвариантное условие, согласно которому никакое недревесное ребро не может быть частично заполненным, а само дерево есть допустимым остовным деревом. Мы также рассмотрим механику соответствующих вычислений в этом разделе несколько позже.

По сути дела, допустимые остовные деревья позволяют нам вычислить потенциал вершин, которые дают приведенные стоимости, которые, в свою очередь, дают допустимые ребра, которые, опять-таки, дают циклы отрицательной стоимости. Наращивание циклов отрицательной стоимости снижает стоимость потока и влечет к изменениям в структуре дерева. Изменения в структуре дерева обусловливают изменения потенциалов вершин, изменения потенциалов вершин приводят к изменению приведенных стоимостей ребер, а изменения приведенных стоимостей влекут за собой изменения во множестве подходящих ребер. Выполнив все эти изменения, мы можем выбрать другое подходящее ребро и начать процесс снова. Это общая реализация алгоритма вычеркивания циклов для решения задачи о потоке минимальной стоимости называется *сетевым симплексным* алгоритмом:

> Постройте подходящее остовное дерево и поддерживайте потенциалы вершины такими, чтобы все вершины остовного дерева имели нулевую приведенную стоимость. Добавьте подходящее ребро в дерево, нарастите поток в цикле, который оно образует с древесными ребрами, и вычеркните из дерева ребро, которое оказалось заполненным или опорожненным, после чего продолжайте этот процесс до тех пор, пока не окажется ни одного подходящего ребра.

Эта реализация является обобщенной, поскольку начальный выбор остовного дерева, метод поддержания потенциалов вершин и метод выбора подходящих ребер не определены. Стратегия выбора подходящих ребер определяет число итераций и выбирается с учетом затрат на реализацию различных стратегий такого выбора и на пересчет потенциалов вершин.

Свойство 22.28. *Если общий сетевой симплексный алгоритм завершает работу, это означает, что он вычисляет поток минимальной стоимости.*

Доказательство: Если указанный алгоритм прекращает работу, то он делает это потому, что в остаточной сети больше нет циклов отрицательной стоимости, а по свойству 22.23 это означает, что полученный максимальный поток обладает минимальной стоимостью. ■

Ситуация, когда алгоритм может не завершить свою работу, обусловлена возможностью того, что процесс наращивания потока в некотором цикле может заполнять или опорожнять сразу несколько циклов, из-за чего в дереве могут оставаться ребра, через которые невозможно проталкивание какого-либо потока. Если мы не можем протолкнуть поток, то не можем выполнить приведение цен, зато можем попасть в бесконечный цикл добавления и удаления ребер с целью получения фиксированной последовательности остовных деревьев. Было найдено несколько способов избежать подобного рода ситуаций,

и мы исследуем их в этом разделе несколько позже, после того как рассмотрим реализации более подробно.

Первый выбор, с необходимостью которого мы столкнемся при разработке сетевого симплексного алгоритма, касается представления остовного дерева. Построение остовного дерева требует решения следующих вычислительных задач:

- Вычисление потенциалов вершин.
- Наращивание потока в циклах (и выявление на нем порожних или заполненных ребер).
- Вставка нового ребра и удаление ребра из образованного при этом цикла.

Каждая из этих задач сама по себе представляет интересное упражнение в разработке структур данных и в проектировании алгоритмов. Существует несколько структур данных и многочисленные алгоритмы, обладающие различными рабочими характеристиками которые нам полезно изучить. Мы начнем это изучение, пожалуй, с простейшей доступной нам структуры данных — с которой мы впервые столкнулись в главе 1 (!) — т.е. с представления дерева в форме родительских связей. После того, как мы проведем анализ алгоритмов и реализаций, которые построены на основе представлении дерева в форме родительских связей для выше перечисленных задач и опишем, как следует их использовать в контексте сетевого симплексного алгоритма, мы обсудим альтернативные структуры данных и алгоритмы.

Подобно тому, как мы поступали в ряде других реализациях, рассмотренных в данной главе, начиная от реализации вычисления максимального пути методом построения аугментальных путей, мы сохраняем связи с сетевым представлением, обеспечивающим доступ к параметрам потока, а не простые индексы в представлении дерева, что позволит получать доступ к величинам потоков, не теряя времени на доступ к именам вершин.

Программа 22.10 представляет реализацию, которая присваивает потенциалы вершинам за время, пропорциональное V. В ее основе лежит следующая идея, проиллюстрированная на рис. 22.47. Мы начинаем с того, что выбираем любую вершину и в рекурсивном режиме вычисляем потенциалы ее предшественников, следуя вверх по родительскому списку до самого корня, которому по соглашению присваивается потенциал 0. Затем мы выбираем другую вершину и используем родительские связи для рекурсивного вычисления потенциалов ее предшественников. Рекурсивные вычисления заканчиваются, когда мы достигнем предшественника, потенциал которого известен; далее, при выходе из рекурсии мы возвращаемся по тому же пути, вычисляя потенциал каждого узла с использованием потенциала родителя. Процесс продолжается до тех пор, пока не будут вычислены значения потенциалов. Если мы уже прошли по какому-либо пути, то мы больше не посещаем ни одно из его ребер, следовательно, этот процесс протекает в течение времени, пропорционального числу вершин V.

Программа 22.10. Вычисление потенциала вершины

Рекурсивная функция **phiR** следует вверх по дереву вдоль родительской связи до тех пор, пока не выйдет на вершину, потенциал которой имеет силу (в соответствии с соглашением, потенциал корня всегда имеет силу), а затем, завершая рекурсивные вызовы, вычисляет потенциалы вершин, которые она проходит на обратном пути. Она маркирует каждую вершину, потенциал которой вычисляет, устанавливая ей значение метки, равное текущему значению функции **valid**.

```
int phiR(int v)
  {
   if (mark[v] == valid) return phi[v];
   phi[v] = phiR(ST(v)) - st[v]->costRto(v);
   mark[v] = valid;
   return phi[v];
  }
```

Пусть даны две вершины, их *LCA-предшественник* (least common ancestor - наименьший общий предшественник) есть корень минимального поддерева, которое содержит обе эти вершины. Цикл, который мы строим за счет добавления ребра, соединяющего две вершины, состоит из этого ребра плюс ребра двух путей, ведущие из двух этих узлов к их LCA-предшественнику. Этот цикл, образованный благодаря добавлению ребра **v-w**, проходит через **v-w** в **w**, затем вверх по дереву к LCA-предшественнику вершины **v** и **w** (его имя, скажем, **r**), затем вниз по дереву в **v**, таким образом, мы должны на этих двух путях рассматривать ребра в их противоположной ориентации. Программа 22.11 представляет собой реализацию этой идеи в виде функции, которая наращивает поток в заданном цикле и возвращает ребро, заполненное или опорожненное в процессе этого наращивания.

Как и раньше, мы наращиваем поток в цикле посредством одного обхода путей с тем, чтобы выявить максимальную величину потока, которую мы можем протолкнуть через их ребра, с последующим вторым обходом обоих этих путей с целью протолкнуть через них поток.

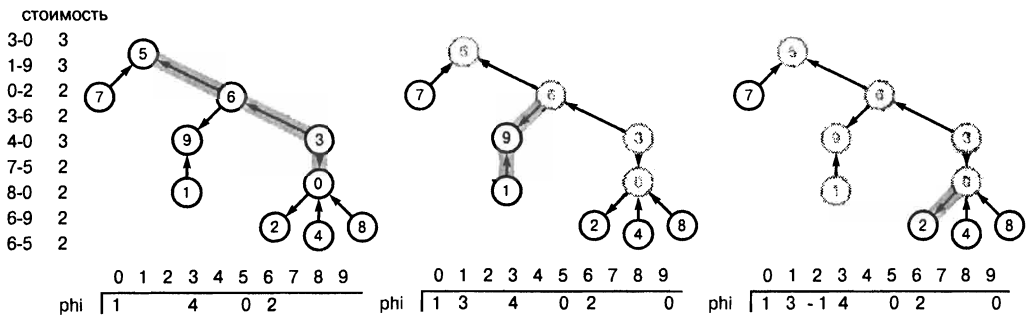


РИСУНОК 22.47. ВЫЧИСЛЕНИЕ ПОТЕНЦИАЛОВ С ИСПОЛЬЗОВАНИЕМ РОДИТЕЛЬСКИХ СВЯЗЕЙ

*Мы начнем вычисления с вершины **0**, пройдем вдоль пути в корень, установим значение **pt[5]** равным нулю, а затем, следуя вниз тем же путем, установим вершине **6** такое значение, чтобы **pt[6] - pt[5]** было равно стоимости ребра **6-5**, затем установим значение **pt[3]** таким, чтобы **pt[3] - pt[6]** было равным стоимости ребра **3-6** и т.д. (диаграмма слева). Затем мы начинаем с вершины **1** и следуем вдоль родительских связей, пока не столкнемся с вершиной, потенциал которой известен (в данном случае это **6**), и следуем вниз по этому пути с тем, чтобы вычислить потенциалы вершин **9** и **1** (диаграмма в центре). Далее, отправляясь от вершины **2**, мы можем рассчитать ее потенциал, взяв за основу потенциал ее родителя (диаграмма справа). Когда мы отправимся из вершины **3**, мы видим, что ее потенциал уже известен, и т.д. В рассматриваемом примере, когда мы проверяем каждую вершину после вершины **1**, мы узнаем, что ее потенциал уже подсчитан, либо мы можем вычислить его, взяв за основу потенциал ее родителя. Мы никогда не проходим по одному и тому же ребру дважды, независимо от того, какую структуру имеет дерево, поэтому суммарное время этих вычислений линейно.*

В данном случае нет необходимости рассматривать ребра в том порядке, в каком они расположены в цикле; вполне достаточно просто рассмотреть каждое из них (в любом направлении). Соответственно, вполне достаточно пройти каждый путь из этих узлов в направлении их LCA-предшественника. Чтобы нарастить поток в цикле, образованном при добавлении ребра v-w, мы проталкиваем поток из v в w, из v вдоль пути к LCA-предшественнику r и из w вдоль пути в r, но в обратном направлении для каждого ребра. Программа 22.11 реализует описанную идею в форме функции, которая наращивает поток в цикле и возвращает ребра, которые заполняются или опорожняются в процессе этого наращивания.

Программа 22.11. Наращивание потока вдоль цикла

Чтобы найти наименьшего общего предшественника двух вершин, мы, отправляясь из них вверх по дереву, синхронно помечаем узлы. LCA-предшественник есть корень в тех случаях, когда он является единственным обнаруженным помеченным узлом. В противном случае LCA-предшественник есть первый обнаруженный помеченный узел, не являющийся корнем. Чтобы нарастить поток, мы используем функцию, аналогичную используемой в программе 22.3; она сохраняет пути в векторе **st** и возвращает ребро, которое было опорожнено или заполнено в процессе наращивания (*см. рассуждения по тексту*).

```
int lca(int v, int w)
  { mark[v] = ++valid; mark[w] = valid;
    while (v != w)
      {
       if (v != t) v = ST(v);
       if (v != t && mark[v] == valid) return v;
       mark[v] = valid;
       if (w != t) w = ST(w);
       if (w != t && mark[w] == valid) return w;
       mark[w] = valid;
      }
    return v;
  }
Edge *augment(Edge *x)
  { int v = x->v(), w = x->w(); int r = lca(v, w);
    int d = x->capRto(w);
    for (int u = w; u != r; u = ST(u))
      if (st[u]->capRto(ST(u)) < d)
        d = st[u]->capRto(ST(u));
    for (int u = v; u != r; u = ST(u))
      if (st[u]->capRto(u) < d)
        d = st[u]->capRto(u);
    x->addflowRto(w, d); Edge* e = x;
    for (int u = w; u != r; u = ST(u))
      { st[u]->addflowRto(ST(u), d);
        if (st[u]->capRto(ST(u)) == 0) e = st[u]; }
    for (int u = v; u != r; u = ST(u))
      { st[u]->addflowRto(u, d);
        if (st[u]->capRto(u) == 0) e = st[u]; }
    return e;
  }
```

Реализация в программе 22.11 использует простую технологию, чтобы избежать затрат на инициализацию всех меток, которую нужно выполнять каждый раз при вызове программы. Мы храним эти метки как глобальные переменные, инициализированные нуля-

ми. Каждый раз, когда мы ищем LCA-предшественника, мы увеличиваем значение глобального счетчика и помечаем вершины, назначая соответствующим им элементам вектора, индексированного именами вершин, текущее значение этого счетчика. После инициализации этот метод дает возможность выполнять вычисления за время, пропорциональное длине цикла. В условиях типовых задач мы можем проводить наращивание потока во многих циклах небольших размеров и достичь заметной экономии времени. Как мы убедимся далее, этот метод экономии времени можно с успехом применять и в других частях реализации.

Наша третья задача, требующая манипулирования элементами дерева, есть подстановка вместо ребра **u-v** другого ребра в цикле, который это другое ребро образует с древесными ребрами. В программе 22.12 реализована функция, выполняющая эту задачу на дереве, представленном в виде родительских связей. И снова LCA-предшественник вершин **u** и **v** играет важную роль, поскольку удаляемое ребро лежит либо на пути из **u** в LCA, либо на пути из **v** в LCA. Удаление ребра приводит к отсоединению от дерева всех его потомков, однако мы можем скомпенсировать нанесенный ущерб, меняя направление связи между ребром **u-v** и удаленным ребром на обратные, как показано на рис. 22.48.

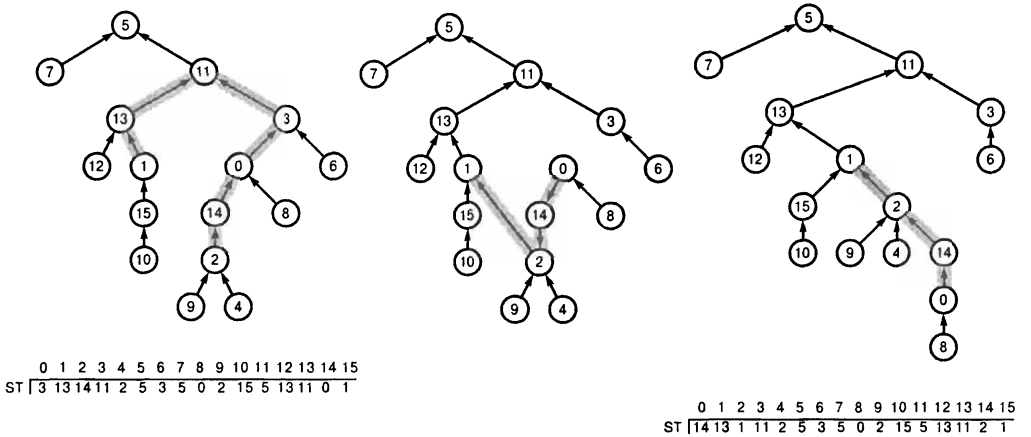


РИСУНОК 22.48. ПОДСТАНОВКА ОСТОВНОГО ДЕРЕВО

Этот пример служит иллюстрацией базовой операции манипулирования деревом в сетевом симплексном алгоритме для случая представления графа в виде родительских связей. На диаграмме слева изображено демонстрационное дерево, в котором все связи направлены вверх, как показывает структура ***ST*** *родительских связей. (В рассматриваемой нами программе функция* ***ST*** *вычисляет родителя заданной вершины по указателю, хранящемуся в векторе* ***st****, индексированном вершинами.) Добавление ребра 1-2 приводит к образованию цикла, в который входят пути из вершин* ***1*** *и* ***2****, ведущие к из LCA-предшественнику, в роли которого выступает вершина* ***11****. Если мы удалим одно из этих ребер, скажем,* ***0-3****, структура сохранит древовидный характер. Чтобы внести изменения в массив родительских связей, отражающие эти изменения, мы меняем направления всех связей из* ***2*** *в* ***3*** *(диаграмма в центре). Дерево справа есть то же дерево, в котором позиции узлов изменены таким образом, что все связи направлены вверх в соответствии со значениями массива родительских связей, представляющего рассматриваемое дерево (диаграмма справа).*

Эти три реализации поддерживают базовые операции, которые служат основой сетевого симплексного алгоритма: можно выбирать подходящее ребро, изучая приведенные стоимости и потоки; можно воспользоваться представлением остовного дерева в виде родительских связей для наращивания потока в отрицательном цикле, образованном древесными деревьями и выбранным подходящим ребром; и можно обновлять дерево и выполнять пересчет потенциалов. Эти операции показаны на примере транспортной сети на рис. 22.49 и 22.50.

Программа 22.12. Подстановка остовного дерева

Функция **update** добавляет ребро в остовное дерево и удаляет из образовавшегося при этом цикла некоторое ребро. Удаляемое ребро лежит на пути из одной из двух вершин, соединенных добавленным ребром, к их LCA-предшественнику. Эта реализация использует функцию **onpath**, осуществляющую выбор удаляемого ребра, и функцию **reverse**, обеспечивающую изменение направления ребра на обратное на пути, соединяющем ее с добавляемым ребром.

```
bool onpath(int a, int b, int c)
  {
    for (int i = a; i != c; i = ST(i))
      if (i == b) return true;
    return false;
  }
void reverse(int u, int x)
  { Edge *e = st[u];
    for (int i = ST(u); i != x; i = ST(i))
      { Edge *y = st[i]; st[i] = e; e = y; }
  }
void update(Edge *w, Edge *y)
  { int u = y->w(), v = y->v(), x = w->w();
    if (st[x] != w) x = w->v();
    int r = lca(u, v);
    if (onpath(u, x, r))
      { reverse(u, x); st[u] = y; return; }
    if (onpath(v, x, r))
      { reverse(v, x); st[v] = y; return; }
  }
```

На рис. 22.49 показана инициализация структур данных, использующих фиктивное ребро с максимальным потоком в нем, как и на рис. 22.42. На нем показано исходное допустимое остовное дерево, представленное в виде родительских связей, соответствующие потенциалы вершин, приведенные стоимости недревесных ребер и исходное множество подходящих ребер. Наряду с этим, вместо того чтобы вычислять величину максимального потока в реализации, мы используем величину истечения истока, которая гарантированно не меньше величины максимального потока; мы используем здесь величину с тем, чтобы легче проследить работу алгоритма.

Рисунок 22.50 служит иллюстрацией изменений в структурах данных для каждой последовательности подходящих ребер и наращивания на базе циклов с отрицательной стоимостью. Эта последовательность не отражает какого-либо конкретного метода выбора подходящих ребер; она представляет выборы, которые делают аугментальные пути такими, какими они показаны на рис. 22.42. На этих рисунках изображены все потенциалы вершин и все приведенные стоимости по завершении каждого наращивания цикла, даже

если многие из этих числовых значений определены неявно и не обязательно вычислены явно типовыми реализациями. Назначение этих рисунков состоит в том, чтобы служить иллюстрацией общего продвижения алгоритма и состояния структур данных по мере перехода алгоритма от одного допустимого остовного дерева к другому за счет простого добавления подходящего ребра и удаления некоторого древесного ребра из образовавшегося цикла.

Один из критических фактов, иллюстрацией которого служит пример, представленный на рис. 22.50, заключается в том, что алгоритм может не остановиться, поскольку порожние или заполненные ребра остовного дерева могут воспрепятствовать проталкиванию потока вдоль отрицательного цикла, который мы выявляем. Таким образом, мы можем выявить походящее ребро и отрицательный цикл, который оно образует с ребрами остовного дерева, тем не менее, максимальная величина потока, которую мы можем протолкнуть через этот цикл, может оказаться равной 0. И в этом случае мы выполняем подстановку подходящего ребра вместо одного из ребер цикла, однако нам не удастся снизить стоимость потока. Чтобы добиться того, чтобы алгоритм самостоятельно прекращал свою работу, необходимо выполнять специальную проверку, иначе алгоритм будет выполнять бесконечную последовательность наращиваний потока на нулевую величину.

РИСУНОК 22.49. ИНИЦИАЛИЗАЦИЯ В СЕТЕВОМ СИМПЛЕКСНОМ АЛГОРИТМЕ

*Чтобы выполнить инициализацию структуры данных для сетевого симплексного алгоритма, мы начинаем с того, что всем ребрам назначаем нулевой поток (диаграмма слева), затем добавляем фиктивное ребро **0-5** из истока в сток, величина потока в котором не меньше величины максимального потока (для ясности мы воспользуемся значением, равным здесь величине максимального потока). Значение стоимости **9** фиктивного ребра больше стоимости любого цикла сети; в этой реализации мы используем значение CV. Фиктивное ребро в транспортной сети не показано, однако оно включено в остаточную сеть (диаграмма в цикле).*

*Мы инициализируем остовное дерево стоком в качестве корня и с истоком в качестве его единственного потомка, и деревом поиска графа, индуцированного остальными узлами остаточной сети. Такая реализация использует представление дерева в виде родительских связей в массиве **st** и функцию **ST**; на наших рисунках изображена эта реализация и две других: корневая реализация, показанная справа, и множество заштрихованных ребер в остаточной сети.*

*Потенциалы вершин содержатся в массиве **pt** и вычисляются с учетом структуры дерева с таким расчетом, чтобы разность потенциалов вершин конкретного древесного ребра была равна его стоимости.*

*В столбце, обозначенном как **costR** и расположенном в центре диаграммы, показаны приведенные стоимости недревесных ребер, которые вычислены для каждого ребра путем прибавления разности потенциалов вершин к их стоимости. Приведенная цена древесных ребер равна нулю и в столбце не проставляется. Порожние ребра с отрицательными приведенными стоимостями и заполненные ребра с положительными приведенными стоимостями (подходящие ребра) помечены звездочками.*

РИСУНОК 22.50. ОСТАТОЧНАЯ СЕТЬ И ОСТОВНЫЕ ДЕРЕВЬЯ (СЕТЕВОЙ СИМПЛЕКСНЫЙ АГЛОРИТМ)

Каждый ряд этого рисунка соответствует итерации сетевого симплексного алгоритма по завершении инициализации, представленной на рис. 22.49. На каждой итерации алгоритм выбирает подходящее ребро, наращивает поток в цикле и обновляет структуры данных следующим образом: сначала наращивается поток, включая соответствующие изменения в остаточной сети. Во-вторых, древовидная структура ***ST*** *подвергается изменению за счет добавления подходящего ребра и удаления соответствующего ребра из цикла, которое подходящее ребро образует с древесными ребрами. В третьих, таблица потенциалов* ***phi*** *обновляется с целью отобразить изменения в структуре дерева. В-четвертых, приведенные стоимости недревесных ребер (столбец, обозначенный меткой* ***costR*** *в центре) обновляются с целью отобразить изменения значений потенциалов, и эти значения используются для выявления порожних ребер с отрицательной приведенной стоимостью и заполненных ребер с положительной приведенной стоимостью как подходящих ребер (помечены звездочками на приведенных стоимостях). Реализации не обязательно должны выполнять все эти вычисления (они просто должны вычислить изменения потенциалов и приведенных стоимостей, этого достаточно для выявления подходящих ребер), однако мы приводим здесь все числа, чтобы дать полное представление о работе рассматриваемого алгоритма.*

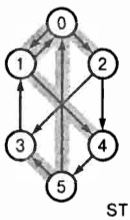
Завершающее наращивание в этом примере вырождается. Оно не увеличивают потока, но и не выявляет подходящих ребер, откуда следует, что этот поток есть максимальный поток минимальной стоимости.

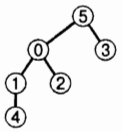
Добавить 3-5, нарастить поток на +2 on 0-1-3-5-0, удалить 1-3

	пропускная способность	стоимость	поток
0-1	3	3	**2**
0-2	3	1	**0**
1-3	2	1	**2**
1-4	2	1	**0**
2-3	1	4	**0**
2-4	2	2	**0**
3-5	2	2	**2**
4-5	2	1	**0**
0-5	6	9	**2**

суммарная стоимость 30

приведенная стоимость		
0-1		1-0
0-2		2-0
1-3	-3	3-1
1-4		4-1
2-3	-2*	3-2
2-4	-1*	4-2
3-5		5-3
4-5	-4*	5-4
5-0		5-0





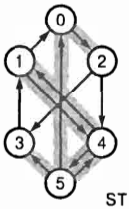
	0	1	2	3	4	5
ST	5	0	0	5	1	5
phi	0	-3	-1	-7	-4	-9

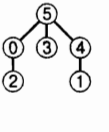
Добавить 4-5, нарастить поток на +1
в цикле 0-1-4-5-0, удалить 0-1

0-1	3	3	**3**
0-2	3	1	**0**
1-3	2	1	**2**
1-4	2	1	**1**
2-3	1	4	**0**
2-4	2	2	**0**
3-5	2	2	**2**
4-5	2	1	**1**
0-5	6	9	**1**

суммарная стоимость 26

приведенная стоимость		
0-1	-4	1-0
0-2		2-0
1-3	1*	3-1
1-4		4-1
2-3	-2*	3-2
2-4	-5*	4-2
3-5		5-3
4-5		5-4
		5-0





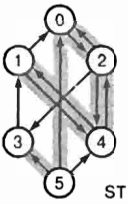
	0	1	2	3	4	5
ST	5	4	0	5	5	5
phi	0	-7	-1	-7	-8	-9

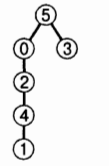
Добавить 2-4, нарастить поток на +1
в цикле 0-2-4-5-0, удалить 4-5

0-1	3	3	**3**
0-2	3	1	**1**
1-3	2	1	**2**
1-4	2	1	**1**
2-3	1	4	**0**
2-4	2	2	**1**
3-5	2	2	**2**
4-5	2	1	**2**
0-5	6	9	**0**

суммарная стоимость: 21

приведенная стоимость		
0-1	1*	1-0
2-0		2-0
1-3	-4	3-1
1-4		4-1
2-3	-2*	3-2
2-4		4-2
3-5		5-3
4-5	-5	5-4
		5-0





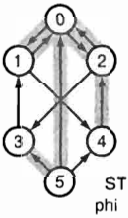
	0	1	2	3	4	5
ST	5	4	0	5	2	5
phi	3	1	0	0	-3	-4

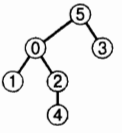
Добавить 1-0, нарастить поток на +1
в цикле 1-0-2-4-1, удалить 4-1

0-1	3	3	**2**
0-2	3	1	**2**
1-3	2	1	**2**
1-4	2	1	**0**
2-3	1	4	**0**
2-4	2	2	**2**
3-5	2	2	**2**
4-5	2	1	**2**
0-5	6	9	**0**

суммарная стоимость: 20

приведенная стоимость		
0-1		1-0
0-2		2-0
1-3	-3	3-1
1-4	1	4-1
2-3	-2*	3-2
2-4		4-2
3-5		5-3
4-5	-5	5-4
		5-0





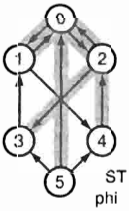
	0	1	2	3	4	5
ST	5	0	0	5	2	5
phi	0	-3	-1	-7	-3	-9

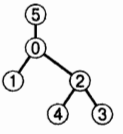
Добавить 2-3, нарастить поток на 0
в цикле 1-0-2-3-1, удалить 3-5

0-1	3	3	**2**
0-2	3	1	**2**
1-3	2	1	**2**
1-4	2	1	**0**
2-3	1	4	**0**
2-4	2	2	**2**
3-5	2	2	**2**
4-5	2	1	**2**
0-5	6	9	**0**

суммарная стоимость: 20

приведенная стоимость		
0-1		1-0
0-2		2-0
1-3	-1	3-1
1-4	1	4-1
2-3		3-2
2-4		4-2
3-5	-2	5-3
4-5	-5	5-4
		5-0





	0	1	2	3	4	5
ST	5	0	0	2	2	5
phi	0	-3	-1	-5	-3	-9

Если в аугментальном цикле содержится более одного заполненного или порожнего ребра, то алгоритм подстановки, реализованный в программе 22.12, всегда удаляет из дерева то из них, которое расположено ближе других к LCA-предшественнику двух вершин, образующих подходящее ребро. К счастью, было доказано, что такая стратегия выбора ребра для удаления из цикла обеспечивает останов алгоритма (см. *раздел ссылок*).

Завершающий выбор, с которым нам приходится сталкиваться при разработке сетевого симплексного алгоритма, касается стратегии выявления подходящих ребер и выбора одного из них для включения в дерево. Нужно ли заводить специальную структуру данных для хранения подходящих ребер? Если нужно, то какой сложностью должна обладать такая структура данных? Ответ на эти вопросы в какой-то степени зависит от приложения и динамических характеристик процесса решения задачи в конкретных случаях. Если суммарное количество подходящих ребер невелико, то целесообразно заводить специальную структуру данных, если большая часть ребер входит в число подходящих в течение продолжительного времени, то в специальной структуре нет необходимости. Поддержание отдельных структур позволяет снизить затраты ресурсов на поиск подходящих ребер, но в то же время может потребовать выполнения дорогостоящих операций по обновлению результатов вычислений. Каким критерием мы воспользуемся для выбора нужного подходящего ребра из некоторого множества таких ребер? И снова нам предоставляется богатый выбор. Мы проанализируем примеры в наших реализациях, затем рассмотрим существующие альтернативы. Например, программа 22.13 демонстрирует функцию, которая находит подходящее ребро минимальной приведенной стоимости: ни одно другое ребро не позволит построить цикл, в случае включения которого наращивание потока в цикле не приведет к уменьшению суммарной стоимости.

Программа 22.13. Поиск подходящих ребер

Данная функция находит подходящее ребро минимальной приведенной стоимости. Это простая реализация производит обход всех ребер в сети.

```
int costR(Edge *e, int v)
  { int R = e->cost() + phi[e->w()] - phi[e->v()];
    return e->from(v) ? R : -R; }
Edge  *besteligible()
{ Edge *x = 0;
  for (int v = 0, min = C*G.V(); v < G.V(); v++)
    {
      typename Graph::adjIterator A(G, v);
      for (Edge* e = A.beg(); !A.end(); e = A.nxt())
        if (e->capRto(e->other(v)) > 0)
          if (e->capRto(v) == 0)
            if (costR(e, v) < min)
              { x = e; min = costR(e, v); }
    }
  return x;
}
```

Программа 22.14 представляет собой полную реализацию сетевого симплексного алгоритма, которая использует стратегию выбора подходящего ребра, позволяющую получить отрицательный цикл, стоимость которого максимальна по абсолютному значению. Эта реализация заимствует из программ 22.10 и 22.13 функции, манипулирующие элементами деревьев и выполняющие поиск подходящих ребер, в то же время к ним примени-

мы замечания, сделанные нами в адрес реализации с вычеркиванием циклов (программа 22.9), а именно: это хорошо, что такая небольшая порция программных кодов обладает достаточной мощью, чтобы обеспечить получение полезных решений в контексте общей модели решения задач, которая позволяет решать задачи о потоках минимальной стоимости.

Таблица 22.14. Сетевой симплексный алгоритм (базовая реализация)

Данный класс использует сетевой симплексный алгоритм для решения задачи о потоке минимальной стоимости. Он использует стандартную функцию поиска в глубину **dfsR** на исходном дереве (см. упражнение 22.117), затем входит в цикл, в котором использует функции из программ 22.10–22.13 для вычисления потенциалов всех вершин, осуществляет проверку всех ребер с целью нахождения такого из них, которое образует отрицательный цикл минимальной стоимости, и производит наращивание потока в этом цикле.

```
template <class Graph, class Edge> class MINCOST
{ const Graph &G; int s, t; int valid;
   vector<Edge *> st; vector<int> mark, phi;
   void dfsR(Edge);
   int ST(int);
   int phiR(int);
   int lca(int, int); Edge *augment(Edge *);
   bool onpath(int, int, int);
   void reverse(int, int);
   void update(Edge *, Edge *);
   int costR(Edge *, int); Edge *besteligible();
public:
   MINCOST(Graph &G, int s, int t) : G(G), s(s), t(t)
     st(G.V()), mark(G.V(), -1), phi(G.V())
   {
     Edge *z = new EDGE(s, t, M*G.V(), C*G.V());
     G.insert(z);
     z->addflowto(t, z->cap());
     dfsR(z);
     for (valid = 1; ; valid++ )
      {
        phi[t] = z->costRto(s); mark[t] = valid;
        for (int v = 0; v < G.V(); v++)
          if (v != t) phi[v] = phiR(v);
        Edge *x = besteligible();
        if (costR(x, x->v()) == 0) break;
        update(augment(x), x);
      }
   G.remove(z); delete z;
}
```

Производительность в худшем случае, характерном для программы 22.14, по меньшей мере, в V раз меньше, чем аналогичный параметр для реализации с вычеркиванием циклов, представленной программой 22.9, поскольку затраты времени на один цикл составляют E (чтобы найти подходящее ребро), а не VE (чтобы отыскать отрицательный цикл). И хотя есть подозрение, что применение максимального наращивания приведет к меньшему числу операций наращивания, чем для случая, когда берется первый попавшийся отрицательный цикл, как это имеет место в условиях алгоритма Беллмана-Форда, тем не менее, как удалось показать, это подозрение лишено оснований. Конкретные ограниче-

ния на число аугментальных циклов трудно реализовать, и как обычно, эти границы намного выше, чем числа, с которыми мы сталкиваемся на практике. Как уже говорилось выше, получены теоретические результаты, показывающие, что некоторые стратегии гарантируют, что число аугментальных циклов ограничено полиномом от числа ребер, в то же время в практических реализациях обычно принимается экспоненциальный худший случай.

В свете всех этих соображений существует множество вариантов улучшения производительности рассматриваемых алгоритмов. Например, программа 22.15 является еще одной реализацией сетевого симплексного алгоритма. Прямолинейная реализация в программе 22.14 всегда требует времени, пропорционального V, на обновление потенциалов дерева, и всегда затрачивает время, пропорциональное E, на обнаружение подходящего ребра с максимальной приведенной стоимостью. Реализация программы 22.15 имеет целью ликвидировать упомянутые выше затраты в типовых сетях.

Программа 22.15. Сетевой симплексный алгоритм (усовершенствованная реализация)

Замена ссылок на **phi** обращениями к **phiR** в функции **R** и замена цикла **for** в конструкторе программы 22.14 данным программным кодом позволяет получить реализацию сетевого симплексного алгоритма, которая обеспечивает экономию времени на каждой итерации за счет того, что потенциалы вычисляются, когда это необходимо, и за счет выбора первого обнаруженного ею подходящего ребра.

```
int old = 0;
for (valid = 1; valid != old; )
  {
    old = valid;
    for (int v = 0; v < G.V(); v++)
    {
      typename Graph::adjIterator A(G, v);
      for (Edge* e = A.beg(); !A.end(); e = A.nxt())
        if (e->capRto(e->other(v)) > 0)
          if (e->capRto(v) == 0)
            { update(augment(e), e); valid++; }
    }
  }
```

Во-первых, даже если выбор максимального ребра приводит к меньшему числу итераций, затраты на проверку каждого ребра с целью найти максимальное ребро, могут оказаться нецелесообразными. Мы бы могли выполнить многочисленные операции наращивания потоков в коротких циклах за время, затрачиваемое на просмотр всех ребер. Соответственно, имеет смысл рассмотреть стратегию использования *произвольного* подходящего ребра, а не тратить время на поиск какого-то конкретного подходящего ребра. В худшем случае не пришлось бы проверять все ребра или их большую часть, чтобы отыскать подходящее ребро, но обычно мы рассчитываем на то, что придется проверять сравнительно небольшое количество ребер, чтобы отыскать подходящее ребро. Один из подходов заключается в том, чтобы каждый раз начинать с начала; другой подход предусматривает случайный выбор исходной точки (см. упражнение 22.126). Такое использование случайного фактора делает маловероятным появление искусственных длинных последовательностей аугментальных путей.

Во-вторых, для вычисления потенциалов мы принимаем "*отложенный*" подход. Вместо того, чтобы вычислять все потенциалы в векторе **phi**, индексированном именами вер-

шин, а потом обращаться к ним по мере необходимости, мы вызываем функцию **phiR**, чтобы получить каждое значение потенциала; она проходит вверх по дереву с тем, чтобы найти допустимый потенциал, а затем вычисляет все необходимые потенциалы на этом пути. Чтобы реализовать такой подход, мы просто заменяем доступ к массиву **phi[u]** на функцию, которая определяет стоимость использования вызова функции **phiR(u)**. В худшем случае мы вычисляем все потенциалы так же, как и раньше, однако если мы исследуем только несколько подходящих ребер, то мы вычисляем только те потенциалы, которые необходимы для выявления этих ребер.

Подобные изменения никак не отражаются на производительности алгоритма в худшем случае, но они несомненно повышают его быстродействие в практических приложениях. Несколько других соображений по повышению производительности сетевого симплексного алгоритма исследуются в упражнениях (см. упражнения 22.126–22.130), и они представляют только незначительную часть того, что было предложено.

Как мы неоднократно подчеркивали на протяжении всей этой книги, задача анализа и сравнения алгоритмов на графа сложна сама по себе. С появлением сетевого симплексного алгоритма эта задача еще больше усложняется за счет различных подходов к реализации и широкому разнообразию приложений, с которыми нам приходится сталкиваться (см. раздел 22.5). Какая из реализаций наилучшая? Имеем ли мы право сравнивать реализации, в основу которых положены доказуемые границы для худших случаев? Насколько точно мы можем выразить в конкретных числах различие в производительности различных реализаций применительно к конкретным приложениям? Должны ли мы использовать различные реализации, приспособленные под конкретное приложение?

Мы рекомендуем читателям набраться большего опыта решения вычислительных задач с использованием различных реализаций сетевого симплексного алгоритма и найти ответ на некоторые из предложенных вопросов, проведя эмпирические исследования, подобные тем, которые мы неоднократно и настоятельно рекомендовали на протяжении всей книги. В стремлении отыскать решение задач о потоке минимальной стоимости, мы сталкиваемся со знакомыми труднорешаемыми проблемами, тем не менее, опыт, который мы приобрели при решении задач с возрастающей трудностью на протяжении этой книги, формирует у вас солидный фундамент для разработки эффективных реализаций, которые способны эффективно решать широкие классы важных практических задач. Некоторые из этих исследований описаны в упражнениях в конце этого и следующего разделов, однако, эти упражнения следует рассматривать только в качестве отправной точки. Каждый читатель может провести новые эмпирические исследования, которые способны пролить свет на те или иные реализации и приложения, представляющие интерес в научном плане.

Возможность кардинально повысить производительность приложений, которые остро в этом нуждаются, за счет правильного развертывания классических структур данных и алгоритмов (либо разработки новых) для базовых задач, делает изучение реализаций сетевого симплексного алгоритма благодатной областью исследований, к тому же существует обширная литература по реализациям этого алгоритма. В прошлом прогресс в этой области имел решающее значение, ибо он помог уменьшить огромные затраты на решение сетевых симплексных задач. Исследователи предпочитают полагаться на тщательно подобранные библиотеки в своем стремлении решить эти задачи, что, собственно, и сегодня оправдывает себя во многих случаях. Тем не менее, таким библиотекам все труд-

нее шагать в ногу с современными исследованиями и приспосабливаться к новым задачам, которые возникают в новых приложениях. Быстродействие и компактные размеры современных компьютеров, доступные реализации, подобные программам 22.12 и 22.13, могут послужить отправными точками для разработки эффективных инструментальных средств решения задач для многих приложений.

Упражнения

▷ **22.116**. Вычислите максимальный поток с ассоциированным с ним допустимым остовным деревом для транспортной сети, показанной на рис. 22.10.

▷ **22.117**. Реализуйте функцию **dfsR** для программы 22.14.

○ **22.118**. Реализуйте функцию, которая удаляет циклы из частично заполненных деревьев из заданного сетевого потока и строит допустимое остовное дерево для итогового потока, как показано на рис. 22.44. Упакуйте созданную функцию таким образом, чтобы ее можно было использовать для построения исходного дерева в программе 22.14 или в программе 22.15.

22.119. В примере, представленном на рис. 22.46, покажите, как отразится на таблицах потенциалов смена на обратную ориентации ребра, соединяющего вершины **6** и **5**.

○ **22.120**. Постройте транспортную сеть и покажите такую последовательность аугментальных ребер, что общий сетевой симплексный алгоритм не останавливается.

▷ **22.121**. Покажите в стиле рис. 22.47 процесс вычисления потенциалов дерева с корнем в вершине 0, которое показано на рис. 22.46.

22.122. Покажите в стиле рис. 22.50 процесс вычисления максимального потока минимальной стоимости в транспортной сети, изображенной на рис. 22.10, отправляясь от базового максимального потока и связанного с ним базового остовного дерева, о котором шла речь в упражнении 22.116.

○ **22.123**. Предположим, что все недревесные ребра пусты. Напишите функцию, которая вычисляет потоки в древесных ребрах, помещая поток в ребре, соединяющем вершину **v** с ее родителем в дереве, в **v**-й элемент вектора **flow**.

○ **22.124**. Выполните упражнение 22.123 для случая, когда некоторые недревесные ребра могут оказаться заполненными.

22.125. Воспользуйтесь программой 22.12 в качестве основы для построения алгоритма MST (алгоритм вычисления минимального остовного дерева). Проведите эмпирические исследования, сравнивая полученную вами реализацию с тремя базовыми алгоритмами вычисления MST, описанными в главе 20 (см. упражнение 20.66).

22.126. Опишите, какие изменения потребуется внести в программу 22.15, чтобы она каждый раз начинала поиск подходящего ребра со случайно выбранного ребра, а не с самого начала.

22.127. Модифицируйте полученное вами решение в упражнении 22.126 таким образом, чтобы каждый раз, когда программа выполняет поиск подходящего ребра, она начинает его с того места, в котором завершился предыдущий поиск.

22.128. Внесите в приватные функции-элементы из данного раздела такие изменения, которые позволили бы поддерживать трехсвязные древовидные структуры, включаю-

щие родителя каждого узла, крайнего слева потомка и правого брата (см. раздел 5.4). Используемые вами функции, обеспечивающие наращивание потока в циклах и подстановку подходящего ребра вместо древесного ребра, должны выполняться за время, пропорциональное продолжительности цикла наращивания потока, а функции, вычисляющие потенциалы, должны выполняться за время, пропорциональное размеру меньшего из двух поддеревьев, которые образуются в результате удаления древесного ребра.

○ **22.129**. Внесите в приватные функции-элементы из данного раздела такие изменения, которые, в дополнение к базовому вектору дерева родительских связей, позволили бы поддерживать два других вектора, индексированные именами вершин: один из них содержит расстояние каждой вершины до корня, а другой вектор — потомка каждой вершины при поиске в глубину (DFS). Используемые вами функции, обеспечивающие наращивание потока в циклах и подстановку подходящего ребра вместо древесного ребра, должны выполняться за время, пропорциональное продолжительности цикла наращивания потока, а функции, вычисляющие потенциалы, должны выполняться за время, пропорциональное размеру меньшего из двух поддеревьев, которые образуются в результате удаления древесного ребра.

● **22.130**. Проведите исследование идеи использования обобщенной очереди подходящих ребер. Рассмотрите различные реализации такой обобщенной очереди и различные ее усовершенствования, позволяющие избежать чрезмерных вычислений стоимостей ребер, например, уделять меньше внимания подмножествам подходящих ребер за счет ограничения размеров очереди или, возможно, позволить некоторым неподходящим ребрам оставаться в очереди.

● **22.131**. Выполните эмпирические исследования с целью определить число итераций, число вычислений потенциалов вершин и отношение времени прогона к E для нескольких версий сетевого симплексного алгоритма на различных видах сетей (см. упражнение 22.7–22.12). Рассмотрите различные алгоритмы, описанные в тексте и предыдущих упражнениях, и сосредоточьте свое внимание на тех из них, которые показывают себя с лучшей стороны на крупных разреженных графах.

● **22.132**. Напишите клиентскую программу, которая выполняет графическую анимацию динамики сетевых симплексных алгоритмов. Ваша программа должна создавать изображения, подобные показанным на рис. 22.50 и других рисунках этого раздела (рис. 22.48). Протестируйте полученную реализацию на эвклидовых сетях, описанных в упражнениях 22.7–22.12.

22.7. Сведение к задаче о потоке минимальной стоимости

Поток минимальной стоимости представляет собой общую модель, которая может охватывать множество полезных практических задач. В данном разделе мы дадим обоснование этого утверждения, доказав возможность сведения самых разнообразных задач к задаче о потоке минимальной стоимости.

Вполне очевидно, что задача о потоке минимальной стоимости носит более общий характер, нежели задача о максимальном потоке. В частности, если мы назначим в построении, показанном на рис. 22.42, фиктивному ребру стоимость, равную единице, а другим ребрам стоимость, равную нулю, то любой максимальный поток минимальной стоимос-

ти минимизирует поток в фиктивном ребре и тем самым увеличивает поток до максимального в исходной сети. В силу этого обстоятельства, все задачи, рассмотренные в разделе 22.4, которые сводятся к задаче о максимальном потоке, сводятся также и к задаче о потоке минимальной стоимости. Это множество задач, помимо прочих, включает задачи о двудольном сочетании, о допустимых потоках и о минимальном сечении.

Однако больший интерес представляет тот факт, что мы можем проверить возможность использования свойств полученных нами алгоритмов решения задачи о потоке минимальной стоимости для разработки новых общих алгоритмов решения задачи о максимальном потоке. Мы уже отмечали, что общий алгоритм вычеркивания циклов, применяемый при решении задачи о максимальной потоке минимальной стоимости, позволяет получить общий алгоритм вычисления аугментальных путей для задачи о максимальном потоке. В частности, такой подход приводит к реализации, которая находит аугментальные пути без поиска в сети (см. упражнения 22.133 и 22.134). С другой стороны, этот алгоритм может строить аугментальные пути с нулевым потоком, в связи с чем его рабочие характеристики трудно оценить (*см. раздел ссылок*).

Задача о максимальном потоке минимальной стоимости носит более общий характер, нежели задача поиска кратчайшего пути. Это утверждение следует из возможности ее сведения к другой задаче.

Свойство 22.29. *Задача о кратчайшем пути с единственным источником (в сетях, в которых нет отрицательных циклов) сводится к задаче о допустимых потоках минимальной стоимости.*

Доказательство: Пусть дана задача поиска кратчайшего пути с единственным источником (сеть и исток в вершине **s**), требуется построить транспортную сеть с теми же вершинами, ребрами и стоимостями ребер и наделить ребра неограниченной пропускной способностью. Добавьте новую вершину истока с ребром, ведущим в **s** и имеющим стоимость ноль и пропускную способность $V-1$, а также новую вершину стока с ребрами, ведущими из всех других вершин и имеющими нулевые стоимости и пропускные способности, равные 1. Это построение представлено на рис. 22.51.

Решим задачу о допустимом потоке минимальной стоимости на этой сети. При необходимости удалим из решения все циклы с тем, чтобы получить решение с остовным деревом. Это остовное дерево непосредственно соответствует остовному дереву кратчайших путей в исходной сети. Подробное доказательство этого факта оставляем читателям на самостоятельную проработку (см. упражнение 22.128). ■

Таким образом, все задачи, рассмотренные в разделе 21.6, которые сводятся к задаче поиска кратчайших путей в сети с единственным источником, также сводятся к задаче о потоке минимальной стоимости. Это множество задач включает задачу календарного планирования работ с конечными сроками завершения и с разностными и прочими ограничениями.

Как мы убедились при изучении задач о максимальном потоке, имеет смысл проанализировать во всех подробностях работу сетевого симплексного алгоритма при решении задачи поиска кратчайших путей с использованием приведения, описанного в свойстве 22.29. В этом случае данный алгоритм поддерживает остовное дерево с корнем в истоке, во многом напоминая алгоритмы на базе поиска, которые рассматривались главе 21, однако потенциалы и приведенные стоимости дают дополнительную гибкость при разработке методов выбора следующего ребра для включения в дерево.

В общем случае мы не используем тот факт, что задача о потоке минимальной стоимости есть правильное обобщение задачи о минимальном потоке и задачи поиска кратчайших путей, поскольку в нашем распоряжении имеются специальные алгоритмы, гарантирующие более высокую производительность при решении обеих этих задач. Однако если таких реализаций нет, то качественная реализация сетевого симплексного алгоритма, возможно, обеспечит быстрое решение конкретных примеров обеих задач. Разумеется, мы должны избегать циклов сведения одних задач к другим или построения задач обработки сетей, которые используют такие сведения. Например, реализация алгоритма вычеркивания циклов из программы 22.9 использует как алгоритмы вычисления максимального потока, так и алгоритмы поиска кратчайших путей для решения задачи о потоке с минимальной стоимостью (см. упражнение 21.96).

Далее мы проведем анализ нескольких сетевых моделей. Сначала мы покажем, что предположение о том, что все стоимости неотрицательны, не является ограничивающим, так как мы можем преобразовать сети с неотрицательными стоимостями в сеть без таковых.

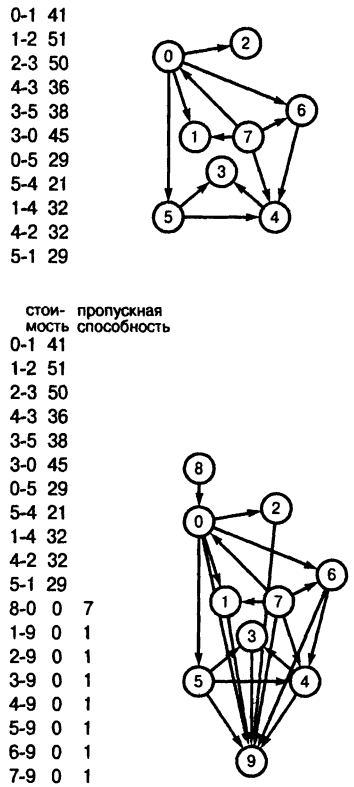


РИСУНОК 22.51. СВЕДЕНИЕ ЗАДАЧИ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ К ДРУГОЙ ЗАДАЧЕ

Поиск дерева кратчайших путей с единственным источником в сети, изображенной в верхней части рисунка, эквивалентно решению задачи о максимальном потоке минимальной стоимости в транспортной сети, показанной в нижней части рисунка.

Свойство 22.30. *В задаче о потоке с минимальной стоимостью мы можем предположить, что стоимость ребер неотрицательна, без потери общности.*

Доказательство: Мы докажем этот факт для допустимых потоков минимальной стоимости в распределительных сетях. Это результат верен и для максимальных потоков с минимальной стоимостью в силу эквивалентности этих двух задач по свойству 22.22 (см. упражнения 22.143 и 22.144).

Пусть дана распределительная сеть, заменим любое ребро **u-v**, имеющее стоимость $x < 0$ и пропускную способность c, на ребро **v-u** с той же пропускной способностью, но имеющее стоимость $-x$ (положительное число). Далее, мы можем уменьшить значение запаса-спроса вершины **u** на c и увеличить значение запаса-спроса вершины **v** на c. Такая операция соответствует проталкиванию c единиц потока из **u** в **v** с соответствующим уточнением сети.

Если в случае ребер с отрицательной стоимостью решение задачи о потоке минимальной стоимости для преобразованной сети помещает поток f в ребро **v-u**, то мы помещаем поток $c - f$ в ребро **u-v** исходной сети; в случае ребер с положительной стоимостью в преобразованной сети назначаются те же потоки, что и в исходной сети. Такое распределение потоков сохраняет ограничения на запас и спрос во всех вершинах.

Поток в ребре **u-v** преобразованной сети привносит в стоимость составляющую fx, а поток в ребре **v-u** исходной сети – составляющую $-cx + fx$. Первый член этого выражения не зависит от потока, следовательно, стоимость любого потока в преобразованной сети равна стоимости соответствующего потока в исходной сети плюс сумма произведений пропускных способностей на стоимости всех ребер с отрицательными стоимостями (они, естественно, представлены отрицательными числами). Поэтому любые потоки минимальной стоимости в преобразованной сети являются потоками минимальной стоимости в исходной сети. ■

Это сведение одной задачи к другой показывает, что мы можем сосредоточить свое внимание только на положительных стоимостях, однако в общем случае на практике мы не удосуживаемся делать это, поскольку полученные в разделах 22.5 и 22.6 реализации имеют дело исключительно с остаточными сетями и без труда выполняют обработку отрицательных стоимостей. Важно иметь в своем распоряжении *некоторую* нижнюю границу стоимостей в ряде контекстов, но эта граница не должна быть нулевой (см. упражнение 22.145).

Далее мы покажем, что, как это делалось в случае задачи о максимальном потоке, при желании можно было бы ограничиться рассмотрением ациклических сетей. Более того, можно было бы также предположить, что ребра не имеют ограничений на пропускную способность (не существует верхней границы величину потока в таких ребрах). Комбинация этих двух вариантов приводит к следующим классическим формулировкам задачи о потоках минимальной стоимости.

Транспортная задача. Решим задачу о потоке минимальной стоимости для двудольной распределительной сети, все ребра которой направлены из вершины с запасом в вершины со спросом и обладают неограниченной пропускной способностью. Как уже было показано в начале данной главы (см. рис. 22.2), обычно эта задача моделирует распределение товаров из складов (вершины с запасом) в розничные торговые точки (вершины со спросом) по каналам распределения (ребра) с конкретной стоимостью поставки единицы товара.

Свойство 22.31. *Транспортная задача эквивалентна задаче о потоке с минимальной стоимостью.*

Доказательство: Пусть задана транспортная задача, мы можем решить ее, присваивая каждому ребру пропускную способность, более высокую, чем значения запаса или спроса вершин, которые оно соединяет, и решая возникающую при этом задачу нахождения допустимого ребра минимальной стоимости на построенной в процессе решения этой задачи распределительной сети. Поэтому нам всего лишь достаточно установить, что решение стандартной задачи сводится к решению транспортной задачи.

Для разнообразия, опишем новое преобразование, которое линейно только на разреженных сетях. Построение, подобное тому, что мы использовали при доказательстве свойства 22.16, позволяет установить этот результат для неразреженных сетей (см. упражнение 22.148).

Пусть дана стандартная распределительная сеть с V вершинами и E ребрами, построим для транспортной задачи двудольную сеть с V вершинами запаса, E вершинами спроса и $2E$ ребрами следующим образом. Для каждой вершины исходной сети включим в двудольную сеть вершину с величиной запаса или спроса, равной соответству-

ющему исходному значению плюс сумма пропускных способностей исходящих ребер. Для каждого ребра **u-v** исходной сети с пропускной способностью **c** в двудольную сеть включим вершину с величиной запаса или спроса **-c** (для ссылок на такую вершину мы воспользуемся обозначением **[u-v]**). Для каждого ребра **u-v** в исходной сети включим в двудольную сеть два ребра: одно из них обладает той же стоимостью и ведет из **u** в **[u-v]**, другое, имея стоимость 0, ведет из **v** в **[u-v]**.

Следующее соответствие один к одному сохраняет стоимости потоков в обеих сетях неизменными: ребро **u-v** исходной сети пропускает поток величины **f** тогда и только тогда, когда в ребре **u-[u-v]** двудольной сети протекает поток величины **c-f** (эти два потока должны давать в сумме значение **c** в силу ограничения на запас-спрос, действующего в вершине **[u-v]**). Следовательно, любой поток минимальной стоимости в одной сети соответствует потоку минимальной стоимости в другой сети. ■

Поскольку мы не рассматривали прямых алгоритмов решения транспортной задачи, рассмотренное выше сведение одной задачи к другой представляет собой разве что академический интерес. Чтобы воспользоваться этим приемом, мы снова должны привести полученную задачу к (другой) задаче вычисления потока минимальной стоимости, воспользовавшись простым сведением, упоминавшимся при доказательстве свойства 22.31. Возможно, что такие сети допускают на практике более эффективные решения, а возможно и нет. Основная цель изучения эквивалентности транспортной задачи задаче отыскания потока минимальной стоимости заключается в том, чтобы понять, позволит ли отказ от пропускных способностей и проведение анализа на двудольных сетях существенно упростить решение задачи вычисления потока минимальной стоимости; тем не менее, это не так.

В этом контексте нам следует рассмотреть другую классическую задачу. Она обобщает задачу о двудольном сочетании, которая подробно изучалась в разделе 22.4. Подобно упомянутой задаче, простота данной задачи также обманчива.

Задача распределения. Для заданного взвешенного двудольного графа найти множество ребер минимальной стоимости, такое, что каждая вершина соединена в точности с единственной другой вершиной.

Например, мы можем обобщить рассматривавшуюся нами ранее задачу о трудоустройстве, чтобы она включала способы количественной оценки желания компании заполучить каждого претендента на рабочее место (скажем, присваивая каждому претенденту целочисленные баллы, при этом наиболее подходящему претенденту присваиваются более низкие баллы) и количественной оценки желания претендента получить рабочее место в той или иной компании. Таким образом, решение задачи распределения способно предложить метод, учитывающий предпочтения обеих сторон.

Свойство 22.32. *Задача распределения сводится к задаче вычисления потока минимальной стоимости.*

Доказательство: Этот результат может быть установлен простым сведением к транспортной задаче. Пусть дана задача распределения, построить транспортную задачу с теми же вершинами и ребрами, при этом все вершины, входящие в одно из множеств, обозначаются как вершины с запасом и им присваивается значение 1, а все вершины, входящие в другое множество, обозначаются как вершины со спросом и им также присваивается значение 1. Каждому ребру необходимо назначить пропускную способность, равную 1, и стоимость, соответствующую весу этого ребра в задаче распределения.

Любое решение этого варианта транспортной задачи есть просто множество ребер с минимальной суммарной стоимостью, при этом каждое ребро соединяет вершину с запасом с вершиной со спросом, благодаря чему непосредственно соответствует решению исходной задачи распределения.

Сведение этого варианта транспортной задачи к задаче вычисления минимального потока позволяет получить построение, по существу эквивалентное построению, которое мы использовали для сведения задачи двудольного сочетания к задаче вычисления максимального потока (см. упражнение 22.158). ■

Это отношение нельзя считать эквивалентностью, поскольку пока еще не существует известного способа сведения общей задачи о потоке минимальной стоимости к задаче распределения. В самом деле, подобно задаче поиска кратчайшего пути с единственным источником и задаче вычисления максимального потока, задача распределения на первый взгляд кажется более легкой для решения, чем задача вычисления потока минимальной стоимости, на том основании, что, как известно, алгоритмы ее решения обеспечивают более высокую асимптотическую производительность, чем широко известные алгоритмы решения задачи вычисления потока минимальной стоимости. Тем не менее, сетевой симплексный алгоритма настолько хорошо отлажен, что при его удачной программной реализации он вполне подходит для решения задачи распределения. Более того, как и в задачах поиска максимального потока и кратчайших путей, есть возможность настроить сетевой симплексный алгоритм, чтобы он обеспечивал повышенную производительность на задаче распределения (*см. раздел ссылок*).

Следующее сведение к задаче вычисления потока минимальной стоимости возвращает нас к базовой задаче поиска путей в графах подобных тем, что впервые рассматривались в разделе 17.7. Как и в случае задачи вычисления эйлерова пути, нам нужен путь, который включает все ребра графа. Признав, что не все графы содержат такой путь, мы можем ослабить ограничение, требующее, чтобы каждое ребро встречалось на пути только один раз.

Задача о почтальоне. Пусть дана некоторая сеть (взвешенный орграф), найти циклический путь минимального веса, включающий каждое ребро, по меньшей мере, один раз (см. рис. 22.52). Напомним, что основное определение, данное в главе 17, проводит различие между циклическими путями (которые могут неоднократно посещать те или иные вершины и ребра) и циклами (состоящими из различных вершин, за исключением первой и последней, которые совпадают).

Решение этой задачи может также описывать наилучший маршрут почтальона (который должен пройти все улицы на своем пути). Решение этой задачи может также дать описание маршрута снегоочистителя во время снежной бури; существует множество других аналогичных применений.

Задача о почтальоне представляет собой задачу поиска эйлерова пути, которую мы рассматривали в разделе 17.7: решение упражнения 17.92 представляет собой простую проверку

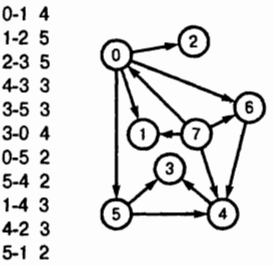


РИСУНОК 22.52. ЗАДАЧА О ПОЧТАЛЬОНЕ

Построение кратчайшего пути, который включает каждое ребро по меньшей мере один раз, представляет собой трудную задачу даже для такой простой сети, как эта, однако данная задача может быть эффективно решена за счет сведения к задаче о потоке минимальной стоимости.

на существование в орграфе эйлерова пути, а программа 17.14 является эффективным способом найти эйлеров путь в таких орграфах. Этот путь решает задачу о почтальоне, поскольку он включает каждое ребро в точности один раз – ни один другой путь не может обладать более низким весом. Задача усложняется, когда полустепени захода и полустепени исхода не обязательно равны. В общем случае, некоторые ребра приходится проходить более одного раза: задача заключается в том, чтобы минимизировать суммарный вес многократно посещаемых ребер.

Свойство 22.33. *Задача о почтальоне сводится к задаче вычисления потока с минимальной стоимостью.*

Доказательство: Дан вариант задачи о почтальоне (взвешенный орграф), определить распределительную сеть с теми же вершинами и ребрами, причем все значения запаса и спроса в вершине равны 0, стоимость ребра устанавливается равной весу соответствующего ребра, на пропускную способность ребер не накладываются никакие ограничения сверху, в то же время все пропускные способности ребер должны быть больше 1. Мы рассматриваем величину **f** потока в ребре **u-v** следующим образом: почтальон должен пройти по ребру **u-v** в сумме **f** раз.

Найдите поток минимальной стоимости для этой сети, воспользовавшись преобразованием, описанным в упражнении 22.146, с целью снятия ограничений нижней границы на пропускную способность ребер. Теорема о декомпозиции потоков утверждает, что мы можем выразить поток как множество циклов, следовательно, мы можем построить циклический путь из этого потока точно так же, как мы строили эйлеров путь на эйлеровом графе: мы производим обход некоторого цикла, выходя из этого цикла для обхода другого цикла всякий раз, когда выходим на вершину, которая входит в этот другой цикл. ■

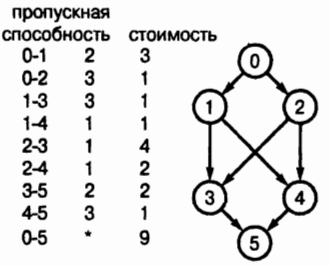
Подробный анализ задачи о почтальоне показывает, насколь зыбкая граница отделяет тривиальные проблемы от труднорешаемых в области алгоритмов на графах. Предположим, что мы рассматриваем двухсторонний вариант этой задачи, когда сеть неориентированная, а почтальон должен проходить каждое ребро в обоих направлениях. Затем, как мы отмечали в разделе 18.5, поиск в глубину (или любой другой поиск на графе) даст нам немедленное решение. Если, однако, достаточно одного обхода ребра в каком-либо направлении, то формулировка задачи окажется намного сложнее, чем просто сведение к задаче о потоке с минимальной стоимостью, с которой мы только что ознакомились, тем не менее, задача все еще не выходит из категории решаемых. Если некоторые ребра ориентированы, а другие нет, проблема переходит в категорию NP-трудных (см. *раздел ссылок*).

Это всего лишь небольшая часть из нескольких десятков практических задач, формулируемых как задачи о потоке минимальной стоимости. Задача о потоке минимальной стоимости более многообразна, нежели задачи вычисления максимального потока или построения кратчайшего пути, а сетевой симплексный алгоритм эффективно решает все задачи, охватываемые этой моделью.

Подобно тому, как мы поступали при изучении максимального потока, мы можем выяснить, как любую задачу о минимальном потоке представить в виде LP-задачи (см. рис. 22.53). Ее постановка есть простое расширение постановки задачи о максимальном потоке: мы добавляем уравнения, которые устанавливают значение фиктивной перемен-

ной равным стоимости потока, затем ставят цель минимизировать эту переменную. LP-модель позволяет добавить произвольные (линейные) ограничения. Некоторые из этих ограничений могут привести к задачам, эквивалентным задачам о потоках минимальной стоимости, другие – нет. Другими словами, многие задачи не сводятся к задачам о потоках минимальной стоимости: в частности, линейное программирование охватывает намного более широкое множество задач. Задача о потоке минимальной стоимости представляет собой следующий шаг в направлении той обобщенной модели решения задачи, которая будет рассматриваться в части 8.

Существуют другие модели, которые носят еще более общий характер, чем LP-модель; однако LP-модель обладают дополнительным достоинством, заключающимся в том, что LP-задачи в общем случае более сложные, чем задачи о потоке минимальной стоимости, и были разработаны более эффективные алгоритмы их решения. В самом деле, возможно, наиболее важный из этих алгоритмов известен как *симплексный* метод: сетевой симплексный метод есть специализированная версия симплексного метода, применимого к подмножеству LP-задач, которые соответствуют задачам о потоке минимальной стоимости, а понимание сетевого симплексного алгоритма можно рассматривать как первый шаг к пониманию полного симплексного алгоритма.



максимизировать -c в условиях следующих ограничений

$$x_{01} \le 2$$
$$x_{02} \le 3$$
$$x_{13} \le 3$$
$$x_{14} \le 1$$
$$x_{23} \le 1$$
$$x_{24} \le 1$$
$$x_{35} \le 2$$
$$x_{45} \le 3$$
$$x_{50} = x_{01} + x_{02}$$
$$x_{01} = x_{13} + x_{14}$$
$$x_{02} = x_{23} + x_{24}$$
$$x_{13} + x_{23} = x_{35}$$
$$x_{14} + x_{24} = x_{45}$$
$$x_{35} + x_{45} = x_{50}$$

РИСУНОК 22.53. ЗАДАЧА О МАКСИМАЛЬНОМ ПОТОКЕ МИНИМАЛЬНОЙ СТОИМОСТИ В LP-ФОРМУЛИРОВКЕ

Эта линейная программа эквивалентна задаче о максимальном потоке минимальной стоимости для примера сети, представленного на рис. 22.40. Равенства, описывающие условия для вершин, и неравенства, описывающие условия для ребер, те же, что и на рис.22.39, но при этом цели разные. Переменная c *представляет собой суммарную стоимость, которая является линейной комбинацией других переменных.*

В этом случае:

$c = -9x_{50} + 3x_{01} + x_{02} + x_{13} + x_{14} + 4x_{23} + 2x_{24} + 2x_{35} + x_{45}$.

Упражнения

○ **22.133**. Покажите, что если сетевой симплексный алгоритм вычисляет максимальный поток, остовное дерево есть объединение ребра **t-s**, дерева, содержащего вершину **s**, и дерева, содержащего вершину **t**.

22.134. Разработайте реализацию, вычисляющую максимальный поток на основе упражнения 22.133. Выберите подходящее ребро случайным образом.

22.135. Покажите в стиле рис. 22.50 процесс вычисления максимального потока в транспортной сети, изображенной на рис. 22.10, используя сведение к задаче, описанной в тексте, и реализацию сетевого симплексного алгоритма из программы 22.14.

22.136. Покажите в стиле рис. 22.50 процесс построения кратчайших путей из вершины 0 в транспортной сети, изображенной на рис. 22.10, используя сведение к задаче, описанной в тексте, и реализацию сетевого симплексного алгоритма из программе 22.14.

○ **22.137**. Докажите, что все ребра остовного дерева, описанного в доказательстве свойства 22.29, включены в пути, ведущие из источника к листьям.

○ **22.138**. Докажите, что остовное дерево, описанное в доказательстве свойства 22.29, соответствуют дереву кратчайших путей в исходной сети.

22.139. Предположим, что вы используете сетевой симплексный алгоритм для решения задачи, полученной в результате сведения задачи о кратчайших путях из единственного источника, как описано в доказательстве свойства 22.29. (*i*). Докажите, что этот алгоритм никогда не использует аугментальный путь нулевой стоимости. (*ii*) Покажите, что ребро, которое выходит из цикла, всегда есть родитель вершины назначения ребра, добавляемого в цикл. (*iii*) Как следствие упражнения 22.138, сетевой симплексный алгоритм не обязан поддерживать потоки в ребрах. Представьте полную реализацию, использующую все преимущества, вытекающего из этого факта. Выбор нового древесного ребра производится случайным образом.

22.140. Предположим, что мы присваиваем положительную стоимость каждому ребру сети. Докажите, что задача построения дерева кратчайших путей с единственным источником минимальной стоимости сводится к задаче о максимальном потоке минимальной стоимости.

22.141. Предположим, что мы модифицируем задачу планирования работ с конечными сроками завершения из раздела 21.6 таким образом, что работы могут нарушать сроки завершения, при этом они получают некоторую положительную стоимость, если это произойдет. Покажите, что модифицированная таким образом задача сводится к задаче о максимальном потоке минимальной стоимости.

22.142. Реализуйте класс, который обнаруживает максимальные потоки минимальной стоимости в распределительных сетях с отрицательной стоимостью. Воспользуйтесь полученным классом для решения упражнения 22.105 (по условиям которого все стоимости неотрицательны).

○ **22.143**. Предположим, что стоимости ребер **0-2** и **1-3** на рис. 22.40 суть -1, а не 1. Покажите, как найти максимальный поток минимальной стоимости путем преобразования заданной сети в сеть с положительными стоимостями, а затем вычислить максимальный поток с минимальной стоимостью на новой сети.

22.144. Реализуйте класс, который обнаруживает максимальные потоки минимальной стоимости в сетях с отрицательной стоимостью. Воспользуйтесь классом **MINCOST** (который предполагает, что все стоимости неотрицательны).

○ **22.145**. Зависят ли реализации из разделов 22.5 и 22.6 существенным образом от того факта, что цены принимают неотрицательные значения? Если зависят, укажите, в какой степени; если не зависят, покажите, какие настройки (если таковые имеются) требуется выполнить, чтобы заставить работать сеть с отрицательными стоимостями, либо объясните, почему такие настройки невозможны.

22.146. Расширьте полученный вами в упражнении 22.74 АТД допустимого потока за счет введения нижних границ пропускных способностей ребер. Реализуйте класс, который вычисляет максимальный поток минимальной стоимости, который соблюдает эти границы (если такой поток имеет место).

22.147. Покажите, к каким результатам приводит использование метода сведения задач, описанного в тексте, на примере сведения транспортной сети, описанной в упражнении 22.112, к транспортной задаче.

○ **22.148**. Покажите, что задача о максимальном потоке минимальной стоимости сводится к транспортной задаче со всего лишь V дополнительными вершинами и ребрами, если прибегнуть к построению, аналогичному тому, что применялось при доказательстве свойства 22.16.

▷ **22.149**. Реализуйте класс для решения транспортной задачи, основанный на простом сведении задачи о максимальном потоке минимальной стоимости, описанном в доказательстве свойства 22.30.

○ **22.150**. Разработайте реализацию класса для решения задачи вычисления потока минимальной стоимости, в основу которого положено сведение к транспортной задаче, описанное в доказательстве свойства 22.31.

○ **22.151**. Разработайте реализацию класса для решения задачи вычисления потока минимальной стоимости, в основу которого положено сведение к транспортной задаче, описанное в упражнении 22.148.

22.152. Напишите программу, генерирующую случайные экземпляры транспортной задачи, затем используйте их как основу для эмпирических испытаний различных алгоритмов и реализаций на способность решения этой задачи.

22.153. Дайте пример крупной динамической транспортной задачи.

22.154. Проведите эмпирические исследования по сравнению двух различных методов сведения произвольных задач о потоках минимальной стоимости к транспортной задаче, которые обсуждались при доказательстве свойства 22.31.

22.155. Напишите программу, генерирующую случайные экземпляры задачи о назначениях, затем используйте их в качестве основы для эмпирических испытаний различных алгоритмов и реализаций на способность решения этой задачи.

22.156. Дайте пример крупной динамической задачи о назначениях.

22.157. Задача о трудоустройстве, описанная в тексте, дает преимущества работодателям (их суммарная стоимость максимизирована). Сформулируйте версию этой задачи, в рамках которой претенденты на рабочие места также могли бы высказывать свои пожелания. Объясните, как решить вашу версию задачи.

22.158. Проведите эмпирические исследования по сравнению производительности двух реализаций сетевого симплексного алгоритма, описанных в разделе 22.6, применительно к решению вариантов задачи о назначениях (см. упражнение 22.155), с V вершинами и E ребрами и разумно выбранным множеством значений V и E.

22.159. Очевидно, что задача о почтальоне не имеет решения для сетей, которые не принадлежат к числу сильно связных (почтальон может нанести визит только в вершины, содержащиеся в сильной компоненте, в которой он начинает свой обход), однако этот факт не оговаривается свойством 22.33. Что произойдет, если мы попытаемся применить сведение к сети, которая не является сильно связной?

22.160. Проведите эмпирические исследования для различных взвешенных графов (см. упражнения 21.4–21.8) с тем, чтобы определить среднюю протяженность пути почтальона.

22.161. Дайте прямое доказательство того, что задача о кратчайшем пути из единственного источника сводится к задаче о назначениях.

22.162. Дайте формулировку произвольной задачи о назначениях как LP-задачи.

- **22.163**. Выполните упражнение 22.18 для случая, когда значение стоимости, назначенной каждому ребру, есть -1 (таким образом, вы минимизируете неиспользованное пространство грузовых машин).
- **22.164**. Найдите такую модель стоимостей для упражнения 22.18, чтобы ее решением был максимальный поток, требующий минимальное число дней.

22.8. Перспективы

Существуют четыре причины, в силу которых проводимые нами исследования алгоритмов достигают своей кульминации при изучении алгоритмов вычисления сетевых потоков. Во-первых, модель потоков в сети придает законную силу практическому применению абстракции графа в бесчисленных приложениях. Во-вторых, изученные нами алгоритмы вычисления максимальных потоков и потоков с минимальной стоимостью, суть естественные расширения алгоритмов на графах, изученные нами на примерах простых задач. В третьих, программные реализации этих алгоритмов показывают, сколь важную роль играют фундаментальные алгоритмы и структуры данных для достижения высокой производительности. В четвертых, модели максимальных потоков и потоков с минимальной стоимостью служат примерами применения подхода разработки все более общих моделей решения задач и их использования для решения широких классов задач. Наша способность разрабатывать эффективные алгоритмы, обеспечивающие решение таких задач, оставляет нам открытой возможность разработки еще более общих моделей, решающих такие задачи.

Прежде чем переходить к более детальному рассмотрению этих проблем, подготовим дальнейший контекст, перечислив важные задачи, которые нам не удалось изучить в этой главе, даже несмотря на то что они очень тесно связаны с известными задачами.

Задача о максимальном сочетании (maximum matching). В графах со взвешенными ребрами найти подмножество ребер, в котором ни одна из вершин не появляется больше одного раза и суммарный вес которых таков, что никакое другое множество ребер не имеет большего суммарного веса. Мы можем свести задачу о *сочетании с максимальным кардинальным числом (maximum cardinality matching)* в невзвешенных графах непосредственно к этой задаче, установив всем ребрам веса, равные 1.

Задача о назначениях и задача о сочетании с максимальным кардинальным числом сводятся к задаче максимального сочетания для графов общего вида. С другой стороны, максимальное сочетание не сводится к потокам минимальной стоимости, поэтому и рассмотренные выше алгоритмы к ней не применимы. Эта задача относится к числу решаемых, хотя объем вычислений в случае крупных графов, довольно-таки велик. Рассмотрение многочисленных методов, предложенных в попытках решить задачу сочетания на графах общего вида, заняло бы целый том: эта проблема относится к числу наиболее интенсивно исследуемых в теории графов. В этой книге мы подвели черту под потоком минимальной стоимости, но мы вернемся к задаче о максимальном сочетании в части 8.

Задача о многопродуктовом потоке (multicommodity flow). Предположим, что нам нужно вычислить второй поток, такой, что сумма обоих потоков в ребре ограничена пропускной способностью ребра, оба потока находятся в равновесии и их суммарная стоимость минимальна. Такое изменение моделирует наличие двух различных типов материалов в задаче о распределении товаров (merchandise-distribution problem); например, следует ли

нам загрузить больше гамбургеров или больше картошки в грузовую машину, направляющуюся в ресторан быстрого питания? Подобного рода изменения делают эту задачу более трудной и требуют более совершенных алгоритмов ее решения, нежели рассмотренные в этой книге; например, не известен ни один аналог теоремы о максимальном потоке минимальной стоимости, который выполняется в общем случае. Постановка этой задачи как LP-задачи есть простое расширение примера, представленного на рис. 22.53, следовательно, эта задача относится к числу решаемых (поскольку решаема LP-задача).

Выпуклость и нелинейные стоимости. Простые функции стоимости, которые мы рассматриваем, представляют собой линейную комбинацию некоторых переменных, а используемые нами алгоритмы для их решения существенно зависят от простых математических структур, лежащих в их основе. Например, когда мы минимизируем расстояния, мы имеем дело с суммами квадратов стоимостей. Такие задачи не могут быть сформулированы как LP-задачи, они требуют специальных моделей решения задач, которые обладают еще большими возможностями. Многие из этих задач не принадлежат к числу легко решаемых.

Задачи календарного планирования (составления расписаний). В качестве примеров мы рассмотрели несколько таких задач. Они являются представителями нескольких сот различных задач календарного планирования. Научно-исследовательская литература изобилует описаниями исследований отношений между этими задачами и разрабатываемыми алгоритмами их решения и соответствующими программными реализациями (см. *раздел ссылок*). В самом деле, мы могли бы воспользоваться алгоритмами составления расписаний вместо алгоритмов вычисления потоков в сетях, чтобы разрабатывать идеи, обеспечивающие построение обобщенных моделей решения задач, и реализации сведения конкретных задач к этой модели (то же самое можно сказать и о соответствии). Многие задачи составления расписания приводятся к модели потока минимальной стоимости.

Возможности комбинаторных вычислений поистине велики, поэтому изучение задач подобного типа, несомненно, будут занимать умы исследователей еще многие и многие годы. Мы вернемся к некоторым из этих задач в части 8, в контексте поиска решений трудно решаемых задач.

Мы ознакомились лишь с небольшой частью исследованных алгоритмов решения задач о максимальных потоках и о потоках минимальной стоимости. Как было показано в упражнениях на протяжении данной главы, комбинации многочисленных их вариантов при их использовании в различных частях алгоритмов общего вида порождает широкое множество самых разнообразных алгоритмов. Алгоритмы и структуры данных для базовых вычислительных задач играют важную роль в обеспечении эффективности многих из этих подходов; и в самом деле, некоторые важные алгоритмы общего назначения, с которыми мы ознакомились, были разработаны во время поиска эффективной реализации алгоритма вычисления потоков в сети. Эта тема все еще находится в центре внимания многих разработчиков. Разработка все более совершенных алгоритмов решения задач вычисления потоков в сетях, несомненно, зависит от правильного использования базовых алгоритмов и структур данных.

Широкая область применения алгоритмов вычисления потоков в сети и экстенсивное использование метода сведения к другим задачам с целью еще большего расширения этой области заставляют нас рассмотреть здесь некоторые последствия применения понятия

сводимости. Для обширного класса комбинаторных алгоритмов эти задачи представляют собой водораздел, когда мы поставлены перед необходимостью, с одной стороны, изучать эффективные алгоритмы решения конкретных задач, а, с другой стороны, изучать обобщенные моделей решения задач. Мощные силы склоняют нас то в одну, то в другую сторону.

У нас есть все основания разрабатывать максимально универсальную модель, поскольку чем выше уровень универсальности модели, тем больше задач она охватывает, благодаря чему повышается коэффициент полезного действия алгоритма, который может решать задачу, сводимую к этой модели. Разработка такого алгоритма может представлять собой трудно решаемую, если не неразрешимую задачу. Если в нашем распоряжении нет алгоритма, который гарантирует приемлемую эффективность решения, у нас обычно имеется алгоритм, который хорошо работает на специальных классах задач, которые нас интересуют. Специальные аналитические выкладки дают неконкретные результаты, однако, довольно-таки часто мы опираемся на эмпирические исследования. Действительно, практики обычно проводят апробацию наиболее общей модели из числа доступных (или той, которая предлагает хорошо отлаженный пакет решений) и не идут дальше, если модель не выходит за пределы приемлемого времени. Тем не менее, мы должны избегать использования чрезмерно общих моделей, ибо это может привести к излишним затратам времени на решение тех задач, для которых более специализированные модели могут обеспечить более высокую эффективность.

С другой стороны, мы склонны искать более совершенные алгоритмы решения важных специальных задач, в частности, алгоритмы решения крупных задач или большого числа меньших задач, когда вычислительные ресурсы являются узким местом. Как мы уже могли убедиться на многих примерах в этой книге и в частях 1–4, мы часто находим подходящий алгоритм, который позволяет снизить затраты ресурсов во многие сотни и тысячи раз и даже больше, что исключительно важно, когда мы измеряем затраты в часах или в долларах. Общий подход, описанный в главе 2, который мы успешно применяли во многих областях, исключительно полезен в таких ситуациях, и мы с нетерпением ждем появления более совершенных алгоритмов в спектре алгоритмов на графах и комбинаторных алгоритмов. Возможно, основной недостаток перенесения центра тяжести на специальные алгоритмы заключается в том, что довольно часто небольшое изменение, внесенное в модель, приводит к тому, что алгоритм становится для нее непригодным. В то же время, когда мы прибегаем к помощи чрезмерно обобщенной модели и алгоритма, обеспечивающего решения нашей задачи, мы меньше подвержены воздействиям упомянутого недостатка.

Программные библиотеки, охватывающие множество алгоритмов, с которыми мы ознакомились, можно найти во многих средах программирования. Такие библиотеки, несомненно, представляют собой важные ресурсы, которые нужно принимать во внимание при решении конкретных задач. Тем не менее, иногда такими библиотеками трудно пользоваться, они устаревают либо плохо соответствуют текущей задаче. Опытные программисты хорошо представляют себе, насколько важно найти компромисс между использованием преимуществ библиотечного ресурса и чрезмерной зависимостью от этого ресурса (даже если он преждевременно не устареет). Некоторые из рассмотренных ранее реализаций являются эффективными, простыми в разработке и широко используемы-

ми. Адаптация и настройка таких реализаций для решения текущих программ во многих ситуациях представляют собой разумный подход.

Напряженность между теоретические исследованиями, которые ограничены тем, что мы можем доказать, и эмпирическими исследованиями, которые имеют отношение только к текущим проблемам, становятся все более заметными по мере возрастания трудности решаемых нами задач. Теория обеспечивает руководство, которое нам необходимо, чтобы постигнуть суть задачи, а практический опыт подсказывает нам, куда направить свои усилия во время разработки реализаций. Более того, приобретенный опыт решения практических задач открывает новые направления развития теории, сохраняя цикл, дающий возможность расширить класс практических задач, которые мы можем решать.

В конечном итоге, какой бы подход мы не выбрали, цель остается прежней. Нам нужен широкий спектр моделей решения задач, эффективные алгоритмы решения задач в рамках этих моделей, а также эффективные реализации этих алгоритмов, позволяющих нам решать практические задачи. Разработка все более универсальных моделей решения задач (таких как задачи вычисления кратчайших путей, максимальных потоков, потоков минимальной стоимости), все более мощных алгоритмов общего характера (такие как алгоритм Беллмана-Форда решения задачи вычисления кратчайших путей, алгоритм построения аугментальных путей для задачи о максимальном потоке и сетевой симплексный алгоритм для задачи о максимальном потоке минимальной стоимости) позволили нам существенно продвинуться в направлении этой цели. Многое было достигнуто в пятидесятые и шестидесятые годы. Несколько позже появились фундаментальные структуры данных (части 1–4) и алгоритмы, обеспечивающие эффективные реализации этих общих методов (данная книга), ставшие той мощной силой, которая наделила нас способностью решать такой широкий класс крупных задач.

Ссылки, использованные в пятой части

В публикациях, указанных ниже, содержатся описания большей части фундаментальных алгоритмов, которые мы изучали в главах 17–21. Эти книги являются основными справочными пособиями, которые содержат подробный анализ фундаментальных и современных алгоритмов на графах, изобилующих пространными ссылками на современную литературу. В книге Ивена (Even) и монографии Тарьяна (Tarjan) подробно рассматриваются многое из того, что изучалось в этой книге. Оригинальная статья Тарьяна, в которой рассматриваются вопросы применения поиска в глубину для решения задач о связности и ряд других задач, заслуживает дальнейшего изучения, Описанная в главе 19 реализация топологической сортировки на базе очереди истоков взято из книги Кнута (Knuth). Ссылки на источники описаний других специальных алгоритмов, которые мы рассматривали в этой книге, приводятся ниже.

Алгоритмы построения минимальных остовных деревьев в насыщенных графах, которые мы изучали в главе 20, известны давно, однако пионерские статьи Дейкстры (Dijkstra), Прима (Prim) и Крускала (Kruskal) и сегодня заслуживают внимания. В обзоре, написанном Грэхемом (Graham) и Хеллом (Hell), подробно изложена занимательная история этой задачи. Статья Чейзела (Chazelle) описывает современное положение дел в поиске линейного алгоритма построения минимальных остовных деревьев.

Книга, написанная Ахуджа (Ahuja), Маньянти (Magnanti) и Орлином (Orlin) представляет собой всестороннее исследование алгоритмов вычисления потоков в сетях (и алгоритмов построения кратчайших путей). В этой книге вы найдете дополнительную информацию, касающуюся фактически каждой темы, рассмотренной в главах 21 и 22. Другим источником дальнейших материалов по этой теме может послужить классический труд Пападимитриу (Papadiminriou) и Стейлица (Steiglitz). Несмотря на то что в этой книге, главным образом, рассматриваются более сложные вопросы, в ней вы найдете подробное описание многих алгоритмов, изучаемых в данной книге. Обе книги содержат многочисленные и подробные ссылки на источники в научно-исследовательской литературе. Классическая работа Форда (Ford) и Фалкерсона (Fulkerson) все еще заслуживает изучения, поскольку в ней впервые вводятся многие фундаментальные понятия.

В этой книге мы кратко коснулись многих актуальных тем, которые подробно рассматриваются в части 8 (которая очень скоро увидит свет), в том числе вопросы сводимости, трудной разрешимости и линейного программирования. Предлагаемый здесь список публикаций охватывает материал, с которым мы здесь подробно ознакомились, и который, тем не менее, не дает полного представления об этой современной тематике. Многое можно почерпнуть из текстов алгоритмов, а книга Пападимитриу и Стейлица послужит прекрасным введением. Этим темам посвящено множество других книг и обширная научно-исследовательская литература.

R. K. Ahuja, T. L.Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

B. Chazelle, "A minimum spanning tree algorithm with inverse-Ackermann type complexity," *Journal of the ACM*, **47** (2000).

T. H. Cormen, C. L. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press and McGraw-Hill, 1990.

E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, **1** (1959).

P. Erd.os and A. Renyi, "On the evolution of random graphs," *Magyar Tud. Akad. Mat. Kutato Int Kozl*, **5** (1960).

S. Even, *Graph Algorithms*, Computer Science Press, 1979.

L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

H. N. Gabow, "Path-based depth-first search for strong and biconnected components," *Information Processing Letters*, **74** (2000).

R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem," *Annals of the History of Computing*, **7** (1985).

D. B. Johnson, "Efficient shortest path algorithms," Journal of the ACM, **24** (1977).

D. E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms*, third edition, Addison-Wesley, 1997.

J. R. Kruskal Jr., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings AMS*, **7**, 1 (1956).

K. Mehlhorn, *Data Structures and Algorithms 2: NP-Completeness and Graph Algorithms*, Springer-Verlag, 1984.

C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, 1982.

R. C. Prim, "Shortest connection networks and some generalizations," *Bell System Technical Journal*, **36** (1957).

R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, **1**, 2 (1972).

R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

Предметный указатель

Г

Д

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Ц

Ч

Э

Научное издание

Седжвик Роберт

ФУНДАМЕНТАЛЬНЫЕ АЛГОРИТМЫ НА C++

АЛГОРИТМЫ НА ГРАФАХ

Заведующий редакцией *С.Н.Козлов*
Научный редактор *Ю.Н.Артеменко*
Верстка *Т.Н.Артеменко*
Главный дизайнер *О.А.Шадрин*
Н/К

ООО «ДиаСофтЮП», 196105, Санкт-Петербург, пр. Ю.Гагарина, д. 1, ком. 108
Лицензия №000328 от 9 декабря 1999 г.

Сдано в набор 10.08.2002. Подписано в печать 08.10.2002. Формат 70×100/16.
Бумага типографская. Гарнитура Таймс. Печать офсетная. Печ.л. 21.
Тираж 3000 экз. Заказ № 857

Отпечатано с готовых диапозитивов
в ФГУП ордена Трудового Красного Знамени «Техническая книга»
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций
198005, Санкт-Петербург, Измайловский пр., 29.