ТИМ РАФГАРДЕН

СОВЕРШЕННЫЙ АЛГОРИТМ

ГРАФОВЫЕ АЛГОРИТМЫ И СТРУКТУРЫ

COMPUTER SCIENCE

Algorithms Illuminated

Part 2: Graph Algorithms and Data Structures

Tim Roughgarden

ТИМ РАФГАРДЕН

СОВЕРШЕННЫЙ АЛГОРИТМ

ГРАФОВЫЕ АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

COMPUTER SCIENCE

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2019

ББК 32.973.2-018
УДК 004.42
Р26

Рафгарден Тим

Р26 Совершенный алгоритм. Графовые алгоритмы и структуры данных. — СПб.: Питер, 2019. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1272-2

Алгоритмы — это сердце и душа computer science. Без них не обойтись, они есть везде — от сетевой маршрутизации и расчетов по геномике до криптографии и машинного обучения. «Совершенный алгоритм» превратит вас в настоящего профи, который будет ставить задачи и мастерски их решать как в жизни, так и на собеседовании при приеме на работу в любую IT-компанию.

Во второй книге Тим Рафгарден, гуру алгоритмов, расскажет о графовом поиске и его применении, алгоритме поиска кратчайшего пути, а также об использовании и реализации некоторых структур данных: куч, деревьев поиска, хеш-таблиц и фильтра Блума.

Серия книг «Совершенный алгоритм» адресована тем, у кого уже есть опыт программирования, и основана на онлайн-курсах, которые регулярно проводятся с 2012 года. Вы перейдете на новый уровень, чтобы увидеть общую картину, разобраться в низкоуровневых концепциях и математических нюансах.

Познакомиться с дополнительными материалами и видеороликами автора (на английском языке) можно на сайте www.algorithmsilluminated.org.

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

ББК 32.973.2-018
УДК 004.42

ISBN 978-0999282908 англ.
ISBN 978-5-4461-1272-2

Оглавление

Памяти Джеймса Уэсли Шина
(1921–2010)

Предисловие

Перед вами вторая книга серии из четырех частей, изданная на основе проводимых мною с 2012 года онлайн-курсов по алгоритмам. Эти курсы, в свою очередь, появились благодаря лекциям для студентов, которые я читал в Стэнфордском университете в течение долгих лет. Первая книга — «Совершенный алгоритм. Основы» — не является обязательным условием для второй части, она самостоятельна и будет доступна любому читателю, кто имеет подготовку, описанную в приведенном ниже разделе «Для кого эта книга?», и кто знаком с асимптотическими обозначениями (рассматриваемыми в приложении В).

О чем эта книга

Вторая часть книги «Совершенный алгоритм» — это вводный курс в основы грамотности по следующим трем темам.

Графовый поиск и приложения. Графы моделируют ряд разных типов сетей, включая дорожные, коммуникационные, социальные сети и сети зависимостей между задачами. Графы могут быть сложными, однако существует несколько невероятно быстрых примитивов для рассуждения о структуре графа. Мы начнем с линейно-временных алгоритмов графового поиска, с приложений в диапазоне от сетевого анализа до построения последовательности выполнения операций.

Кратчайшие пути. В задаче о кратчайшем пути цель состоит в том, чтобы вычислить наилучший маршрут в сети из точки A в точку B. Данная задача имеет очевидные применения, такие как вычисление маршрутов движения, а также встречается в замаскированном виде во многих других универсальных задачах. Мы обобщим один из наших алгоритмов графового поиска и придем к знаменитому алгоритму поиска кратчайшего пути Дейкстры.

Структуры данных. Эта книга сделает вас высокообразованным пользователем нескольких разных структур данных, предназначенных для поддержания эволюционирующего множества объектов с ассоциированными с ними ключами. Основная цель состоит в развитии интуиции о том, какая структура данных является правильной для вашего приложения. Дополнительные разделы содержат рекомендации по реализации этих структур данных с нуля.

Сначала мы обсудим кучи, которые могут быстро идентифицировать хранимый объект с наименьшим ключом, а также полезны для сортировки, реализации очереди с приоритетом и реализации почти линейно-временного

алгоритма Дейкстры. Деревья поиска сохраняют полную упорядоченность ключей хранимых объектов и поддерживают еще более широкий спектр операций. Хеш-таблицы оптимизированы для сверхбыстрых операций поиска и широко распространены в современных программах. Мы также рассмотрим фильтр Блума, близкого родственника хеш-таблицы, который использует меньше пространства за счет случайных ошибок.

Более подробно ознакомиться с содержанием книги можно в разделах «Выводы», которые завершают каждую главу и вычленяют наиболее важные моменты. Разделы книги, помеченные звездочкой, являются наиболее продвинутыми по уровню изложенной информации. Если книга рассчитана на поверхностное ознакомление с темой, то читатель может пропустить их без потери целостности написанного.

Темы, затронутые в трех других частях. *Первая часть книги* «*Совершенный алгоритм. Основы»* охватывает асимптотические обозначения (форму записи *O*-большое и ее близких родственников), алгоритмы «разделяй и властвуй» и основную теорему о рекуррентных соотношениях — основной метод, рандомизированную быструю сортировку и ее анализ, а также линейно-временные алгоритмы отбора. В *третьей части* речь идет о жадных алгоритмах (планирование, минимальные остовные деревья, кластеризация, коды Хаффмана) и динамическом программировании (задача о рюкзаке, выравнивание последовательности, кратчайшие пути, оптимальные деревья поиска). *Четвертая часть* посвящена NP-полноте, тому, что она означает для проектировщика алгоритмов, и стратегиям решения вычислительно неразрешимых задач, включая эвристический анализ и локальный поиск.

Навыки, которые вы приобретете

Освоение алгоритмов требует времени и усилий. Ради чего все это?

Возможность стать более эффективным программистом. Вы изучите несколько невероятно быстрых подпрограмм для обработки данных и некоторые полезные структуры для организации данных, которые сможете непосредственно использовать в ваших собственных программах. Реализация и применение этих алгоритмов расширит и улучшит ваши навыки программирования. Вы также узнаете основные приемы разработки алгоритмов, которые актуальны для решения разнообразных задач в широких областях, получите инструменты для прогнозирования производительности этих алгоритмов. Такие «шаблоны» могут пригодиться вам для разработки новых алгоритмов решения задач, которые возникают в вашей собственной работе.

Развитие аналитических способностей. Алгоритмические описания, мыслительная работа над алгоритмами дают большой опыт. Посредством математического анализа вы получите углубленное понимание конкретных алгоритмов и структур данных, описанных в этой и последующих книгах серии. Вы приобретете навыки работы с несколькими математическими методами, которые широко применяются для анализа алгоритмов.

Алгоритмическое мышление. Научившись разбираться в алгоритмах, вы начнете замечать, что они окружают вас повсюду: едете ли вы в лифте, наблюдаете ли за стаей птиц, управляете ли вы своим инвестиционным портфелем или даже присматриваете за тем, как учится ребенок. Алгорит-

мическое мышление становится все более полезным и превалирующим в дисциплинах, не связанных с информатикой, включая биологию, статистику и экономику.

Знакомство с величайшими достижениями computer science. Изучение алгоритмов напоминает просмотр эффектного клипа с суперхитами последних шестидесяти лет развития computer science. На фуршете для специалистов в области computer science вы больше не будете чувствовать себя не в своей тарелке, если кто-то отпустит шутку по поводу алгоритма Дейкстры. Прочитав эти книги, вы точно сможете поддержать разговор.

Успешность в собеседованиях. На протяжении многих лет студенты развлекали меня рассказами о том, как знания, почерпнутые из этих книг, позволяли им успешно справляться с любым вопросом, на который им требовалось ответить во время собеседования.

В чем особенность книг этой серии

Они преследуют только одну цель: *максимально доступным способом научить вас основам алгоритмов*. Относитесь к ним как к конспекту, который бы у вас был, если бы вы брали нескольких индивидуальных уроков у опытного наставника по алгоритмам.

Существует ряд прекрасных, гораздо более традиционных и энциклопедически выверенных учебников по алгоритмам. Призываю вас разузнать подробнее и найти книги, отвечающие вашему запросу. Кроме того, есть несколько отличных от этих книг, которые ориентируются на программистов, ищущих готовые реализации алгоритмов на конкретном языке программирования. Многие такие реализации находятся в свободном доступе в интернете.

Для кого эта книга?

Весь смысл этой книги, как и онлайн-курсов, на которых она базируется, — быть широко и легко доступной читателю настолько, насколько это возможно. Мои курсы вызвали интерес у людей разных возрастов, общественного положения и профессионального опыта, среди них немало учащихся и студентов, разработчиков программного обеспечения (как состоявшихся, так и начинающих), ученых и мегаспециалистов со всех уголков мира.

Эта книга — не введение в программирование, большим подспорьем было бы владение вами основными навыками программирования на каком-либо распространенном языке (например, Java, Python, C, Scala, Haskell). Чтобы пройти «лакмусовый» тест, обратитесь к разделу 8.2 — если там вам все будет понятно, значит, в остальной части книги у вас не возникнут вопросы. Если вам требуется развить свои навыки программирования, то для этих целей есть несколько прекрасных бесплатных онлайн-курсов, обучающих основам программирования.

По мере необходимости мы также используем математический анализ, чтобы разобраться в том, как и почему алгоритмы действительно работают. Находящиеся в свободном доступе конспекты лекций «Математика для Computer Science» под авторством Эрика Лемана, Тома Лейтона и Альберта Мейера являются превосходным и занимательным повторительным курсом по системе математических обозначений (например, $\sum$ и $\forall$), основам теории доказательств (метод индукции, доказательство от противного и др.), дискретной вероятности и многому другому.

Дополнительные ресурсы

Эта книга основана на онлайн-курсах, которые в настоящее время запущены в рамках проектов Coursera и Stanford Lagunita. Я также создал несколько ресурсов вам в помощь для повторения и закрепления опыта, который можно извлечь из онлайн-курсов.

Видео. Если вы больше настроены смотреть и слушать, нежели читать, обратитесь к материалам на YouTube, доступным на сайте www.algorithmsilluminated.org. Эти видео затрагивают все темы этой серии книг, а также дополнительные, более сложные. Надеюсь, что они лучше источают заразительный энтузиазм к алгоритмам, который, увы, невозможно полностью передать на бумажной странице.

Тестовые задания. Как проверить, что вы действительно усваиваете понятия, представленные в этой книге? Тестовые задания с решениями и объяснениями разбросаны по всему тексту; когда вы сталкиваетесь с одним из них, призываю вас остановиться и подумать над ответом, прежде чем продолжать чтение.

Задачи в конце главы. В конце каждой главы вы найдете несколько относительно простых вопросов на проверку усвоения материала, а затем более трудные и менее сложные задачи, не ограниченные по времени выполнения. Решения этих задач в книгу не включены, но для этого читатели могут обратиться ко мне или взаимодействовать между собой через дискуссионный форум книги (см. ниже).

Задачи по программированию. В конце большинства глав предлагается реализовать программный проект, целью которого является закрепление детального понимания алгоритма путем создания его рабочей реализации.

Наборы данных, а также тестовые примеры и их решения можно найти на сайте www.algorithmsilluminated.org.

Дискуссионные форумы. Основной причиной успеха онлайн-курсов является возможность общения для слушателей, реализованная через дискуссионные форумы. Это позволяет им помогать друг другу в лучшем понимании материала курса, а также отлаживать свои программы посредством взаимодействия и обсуждения. Читатели этих книг имеют такую же возможность благодаря форумам, доступным на сайте www.algorithmsilluminated.org.

Благодарности

Эта и остальные книги не появились бы на свет без того энтузиазма и «интеллектуального голода», которые демонстрируют сотни тысяч слушателей моих курсов по алгоритмам на протяжении многих лет как на кампусе в Стэнфорде, так и на онлайн-платформах. Я особенно благодарен тем, кто оставлял подробные отзывы в отношении более раннего проекта этой книги: Тоне Бласт, Юаню Цао, Джиму Хьюмелсайну, Владимиру Кокшеневу, Байраму Кулиеву, Патрику Монкелбэну и Дэниелу Зингаро.

Я всегда ценю обратную связь от читателей, которые вносят свои предложения. О них лучше всего сообщать на упомянутом выше форуме.

Тим Рафгарден
Лондон, Великобритания
июль 2018

От издательства

Не удивляйтесь, что эта книга начинается с седьмой главы. С одной стороны, она является частью курса «Совершенный алгоритм» Тима Рафгардена, а с другой — самостоятельным изданием, в котором рассматриваются вопросы графов и структур данных. Приложения А и Б вы можете найти в первой книге серии.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение! На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

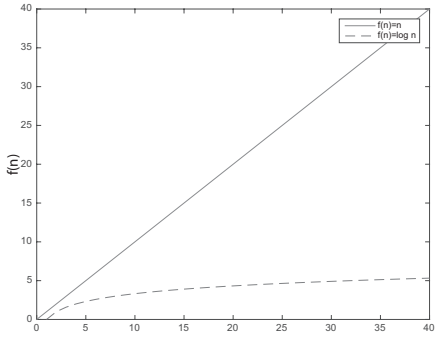
7

Графы: основы

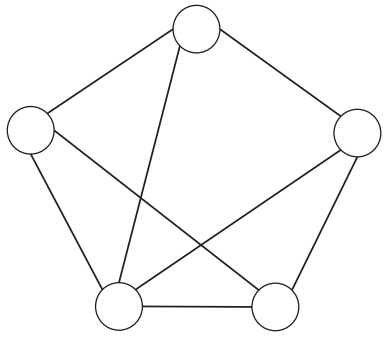
Эта небольшая глава объясняет, что такое графы и в чем их польза, а также показывает наиболее распространенные способы их представления в компьютерной программе. В двух последующих главах речь пойдет об известных и полезных алгоритмах для рассуждений о графах.

7.1. Термины

Когда вы слышите слово «граф», первое, о чем вы думаете, это об оси *X*, оси *Y* и так далее (рис. 7.1, *а*)[1]. Для человека с алгоритмическим мышлением *граф* также может означать представление связей между парами объектов (рис. 7.1, *б*).



а) График (для большинства людей)



б) Граф (в алгоритмах)

Рис. 7.1. В алгоритмах граф представляет множество объектов (например, людей) и попарные связи между ними (например, дружеские отношения)

В информационном плане граф имеет две компоненты — представляемые объекты и их попарные связи. Первые называются *вершинами*, или *узлами* графа[2]. Попарные связи называются *ребрами* графа. Мы обычно обозначаем множества вершин и ребер графа буквами *V* и *E* соответственно, и в неко-

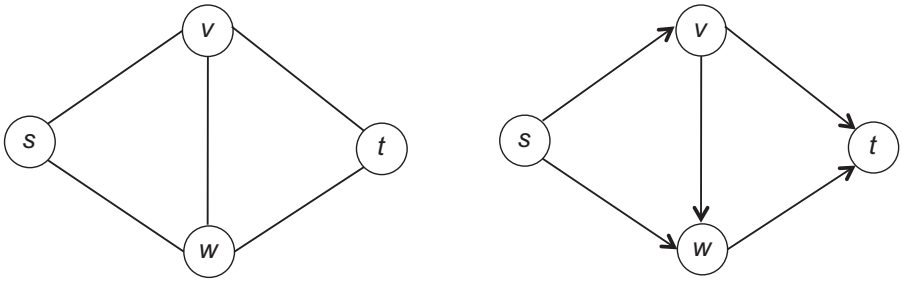
---

[1] В английском языке понятия «график» и «граф» обозначаются одним словом — *graph*. — *Примеч. пер.*

[2] Наличие двух названий для обозначения одной и той же вещи может раздражать, но оба приведенных в книге термина широко используются, и наша задача — познакомить вас с ними. На протяжении всей этой серии книг мы по большей части будем придерживаться термина «вершина».

торых случаях пишем $G = (V, E)$ для обозначения графа G с вершинами V и ребрами E.

Существует два вида графов, ориентированные и неориентированные. Оба важны и повсеместно применяются в приложениях, поэтому вы должны иметь представление о каждом из них. В *неориентированном* графе каждое ребро соответствует неупорядоченной паре $\{v, w\}$ вершин, которые называются *конечными точками* ребра (рис. 7.2, *а*). В таком графе нет разницы между ребром (v, w) и ребром (w, v). В *ориентированном* графе каждое ребро (v, w) представляет собой упорядоченную пару, причем ребро проходит из первой вершины v (называемой *хвостом*) во вторую вершину w (*голову*), как показано на рис. 7.2, *б*[1].



а) Неориентированный граф

б) Ориентированный граф

Рис. 7.2. Графы с четырьмя вершинами и пятью ребрами. Ребра неориентированных и ориентированных графов представляют собой неупорядоченные и упорядоченные пары вершин соответственно

7.2. Несколько приложений

Графы, будучи фундаментальным понятием, то и дело «всплывают» в информатике, биологии, социологии, экономике и прочих научных областях. Вот лишь несколько примеров из их бесчисленного множества.

Дорожная сеть. Когда ПО смартфона вычисляет маршруты движения, оно выполняет поиск в графе, представляющем дорожную сеть, вершины которой соответствуют пересечениям, а ребра — отдельным участкам дороги.

---

[1] Ориентированные ребра иногда называют *дугами*, но в этой серии книг данная терминология применяться не будет.

Всемирная паутина. Интернет может быть смоделирован как ориентированный граф; его вершины будут соответствовать отдельным веб-страницам, ребра — гиперссылкам, направленным со страницы, содержащей гиперссылку, на целевую страницу.

Социальные сети. Социальную сеть можно представить в виде графа, вершины которого соответствуют отдельной личности и ребра которого соответствуют некоторому типу связей. Например, ребро может указывать на дружбу между конечными точками или на то, что одна из конечных точек «читает» другую. Какие из наиболее популярных ныне социальных сетей смоделированы как неориентированный граф, а какие — как ориентированный? (Существуют интересные примеры обоих видов сетей.)

Ограничения по предшествованию. Не менее полезны графы и при решении задач, не имеющих очевидной сетевой структуры. Представьте, что вам нужно выполнить сразу несколько задач с учетом ограничений по предшествованию — возможно, вы студент первого курса университета, который берет дисциплины на выбор и раздумывает над первоочередностью каждой. Одним из способов решения подобной головоломки является применение описанного в разделе 8.5 алгоритма топологической сортировки к следующему ориентированному графу: для каждого теоретического предмета есть одна вершина с ребром, направленным из дисциплины А в дисциплину Б, когда А является необходимым условием для Б.

7.3. Измерение размера графа

В этой книге, как и в первой части[1], мы будем анализировать время работы разных алгоритмов в зависимости от размера входных данных. Когда входные данные представляют собой один массив, как для алгоритма сортировки, существует очевидный способ определить размер входных данных как длину массива. Когда входные данные включают граф, мы должны точно указать, как он представлен и что мы подразумеваем под его размером.

---

[1] *Тим Рафгарден*. Совершенный алгоритм. Основы. — СПб.: Питер, 2019. — 256 с.: ил. — (Серия «Библиотека программиста»).

7.3.1. Число ребер в графе

Размер графа управляется двумя параметрами: числом вершин и числом ребер. Вот наиболее распространенные обозначения этих величин.

ОБОЗНАЧЕНИЯ ДЛЯ ГРАФОВ

Для графа $G = (V, E)$ с множеством вершин V и множеством ребер E:

- $n = |V|$ обозначает число вершин;
- $m = |E|$ обозначает число ребер[1].

Приведенное далее тестовое задание предлагает вам подумать, как число m ребер в неориентированном графе может зависеть от числа n вершин. Для поиска ответа на этот вопрос будем считать, что между каждой парой вершин существует не более одного неориентированного ребра — параллельные ребра не допускаются. Предположим также, что граф является «связным». Мы определим это понятие формально в разделе 8.3; интуитивно связность означает, что граф существует как одно целое без возможности разбить его на две части таким образом, чтобы ребра между этими частями не пересекались. Графы на рис. 7.1, *б* и 7.2, *а* являются связными, в то время как граф на рис. 7.3 — нет.

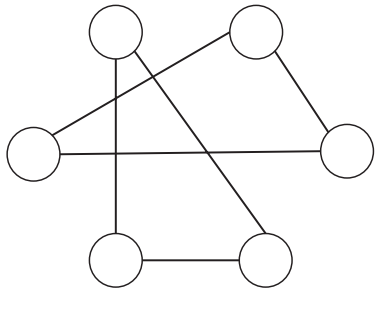


Рис. 7.3. Неориентированный граф, который не является связным

[1] Для конечного множества S число его элементов обозначается как $|S|$.

ТЕСТОВОЕ ЗАДАНИЕ 7.1

Рассмотрим неориентированный граф с n вершинами и без параллельных ребер. Будем считать, что граф является связным, то есть существует как одно целое. Какое минимальное и максимальное число ребер соответственно может иметь такой граф?

а) $n - 1$ и $n(n - 1)/2$

б) $n - 1$ и n^2

в) n и 2^n

г) n и n^n

(Решение и пояснение см. в разделе 7.3.3.)

7.3.2. Разреженные и плотные графы

Теперь, когда тестовое задание 7.1 заставило вас задуматься о том, как число ребер графа может меняться в зависимости от числа вершин, можем обсудить разницу между *разреженными* и *плотными* графами. Знать это отличие крайне важно, потому что некоторые структуры данных и алгоритмы лучше подходят для разреженных графов, а другие — для плотных.

Переведем решение тестового задания 7.1 в асимптотическую форму записи[1]. Во-первых, если неориентированный граф с n вершинами является связным, то число ребер m по крайней мере линейно по n (то есть $m = \Omega(n)$)[2]. Во-вторых, если граф не имеет параллельных ребер, то $m = O(n^2)$[3]. Мы заключаем, что число ребер в связном неориентированном графе без параллельных ребер находится где-то между линейным и квадратичным числом вершин.

---

[1] Обзор форм записи *O*-большое, Омега-большое и Тета-большое приведен в приложении В.

[2] Если граф не обязательно должен быть связным, то может существовать вплоть до нуля ребер.

[3] Если параллельные ребра разрешены, то граф по крайней мере с двумя вершинами может иметь сколь угодно большое число ребер.

Неформально граф является *разреженным*, если число ребер относительно близко к линейному по числу вершин, и является *плотным*, если это число ближе к квадратичному по числу вершин. Например, графы с n вершинами и $O(n \log n)$ ребрами обычно считаются разреженными, в то время как графы с $\Omega(n^2 / \log n)$ ребрами — плотными. Частично плотные графы, такие как графы с $\approx n^{3/2}$ ребрами, могут считаться либо разреженными, либо плотными — в зависимости от конкретного применения.

7.3.3. Решение тестового задания 7.1

Правильный ответ: (а). В связном неориентированном графе с n вершинами и без параллельных ребер число m ребер равно не менее $n - 1$ и не более $n(n - 1)/2$. Для того чтобы понять, почему нижняя граница является правильной, рассмотрим граф $G = (V, E)$. В качестве мысленного эксперимента представьте, что вы строите G по одному ребру за раз, начиная с графа с вершинами V и без ребер. Первоначально, перед добавлением ребер, каждая из n вершин полностью изолирована, поэтому граф тривиально имеет n разных частей. Добавление ребра (v, w) создает эффект слияния части, содержащей v, с частью, содержащей w (рис. 7.4). Тем самым каждое добавление ребра уменьшает число частей не более чем на 1[1]. Для того чтобы добраться до единственной части, состоящей из n частей, вам нужно добавить не менее $n - 1$ ребер. Существует целый ряд связных графов, имеющих n вершин и только $n - 1$ ребер. Такие графы называются *деревьями* (рис. 7.5).

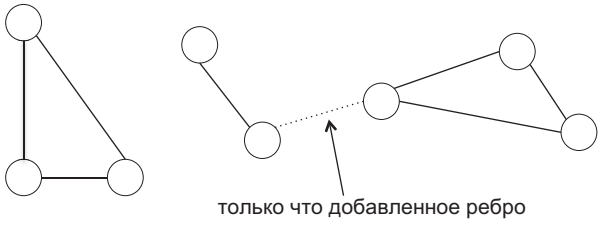


Рис. 7.4. Добавление нового ребра объединяет части, содержащие его конечные точки, в одну часть. В данном примере число разных частей уменьшается с трех до двух

[1] Если обе конечные точки ребра уже находятся в одной и той же части, то число частей не уменьшается.

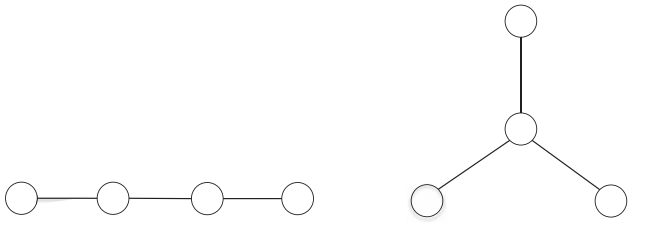


Рис. 7.5. Два связных неориентированных графа с четырьмя вершинами и тремя ребрами

Максимальное число ребер в графе без параллельных ребер достигается *полным графом*, в котором присутствуют все возможные ребра.

Поскольку в n-вершинном графе существует $\binom{n}{2} = \frac{n(n-1)}{2}$ пар вершин, это также является максимальным числом ребер. Например, при $n = 4$ максимальное число ребер равно $\binom{4}{2} = 6$ (рис. 7.6)[1, 2].

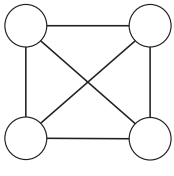


Рис. 7.6. Полный граф на четырех вершинах имеет $\binom{4}{2} = 6$ ребер

[1] $\binom{n}{2}$ произносится как «из n по 2», и также иногда упоминается как «биномиальный коэффициент». Для того чтобы понять, почему число способов выбрать неупорядоченную пару несовпадающих объектов из множества, состоящего из n объектов, равно $n(n-1)/2$, подумайте о выборе первого объекта (из n вариантов), а затем второго, несовпадающего объекта (из $n-1$ оставшихся вариантов). $n(n-1)$ результирующих исходов порождают каждую пару (x, y) объектов дважды (один раз, когда x является первым и y является вторым, один раз, когда y является первым и x является вторым), поэтому в совокупности должно быть $n(n-1)/2$ пар. — *Примеч. пер.*

[2] «из n по 2», потому что существует $\binom{n}{2}$ способов выбрать (неупорядоченное) подмножество из 2 элементов из фиксированного множества из n элементов. — *Примеч. пер.*

7.4. Представление графа

Существует несколько способов кодирования графа для использования в алгоритме. В этой книжной серии мы будем работать в основном с представлением графа в виде списков смежности (раздел 7.4.1), но вы должны знать и о представлении в виде матрицы смежности (раздел 7.4.2).

7.4.1. Списки смежности

Представление графов в виде списков смежности является доминирующим, и в этой книжной серии мы будем использовать его.

ИНГРЕДИЕНТЫ ДЛЯ СПИСКОВ СМЕЖНОСТИ

1. Массив, содержащий вершины графа.
2. Массив, содержащий ребра графа.
3. Для каждого ребра — указатель на каждую из двух его конечных точек.
4. Для каждой вершины — указатель на каждое инцидентное ребро[1].

Представление в виде списков смежности сводится к двум массивам (или же связным спискам, если вы предпочитаете их): один для отслеживания вершин и один для ребер. Эти два массива естественным образом ссылаются друг на друга, причем каждое ребро связано с указателями на его конечные точки, а каждая вершина — с указателями на ребра, для которых она является конечной точкой.

---

[1] Обычно говорят о смежных, или соседних, вершинах, но об инцидентных ребрах. Две вершины называются смежными, если они соединены ребром. Два ребра называются инцидентными, если они имеют общую вершину. Кроме того, вершина и ребро называются инцидентными, если вершина является одной из двух вершин, которые данное ребро соединяют. — *Примеч. пер.*

В случае ориентированного графа каждое ребро отслеживает, какая конечная точка является хвостовой и какая — головной. Каждая вершина v содержит два массива указателей: один для исходящих ребер (для которых v является хвостом) и один для входящих ребер (для которых v является головой).

Каковы потребности в памяти для представления графа в виде списков смежности?

ТЕСТОВОЕ ЗАДАНИЕ 7.2

Сколько места требуется для представления графа в виде списков смежности как функции от числа n вершин и числа m ребер?

а) $\Theta(n)$

б) $\Theta(m)$

в) $\Theta(m+n)$

г) $\Theta(n^2)$

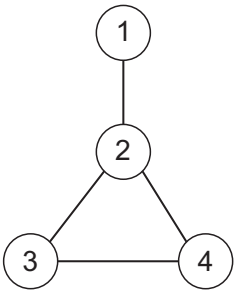
(Решение и пояснение см. в разделе 7.4.4.)

7.4.2. Матрица смежности

Рассмотрим неориентированный граф $G = (V, E)$ с n вершинами и без параллельных ребер и обозначим его вершины $1, 2, 3, ..., n$. Представление графа G в виде матрицы смежности — это квадратная $n \times n$-матрица A, равным образом двумерный массив с нулями и единицами в качестве элементов данных. Каждый элемент A_{ij} определяется как

$$A_{ij} = \begin{cases} 1, & \text{если ребро } (i, j) \text{ принадлежит } E \\ 0 & \text{– в противном случае.} \end{cases}$$

Таким образом, матрица смежности хранит один бит для каждой пары вершин, который отслеживает, присутствует ли ребро или нет (рис. 7.7).



а) Граф

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

б) ...и его матрица смежности

Рис. 7.7. Матрица смежности графа содержит один бит для каждой пары вершин, указывающий на то, существует или нет ребро, соединяющее две вершины

К представлению графа в виде матрицы смежности легко добавить «навороты».

- *Параллельные ребра*. Если граф может иметь несколько ребер с одной и той же парой конечных точек, то A_{ij} может быть определено как число ребер с конечными точками i и j.
- *Взвешенные графы*. Схожим образом, если каждое ребро (i, j) имеет вес w_{ij} — возможно, представляющий стоимость или расстояние, — тогда каждый элемент A_{ij} хранит w_{ij}.
- *Ориентированные графы*. В случае ориентированного графа G каждый элемент A_{ij} матрицы смежности определяется как

$$A_{ij} = \begin{cases} 1, & \text{если ребро } (i, j) \text{ принадлежит } E \\ 0 - \text{в противном случае,} \end{cases}$$

где ребро (i, j) теперь относится к ребру, ориентированному из i в j. Каждый неориентированный граф имеет симметричную матрицу смежности, в то время как ориентированный граф обычно имеет асимметричную матрицу смежности.

Каковы потребности в памяти для представления графа в виде матрицы смежности?

ТЕСТОВОЕ ЗАДАНИЕ 7.3

Сколько места требуется матрице смежности графа как функции от числа n вершин и числа m ребер?

а) $\Theta(n)$

б) $\Theta(m)$

в) $\Theta(m + n)$

г) $\Theta(n^2)$

(Решение и пояснение см. в разделе 7.4.4.)

7.4.3. Сравнение представлений

Столкнувшись с двумя разными способами представления графа, вы, вероятно, задаетесь вопросом: какое из них лучше? Ответ, как это часто бывает с вопросами подобного рода, таков: смотря по обстоятельствам. Во-первых, все зависит от плотности вашего графа — от того, как число m ребер соотносится с числом n вершин. Мораль тестовых заданий 7.2 и 7.3 заключается в том, что матрица смежности является эффективным способом кодирования плотного графа, но расточительна для разреженного графа. Во-вторых, не последнюю роль играет то, какие операции вы хотите поддерживать. В обоих случаях списки смежности имеют больше смысла для алгоритмов и приложений, описанных в данной книжной серии.

Большинство наших графовых алгоритмов будет включать в себя разведывание графа, для чего идеально подходят списки смежности — вы прибываете в вершину, и список смежности сразу же указывает ваши варианты для следующего шага[1]. Матрицы смежности имеют свои собственные применения, но в нашей книжной серии они опущены[2].

---

[1] Если бы у вас был доступ только к матрице смежности графа, сколько бы времени вам потребовалось, чтобы выяснить, какие ребра инцидентны данной вершине?

[2] Например, можно подсчитать число общих соседей каждой пары вершин «скопом», взяв квадрат матрицы смежности графа.

Бóльшая часть современного интереса к быстрым графовым примитивам обусловлена массивными разреженными сетями. Рассмотрим, к примеру, веб-граф (раздел 7.2), вершины которого соответствуют веб-страницам, а ребра — гиперссылкам. Трудно получить его точный размер, но консервативная нижняя граница числа вершин равна 10 миллиардам, или 10^{10}. Хранение и чтение массива такой длины уже требует значительных вычислительных ресурсов, но это вполне в пределах возможностей современных компьютеров. Однако размер матрицы смежности такого графа пропорционален 100 квинтиллионам (10^{20}). Он слишком большой, чтобы хранить или обрабатывать его с использованием сегодняшних технологий. Но веб-граф является разреженным — среднее число исходящих ребер из вершины значительно меньше 100. Следовательно, потребность в памяти для представления в виде списков смежности для веб-графа пропорциональна 10^{12} (триллиону). Это число может оказаться слишком большим для вашего ноутбука, но оно будет в пределах возможностей современных систем обработки данных[1].

7.4.4. Решения тестовых заданий 7.2–7.3

Решение тестового задания 7.2

Правильный ответ: (в). Представление в виде списков смежности требует пространства, линейного по размеру графа (имеется в виду число вершин плюс число ребер), которое является идеальным[2]. Увидеть это довольно непросто. Давайте один за другим пройдемся по всем четырем ингредиентам. Вершинный и реберный массивы имеют длины n и m соответственно, и поэтому требуют пространства $\Theta(n)$ и $\Theta(m)$. Третий компонент связывает два указателя с каждым ребром (по одному для каждой конечной точки). Эти $2m$ указателей вносят дополнительное $\Theta(m)$ к потребности в пространстве.

---

[1] Например, суть оригинального алгоритма `PageRank` от Google, предназначенного для измерения важности веб-страницы, основывалась на эффективном поиске в веб-графе.

[2] Предостережение: здесь ведущий постоянный множитель (или фактор) на порядок больше, чем для матрицы смежности.

Четвертый ингредиент может заставить вас поволноваться. В конце концов, каждая из n вершин может участвовать в $n - 1$ ребрах — по одному в расчете на другую вершину, что, по-видимому, приводит к границе $\Theta(n^2)$. Эта квадратичная оценка была бы точной в очень плотном графе, но в более разреженных графах она чрезмерна. Ключевое понимание состоит в следующем: *для каждого указателя вершина → ребро в четвертом ингредиенте существует соответствующий указатель ребро → вершина в третьем ингредиенте*. Если ребро e инцидентно вершине v, то e имеет указатель на его конечную точку v и, наоборот, v имеет указатель на инцидентное ребро e. Мы заключаем, что указатели в третьем и четвертом ингредиентах находятся во взаимно однозначном соответствии, и поэтому они требуют точно такого же объема пространства, а именно $\Theta(m)$. Итоговый перечень показателей приведен ниже:

	массив вершин	$\Theta(n)$
	массив ребер	$\Theta(m)$
	указатели из ребер в конечные точки	$\Theta(m)$
+	указатели из вершин в ребра	$\Theta(m)$
	всего	$\Theta(m + n)$

Граница $\Theta(m + n)$ применима независимо от того, является ли граф связным или нет и имеет ли он параллельные ребра[1].

Решение тестового задания 7.3

Правильный ответ: (г). Простой способ хранения матрицы смежности — это двумерный $n \times n$-массив бит. Данный массив использует $\Theta(n^2)$ пространства, хотя и с небольшой скрытой константой. В случае плотного графа, в котором число ребер само по себе близко к квадратичному по n, матрица смежности требует пространства, близкого к линейному по размеру графа. Однако в случае разреженных графов, в которых число ребер

[1] Если граф является связным, то $m \geq n - 1$ (по тестовому заданию 7.1), и мы могли бы написать $\Theta(m)$ вместо $\Theta(m + n)$.

ближе к линейному по n, представление в виде матрицы смежности является весьма расточительным[1].

ВЫВОДЫ

- Граф является представлением парных связей между объектами, таких как дружба в социальной сети, гиперссылки между веб-страницами или зависимости между задачами.
- Граф состоит из множества вершин и множества ребер. Ребра являются неупорядоченными в неориентированных графах и упорядоченными в ориентированных графах.
- Граф является разреженным, если число ребер m является близким к линейному по числу вершин n, и плотным, если m является близким к квадратичному по n.
- Представление графа в виде списков смежности поддерживает массивы вершин и ребер, перекрестно ссылающиеся друг на друга естественным образом, и требует линейного пространства в общем числе вершин и ребер.
- Представление графа в виде матрицы смежности поддерживает один бит на пару вершин, для того чтобы отслеживать, какие ребра присутствуют, и требует квадратичного пространства по числу вершин.
- Представление в виде списков смежности является предпочтительным для разреженных графов и для приложений, которые предусматривают разведывание графов.

[1] Это расточительство может быть сокращено за счет трюков по хранению и манипулированию разреженными матрицами, то есть матрицами с большим числом нулей. Например, язык Matlab и пакет SciPy языка Python поддерживают разреженные матричные представления.

Задачи на закрепление материала

Задача 7.1. Пусть $G = (V, E)$ равно неориентированному графу. Под степенью вершины $v \in V$ мы подразумеваем число ребер в E, которые инцидентны вершине v (то есть которые имеют v в качестве конечной точки). Для каждого из следующих условий на графе G выполняется ли условие только плотными графами, только разреженными графами или же как некоторыми разреженными, так и некоторыми плотными графами? Как обычно, $n = |V|$ обозначает число вершин. Допустим, что n является большим (скажем, не менее 10 000).

а) минимум одна вершина G имеет степень не более 10.

б) каждая вершина G имеет степень не более 10.

в) хотя бы одна вершина G имеет степень $n - 1$.

г) каждая вершина G имеет степень $n - 1$.

Задача 7.2. Рассмотрим неориентированный граф $G = (V, E)$, представленный в виде матрицы смежности. С учетом вершины $v \in V$ сколько операций требуется для идентификации ребер, инцидентных вершине v? (Пусть k равно числу таких ребер. Как обычно, n и m обозначают число вершин и ребер соответственно.)

а) $\Theta(1)$.

б) $\Theta(k)$.

в) $\Theta(n)$.

г) $\Theta(m)$.

Задача 7.3. Рассмотрим ориентированный граф $G = (V, E)$, представленный списками смежности, причем каждая вершина хранит массив своих исходящих (но не входящих) ребер. С учетом вершины $v \in V$ сколько операций требуется для идентификации входящих ребер вершины v? (Пусть k равно числу таких ребер. Как обычно, n и m обозначают число вершин и ребер соответственно.)

а) $\Theta(1)$.

б) $\Theta(k)$.

в) $\Theta(n)$.

г) $\Theta(m)$.

8

Поиск в графе и его применения

Эта глава посвящена фундаментальным примитивам для графового поиска и их применениям. Один из интереснейших аспектов этого материала заключается в том, что все алгоритмы, которые мы рассмотрим, являются невероятно быстрыми (линейно-временными с небольшими константами), и понять, почему они работают, может быть довольно непросто. Кульминацией этой главы является вычисление сильно связных компонент ориентированного графа только с двумя проходами поиска в глубину (раздел 8.6) — оно наглядно иллюстрирует то, как быстрые алгоритмы нередко требуют глубокого понимания структуры задачи.

Мы начнем с обзорного материала (раздел 8.1), который охватывает несколько причин, почему вас должен заботить поиск в графе, общую стратегию графового поиска без выполнения какой-либо избыточной работы и высокоуровневое введение в две наиважнейшие стратегии поиска: поиск в ширину (breadth-first search, BFS) и поиск в глубину (depth-first search, DFS). Разделы 8.2 и 8.3 более подробно описывают поиск в ширину, включая применения для вычисления кратчайших путей и связных компонент неориентированного графа. Разделы 8.4 и 8.5 разъясняют поиск в глубину и его использование для вычисления топологического упорядочивания ориентированного ациклического графа (равным образом, для построения последовательности операций при соблюдении ограничений по предшествованию). В разделе 8.6 используется поиск в глубину для вычисления сильно связных компонент ориентированного графа за линейное время. В разделе 8.7 дается объяснение того, как этот быстрый графовый примитив может быть использован для разведывания структуры Всемирной паутины.

8.1. Краткий обзор

В этом разделе дано общее представление алгоритмов графового поиска и их приложений.

8.1.1. Некоторые приложения

Зачем нужно выполнять поиск в графе или выяснять, содержит ли граф путь из точки A в точку B? Вот лишь несколько причин из огромного их множества.

Проверка связности. В физической сети, такой как дорожная сеть или сеть компьютеров, важной проверкой работоспособности является то, что вы можете попасть куда угодно из любого места. То есть для каждого варианта точки *A* и точки *B* в сети должен быть путь от первой к последней.

Связность также может быть важна в абстрактных (нефизических) графах, представляющих попарные связи между объектами. Одна такая сеть, с которой интересно поиграть, — это сеть кинофильмов, где вершины соответствуют киноактерам, и оба связаны неориентированным ребром всякий раз, когда они появляются в одной и той же кинокартине[1]. Например, сколько степеней разделения[2] существует между разными актерами?

Самым известным статистическим показателем этого типа является число Бейкона, которое представляет собой минимальное число переходов по фильмовой сети, необходимое для того, чтобы добраться до вездесущего актера Кевина Бейкона[3]. Так, Кевин Бейкон сам лично имеет число Бейкона, равное 0; каждый актер, который появился с ним в фильме, имеет число Бейкона, равное 1; каждый актер, который появился с актером — обладателем числа Бейкона, равного 1, но кто не является самим Кевином Бейконом, имеет число Бейкона, равное 2, и так далее. Например, Джон Хэмм — наиболее известный по роли Дона Дрейпера из телевизионного сериала *Mad Men* (*«Безумцы»*) — имеет число Бейкона, равное 2. Хэмм никогда не появлялся в фильме с Бейконом, но у него была эпизодическая роль в пьесе *A Single Man* (*«Одинокий мужчина»*), написанной для Колина Ферта, а Ферт и Бейкон снялись в фильме Атома Эгояна *Where the Truth Lies* (*«Где скрывается правда»*) (рис. 8.1)[4].

---

[1] https://oracleofbacon.org/.

[2] Степень разделения (degree of separation) — число уровней связей, разделяющих два произвольных узла сети. — *Примеч. пер.*

[3] Число Бейкона — это импровизация на тему более старой концепции *числа Эрдёша*, названного в честь известного математика Пола Эрдёша, которое измеряет число степеней разделения от Эрдёша в графе соавторства (где вершины являются исследователями, и между каждой парой исследователей, являющихся соавторами статьи, существует ребро).

[4] Между Бейконом и Хэммом также существует ряд других путей с двумя переходами.

Рис. 8.1. Фрагмент видеосети, показывающий, что число Бейкона у Джона Хэмма не превышает 2

Кратчайшие пути. Число Бейкона относится к *кратчайшему* пути между двумя вершинами сети фильмов, который означает путь, использующий наименьшее число ребер. В разделе 8.2 мы увидим, что стратегия графового поиска, именуемая поиском в ширину, естественным образом вычисляет кратчайший путь. Целый ряд других задач сводится к расчету кратчайшего пути, где значение слова «короткий» зависит от приложения (минимизация времени для маршрутов движения или денег на авиабилеты и так далее). Алгоритм кратчайшего пути Дейкстры — предмет главы 9 — основан на поиске в ширину для решения более общих задач нахождения кратчайшего пути.

Планирование. Путь в графе не обязательно должен представлять физический путь по физической сети. Более абстрактно, путь — это последовательность решений, переводящих вас из одного состояния в другое. Алгоритмы графового поиска могут применяться к таким абстрактным графам для вычисления плана достижения целевого состояния из исходного состояния. Представьте, например, что вы хотите применить алгоритм для того, чтобы решить головоломку судоку. Подумайте о графе, вершины которого соответствуют частично завершенным головоломкам судоку (где несколько клеток из 81 пусты, но правила судоку не нарушены), и ориентированные ребра соответствуют заполнению одной новой ячейки головоломки (при условии соблюдения правил судоку). Задача вычисления решения головоломки — это именно задача вычисления ориентированного пути из вершины, соответствующей исходному состоянию головоломки, в вершину, соответствующую завершенной головоломке[1]. Еще один пример: использо-

[1] Поскольку этот граф слишком велик для того, чтобы записать его в явном виде, в практические решатели головоломки судоку встраиваются некоторые дополнительные идеи.

вание роботизированной руки для захвата кофейной кружки — это, по сути, задача планирования. В связном графе вершины соответствуют возможным конфигурациям руки, а ребра соответствуют небольшим и реализуемым изменениям данной конфигурации.

Связные компоненты. Мы также увидим алгоритмы, основывающиеся на графовом поиске для вычисления связных компонент (частей) графа. Определить и вычислить связные компоненты неориентированного графа относительно просто (раздел 8.3). А вот в случае ориентированных графов даже определение того, что должно означать понятие «связная компонента», имеет свои небольшие тонкости. В разделе 8.6 дается это определение и показывается, как применять поиск в глубину (раздел 8.4), для того чтобы вычислять их эффективно. Мы также увидим применения поиска в глубину для построения последовательности операций (раздел 8.5) и для разведывания структуры веб-графа (раздел 8.7).

8.1.2. Бесплатные графовые примитивы

Примеры в разделе 8.1.1 демонстрируют, что поиск в графе является фундаментальным и широко применимым базисным элементом. Хочу порадовать, что в этой главе все наши алгоритмы будут работать сверхбыстро, всего за время $O(m + n)$, где m и n обозначают число ребер и вершин графа[1]. Оно больше времени, необходимого для чтения входных данных, лишь на постоянный множитель![2] Мы заключаем, что эти алгоритмы являются бесплатными примитивами — всякий раз, когда у вас есть графовые данные, вы должны без колебаний применять любой из этих примитивов для сбора информации о том, как он выглядит[3].

---

[1] Кроме того, константы, скрытые в форме записи O-большое, достаточно малы.

[2] В задачах графового поиска и связности нет оснований ожидать, что входной граф является связным. В несвязном случае, где m может быть намного меньше n, размер графа равен $\Theta(m + n)$, но не обязательно $\Theta(m)$.

[3] Можем ли мы добиться лучшего? Нет, вплоть до скрытого постоянного множителя: каждый правильный алгоритм в некоторых случаях должен, по крайней мере, считывать все входные данные целиком.

БЕСПЛАТНЫЕ ПРИМИТИВЫ

Алгоритм с линейным или почти линейным временем выполнения можно рассматривать как примитив, который мы можем использовать по существу «бесплатно», потому что объем используемых вычислений едва превышает объем, необходимый только для чтения входных данных. Когда у вас есть примитив, имеющий отношение к вашей задаче, который является таким очевидно быстрым, почему бы им не воспользоваться? Например, вы всегда можете вычислить связные компоненты графовых данных на этапе предварительной обработки, даже если вы не уверены, как это поможет в дальнейшем. Одна из целей этой книжной серии — снабдить ваш алгоритмический инструментарий как можно большим числом бесплатных примитивов, готовых к применению в любой момент.

8.1.3. Обобщенный графовый поиск

Суть алгоритма поиска в графе заключается в решении следующей задачи.

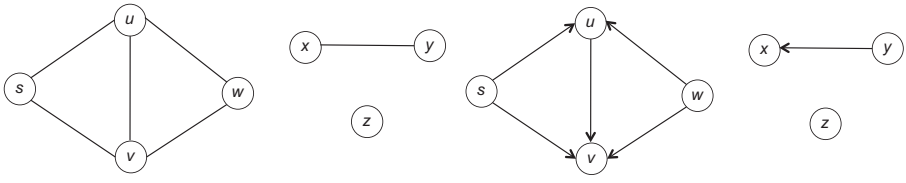
ЗАДАЧА: ПОИСК В ГРАФЕ

Вход: неориентированный или ориентированный граф $G = (V, E)$ и стартовая вершина $s \in V$.

Цель: идентифицировать вершины множества V, достижимые из s в графе G.

Под достижимой вершиной v мы имеем в виду, что существует последовательность ребер в G, которая переносит из s в v. Если G является ориентированным графом, то все ребра пути должны быть пройдены в прямом (исходящем) направлении. Например, на рис. 8.2, *а* множество достижимых вершин (из s) равно $\{s, u, v, w\}$. В ориентированной версии графа на рис. 8.2, *б* ориентированный путь из s в w отсутствует, и через ориентированный путь из s достижимы только вершины s, u и v[1].

[1] В общем случае большинство алгоритмов и аргументов в этой главе одинаково хорошо применимы к неориентированным и ориентированным графам. Большим исключением является вычисление связных компонент, которое является более сложной задачей в ориентированных графах, чем в неориентированных.



а) Неориентированный граф *б*) Ориентированный вариант

Рис. 8.2. В случае *а* множество вершин, достижимых из *s*, равно {*s*, *u*, *v*, *w*}. В случае *б* это множество равно {*s*, *u*, *v*}

Две стратегии графового поиска, на которых мы сосредоточимся, поиск в ширину и поиск в глубину представляют собой разные способы создания обобщенного алгоритма графового поиска. Обобщенный алгоритм систематически находит все достижимые вершины, стараясь не разведывать что-либо дважды. С каждой вершиной он поддерживает дополнительную переменную, которая отслеживает, была ли она уже разведана, устанавливая флаг при первом достижении этой вершины. Обязанностью главного цикла является достижение новой неразведанной вершины в каждой итерации.

ОБОБЩЕННЫЙ ПОИСК (GENERICSEARCH)

Вход: граф $G = (V, E)$ и вершина $s \in V$.

Постусловие: вершина достижима из s тогда и только тогда, когда она помечена как «разведанная».

пометить вершину s как разведанную, все остальные вершины как неразведанные

while существует ребро $(v, w) \in E$ с разведанной v и неразведанной w **do**

выбрать несколько таких ребер (v, w) `// конкретизировано // неполно`

пометить w как разведанную

Данный алгоритм по существу одинаков как для ориентированных, так и для неориентированных графов. В ориентированном случае ребро (v, w), выбранное в итерации цикла `while`, должно быть направлено из разведанной вершины v в неразведанную вершину w.

О ПСЕВДОКОДЕ

Эта серия книг объясняет алгоритмы с использованием комбинации высокоуровневого псевдокода и русского языка (как показано выше). Я исхожу из того, что у вас есть навыки трансляции таких высокоуровневых (общих) описаний в рабочий код на вашем любимом языке программирования. Несколько других книг и ресурсов в интернете предлагают конкретные реализации различных алгоритмов на определенных языках программирования.

Во-первых, преимущество акцентирования высокоуровневых описаний над реализациями, специфичными для конкретного языка, заключается в гибкости: хотя я предполагаю некоторую осведомленность в *каком-то* языке программирования, меня не интересует, какой именно это язык.

Во-вторых, этот подход способствует пониманию алгоритмов на глубоком и концептуальном уровне, не обремененном деталями низкого уровня. Опытные программисты и специалисты в области информатики обычно мыслят и обмениваются информацией об алгоритмах на столь же высоком уровне.

Тем не менее, ничто не может заменить подробного понимания алгоритма, которое вытекает из разработки своей собственной рабочей реализации. Настоятельно рекомендую вам осуществить столько алгоритмов из этой книги, сколько времени будет находиться в вашем распоряжении. (Это будет отличным поводом для того, чтобы познакомиться с новым языком программирования!)

В целях получения указаний обратитесь к разделу задач по программированию в конце главы и вспомогательным тестовым примерам.

Например, на графе рис. 8.2, *а* первоначально помечено как разведанное только наше основное место базирования s. На первой итерации цикла `while` условию цикла удовлетворяют два ребра: (s, u) и (s, v). Алгоритм `GenericSearch`

выбирает одно из этих ребер — скажем, (s, u) — и отмечает вершину u как разведанную. Во второй итерации цикла снова есть два варианта: (s, v) и (u, w). Алгоритм может выбрать (u, w), в каковом случае w помечается как разведанная. Еще одна итерация (после выбора (s, v) или (w, v)) помечает v как разведанную. На этом этапе ребро (x, y) имеет две неразведанные конечные точки, а другие ребра имеют две разведанные конечные точки, и алгоритм останавливается. Как следует ожидать, вершины, помеченные как разведанные — s, u, v и w — являются именно теми вершинами, которые достижимы из s.

Этот обобщенный алгоритм графового поиска конкретизирован неполно, так как в итерации цикла `while` приемлемыми для выбора могут быть несколько ребер (v, w). Поиск в ширину и поиск в глубину соответствуют двум конкретным решениям о том, какое ребро разведывать дальше. Независимо от того, как этот выбор делается, алгоритм `GenericSearch` гарантированно будет правильным (как в неориентированных, так и в ориентированных графах).

Утверждение 8.1. (Правильность обобщенного графового поиска.) *По завершении алгоритма* `GenericSearch` *вершина* $v \in V$ *помечается как разведанная тогда и только тогда, когда существует путь из* s *в* v *в графе* G.

Раздел 8.1.5 предоставляет формальное доказательство утверждения 8.1; можете спокойно его пропустить, если данное утверждение выглядит интуитивно очевидным.

О ЛЕММАХ, ТЕОРЕМАХ И ПРОЧЕМ

В математической записи самые важные технические высказывания имеют статус *теорем*. *Лемма* — это техническое высказывание, которое помогает с доказательством теоремы (во многом, как подпрограмма помогает с реализацией более крупной программы). *Следствие* — это высказывание, которое непосредственно вытекает из уже доказанного результата, например, частный случай теоремы. Мы используем термин *утверждение* (в иных источниках оно именуется пропозицией или высказыванием) для автономных технических высказываний, которые сами по себе не особенно важны.

А как насчет времени работы алгоритма `GenericSearch`? Алгоритм разведывает каждое ребро не более одного раза — после того как ребро (v, w) было разведано в первый раз, обе вершины, v и w, помечаются как разведанные, и ребро не будет рассматриваться снова. Это наводит на мысль о том, что существует возможность реализовать алгоритм, работающий за линейное время, в случае если на каждой итерации цикла `while` мы сможем быстро идентифицировать приемлемое ребро (v, w). Мы подробно проанализируем, как это работает для поиска в ширину и поиска в глубину, в разделах 8.2 и 8.4 соответственно.

8.1.4. Поиск в ширину и в глубину

Каждая итерация алгоритма `GenericSearch` выбирает ребро, которое находится на границе разведываемой части графа, с одной стороны которой конечная точка разведана, а с другой не разведана (рис. 8.3). Таких ребер может быть много, и чтобы полностью определить алгоритм, нам нужен метод выбора одного из них. Мы сосредоточимся на двух наиболее важных стратегиях: поиске в ширину и поиске в глубину. Оба являются отличными способами разведывания графа, и каждый из них имеет свое собственное множество применений.

Поиск в ширину (breadth-first search, BFS). Высокоуровневая идея *поиска в ширину* — или *BFS* для тех, кто в теме, — заключается в том, чтобы тщательно разведывать вершины графа «слоями». Слой 0 состоит только из стартовой вершины s. Слой 1 содержит вершины, соседствующие с s, имея в виду вершины v такие, что (s, v) является ребром графа (направленным из s в v, в случае если G является ориентированным). Слой 2 состоит из соседей вершин слоя 1, которые уже не принадлежат слою 0 или 1, и так далее. В разделах 8.2 и 8.3 мы увидим:

- как реализовывать поиск в ширину с линейным временем, используя структуру данных, именуемую очередью (первым вошел/первым вышел);
- как использовать поиск в ширину для вычисления (за линейное время) длины кратчайшего пути между одной вершиной и всеми другими вершинами, при этом вершины слоя i являются именно вершинами на расстоянии i от s;

- как использовать поиск в ширину для вычисления (за линейное время) связных компонент неориентированного графа.

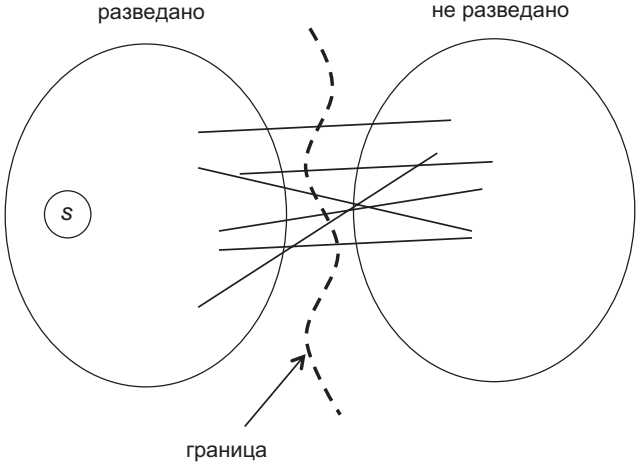


Рис. 8.3. Каждая итерация алгоритма GenericSearch выбирает ребро на границе, при этом одна конечная точка разведана, а другая нет

Поиск в глубину (depth-first search, DFS). *Поиск в глубину* (*DFS*), возможно, является еще более важным. Поиск в глубину использует более агрессивную стратегию разведывания графа, по своему духу очень хорошо соответствуя тому, как вы можете разведывать лабиринт, заходя настолько глубоко, насколько это возможно, и отступая только в случае крайней необходимости. В разделах 8.4–8.7 мы увидим:

- как реализовывать поиск в глубину с линейным временем, используя рекурсию или явно заданную структуру данных, именуемую стеком (последним вошел / первым вышел);
- как использовать поиск в глубину для вычисления (за линейное время) топологического упорядочивания вершин ациклического графа, полезного примитива для задач построения последовательности операций;
- как использовать поиск в глубину для вычисления (за линейное время) сильно связных компонент ориентированного графа с применениями для понимания структуры Всемирной паутины.

8.1.5. Правильность алгоритма GenericSearch

Сейчас мы докажем утверждение 8.1, которое гласит, что по завершении алгоритма `GenericSearch` с входным графом $G = (V, E)$ и стартовой вершиной $s \in V$ вершина $v \in V$ помечается как разведанная тогда и только тогда, когда существует путь из s в v в графе G. Как обычно, если G является ориентированным графом, то путь $s \rightsquigarrow v$ также должен быть ориентированным, при этом все ребра были пройдены в прямом направлении.

Направление утверждения со словосочетанием «тогда и только тогда» должно быть интуитивно понятным: единственный способ, которым алгоритм `GenericSearch` обнаруживает новые вершины, состоит в следовании по путям из s[1].

Направление со словом «когда» утверждает менее очевидный факт, что алгоритм `GenericSearch` ничего не пропускает — он находит каждую вершину, которую мог бы обнаружить. В случае этого направления мы будем использовать доказательство от противного. Напомним, что в этом типе доказательства вы исходите из *противоположного* допущения относительно того, что вы хотите доказать, а затем строите на этом допущении последовательность логически правильных шагов, которая завершается явно ложным формальным утверждением. Такое противоречие подразумевает, что данное допущение не может быть истинным, что и доказывает искомое утверждение.

Итак, предположим, что существует путь из s в v в графе G, но алгоритм `GenericSearch` каким-то образом пропускает его и завершается вершиной v, помеченной как неразведанная. Пусть $S \subseteq V$ обозначает вершины графа G, помеченные как разведанные алгоритмом. Вершина s принадлежит S (по первой строке алгоритма), и вершина v — нет (по принятому допущению). Поскольку путь $s \rightsquigarrow v$ проходит из вершины внутри S в вершину вне S, то минимум одно ребро e пути имеет одну конечную точку u в S, и другую w вне S (при этом e ориентировано из u в w в случае, если G является ориентированным) (рис. 8.4). Но это, друзья мои, невозможно: ребро e будет приемлемым для отбора в цикле `while` алгоритма `GenericSearch`, и алгоритм не сдался бы

[1] Если бы мы хотели быть более педантичными, то доказали бы это направление по индукции на числе итераций цикла.

и разведал по крайней мере еще одну вершину! Нет никакой причины, почему алгоритм `GenericSearch` мог бы остановиться на этом этапе, и поэтому мы достигли противоречия. Это противоречие завершает доказательство утверждения 8.1. *Ч. т. д.*[1]

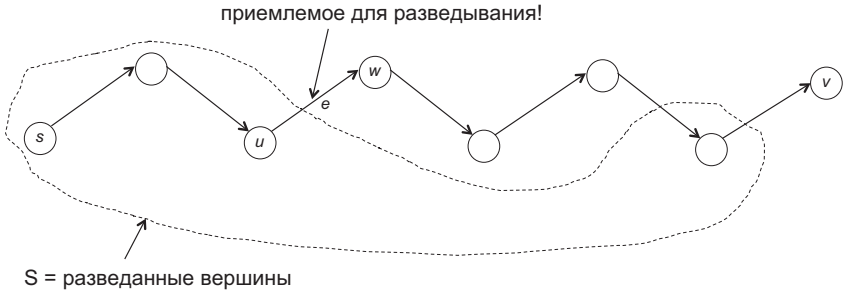


Рис. 8.4. Доказательство утверждения 8.1. До тех пор пока алгоритм GenericSearch еще не обнаружил все достижимые вершины, существует приемлемое ребро, вдоль которого он может разведывать дальше

8.2. Поиск в ширину и кратчайшие пути

Давайте углубимся в нашу первую конкретную стратегию графового поиска — *поиска в ширину*.

8.2.1. Высокоуровневая идея

Поиск в ширину разведывает вершины графа слоями, в порядке увеличения расстояния от стартовой вершины. Слой 0 содержит стартовую вершину *s* и больше ничего. Слой 1 представляет собой множество вершин, которые находятся на расстоянии одного перехода из *s*, то есть содержит соседей *s*. Это вершины, которые при поиске в ширину разведываются сразу после *s*. Например, в графе на рис. 8.5 *a* и *b* являются соседями *s* и образуют слой 1. В общем случае вершинами в слое *i* являются те, которые соседствуют

[1] Ч. т. д. — это аббревиатура выражения «что и требовалось доказать» от латинского *quod erat demonstrandum* (*Q.E.D.*). В математической записи она используется в конце доказательства, для того чтобы отметить его завершение.

с вершиной в слое $i - 1$ и которые уже не принадлежат одному из слоев 0, 1, 2, ..., $i - 1$. Поиск в ширину разведывает все вершины слоя i сразу после завершения разведывания вершин слоя ($i - 1$). (Вершины, недостижимые из s, не принадлежат ни одному слою.) Например, на рис. 8.5 вершинами слоя 2 являются c и d, так как они соседствуют с вершинами слоя 1, но сами они не принадлежат слою 0 или 1. (Вершина s также является соседом вершины слоя 1, но она уже принадлежит слою 0.) Последний слой графа на рис. 8.5 содержит только вершину e.

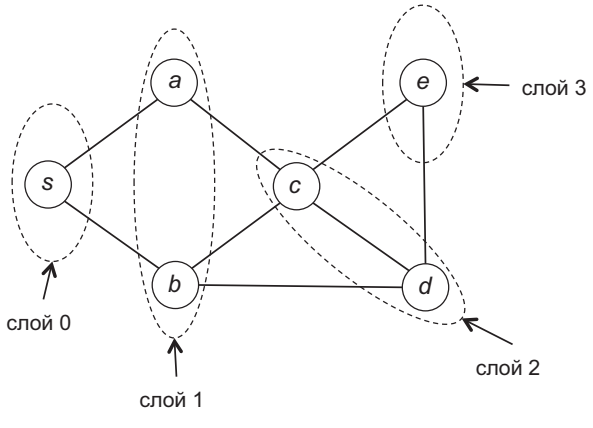


Рис. 8.5. Поиск в ширину обнаруживает вершины слоями. Вершины слоя i являются соседями вершин слоя ($i - 1$), которые не появляются ни в одном более раннем слое

ТЕСТОВОЕ ЗАДАНИЕ 8.1

Рассмотрим неориентированный граф с $n \geq 2$ вершинами. Каково соответственно минимальное и максимальное число разных слоев, которые этот граф мог бы иметь?

а) 1 и $n - 1$

б) 2 и $n - 1$

в) 1 и n

г) 2 и n

(Решение и пояснение см. в разделе 8.2.6.)

8.2.2. Псевдокод для алгоритма BFS

Реализация линейно-временного поиска в ширину требует простой структуры данных с дисциплиной доступа «первым вошел/первым вышел», именуемой *очередью*. Поиск в ширину использует очередь для того, чтобы отслеживать, какие вершины разведывать дальше. Если вы не знакомы с очередями, то сейчас самое время прочитать о них в любом вводном курсе по программированию (или в «Википедии»). Суть очереди состоит в том, что она представляет собой структуру данных, служащую для поддержания списка объектов, и вы можете удалять элементы из ее начала или добавить элементы в ее конец за постоянное время[1].

BFS

Вход: граф $G = (V, E)$, представленный в виде списков смежности, и вершина $s \in V$.

Постусловие: вершина достижима из s тогда и только тогда, когда она помечена как «разведанная».

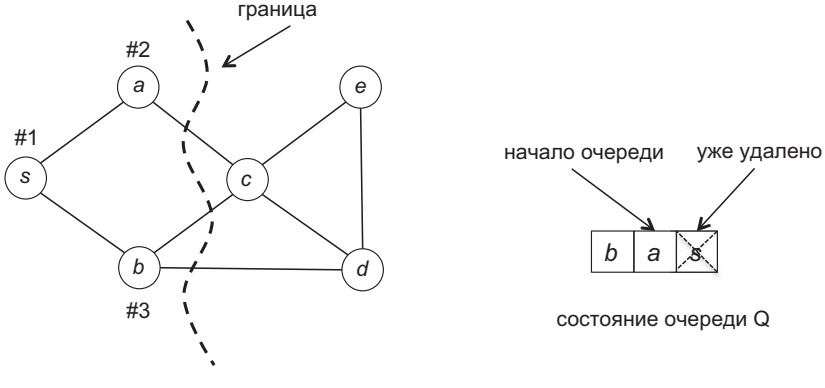
```
1) пометить s как разведанную вершину, все остальные как не-
   разведанные
2) Q := очередь, инициализированная вершиной s
3) while Q не является пустой do
4)     удалить вершину из начала Q, назвать ее v
5)     for каждое ребро (v, w) в списке смежности v do
6)         if w не разведана then
7)             пометить w как разведанную
8)             добавить w в конец Q
```

[1] Возможно, вам никогда не понадобится реализовывать очередь с нуля, так как она встроена в большинство современных языков программирования. Но если такая необходимость возникнет, то вы можете использовать двусвязный список. Либо если вы заранее знаете максимальное число объектов, которые вы, возможно, захотите хранить (то есть $|V|$ в случае BFS), то вы можете обойтись массивом фиксированной длины и парой индексов (которые отслеживают начало и конец очереди).

Каждая итерация цикла `while` разведывает одну новую вершину. В строке 5 алгоритм `BFS` перебирает все ребра, инцидентные вершине v (если G является неориентированным) либо все ребра, исходящие из v (если G является ориентированным)[1]. Неразведанные соседи вершины v добавляются в конец очереди и помечаются как разведанные; со временем они будут обработаны в последующих итерациях алгоритма.

8.2.3. Пример

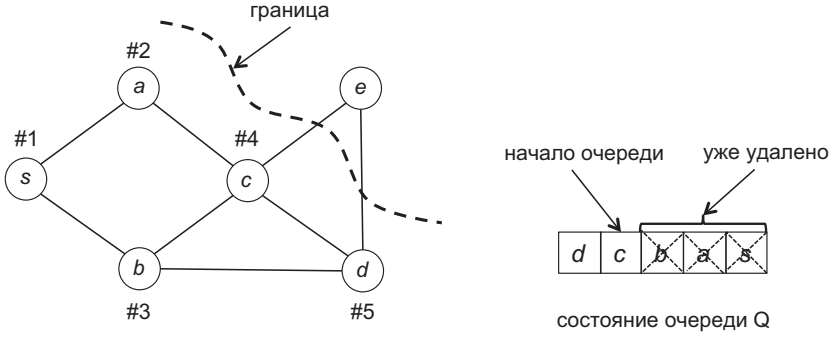
Давайте посмотрим, как наш псевдокод работает для графа на рис. 8.5, пронумеровывая вершины в порядке их вставки в очередь (равным образом, в порядке их разведывания). Стартовая вершина s всегда является первой разведываемой вершиной. Первая итерация цикла `while` извлекает s из очереди Q, а последующий цикл `for` проверяет ребра (s, a) и (s, b) в любом порядке, в котором эти ребра появляются в списке смежности s. Поскольку ни a, ни b не помечены как разведанные, оба вставляются в очередь. Предположим, что ребро (s, a) появилось первым, и поэтому a вставляется перед b. Текущее состояние графа и очереди теперь будет следующим:



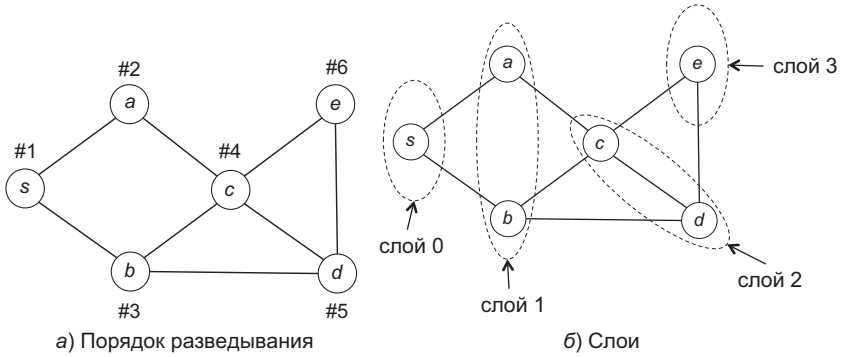
Следующая итерация цикла `while` извлекает вершину a из начала очереди и рассматривает ее инцидентные ребра (s, a) и (a, c). Она пропускает первое из двух после того, как перепроверит, что s уже помечена как разведанная, и до-

[1] Это именно тот шаг, на котором очень удобно иметь входной граф, представленный посредством списков смежности.

бавляет (ранее неразведанную) вершину *c* в конец очереди. Третья итерация извлекает вершину *b* из начала очереди и добавляет вершину *d* в конец (так как *s* и *c* уже помечены как разведанные, они пропускаются). Новая картина имеет следующий вид:



На четвертой итерации вершина *c* удаляется из начала очереди. Из всех ее соседей вершина *e* является единственной не встречавшейся ранее, и она будет добавлена в конец очереди. Последние две итерации извлекают *d*, и затем *e* из очереди и перепроверяют, что все их соседи уже разведаны. Очередь затем становится пустой, и алгоритм останавливается. Вершины разведывались в порядке расположения слоев, при этом вершины слоя *i* были разведаны сразу после вершин слоя (*i* – 1) (рис. 8.6).



а) Порядок разведывания *б*) Слои

Рис. 8.6. В поиске в ширину вершины слоя *i* разведываются сразу после вершин слоя (*i* – 1)

8.2.4. Правильность и время выполнения

Поиск в ширину обнаруживает все вершины, достижимые из стартовой вершины, и выполняется за линейное время. Более уточненная граница времени выполнения в приведенной ниже теореме 8.2 (*в*) пригодится для нашего линейно-временного алгоритма вычисления связных компонент (описанного в разделе 8.3).

Теорема 8.2. (Свойства алгоритма `BFS`.) *Для каждого неориентированного либо ориентированного графа $G = (V, E)$, представленного в виде списков смежности, и для каждой стартовой вершины $s \in V$:*

а) по завершении алгоритма BFS вершина $v \in V$ помечается как разведанная тогда и только тогда, когда существует путь из s в v в графе G;

б) время работы алгоритма BFS равно $O(m + n)$, где $m = |E|$ и $n = |V|$;

в) время работы строк 2–8 алгоритма BFS составляет

$$O(m_s + n_s),$$

где m_s и n_s обозначают соответственно число ребер и вершин, достижимых из s в графе G.

Доказательство: часть (*а*) следует из гарантии утверждения 8.1 для алгоритма обобщенного графового поиска `GenericSearch`, частным случаем которого является алгоритм `BFS`[1]. Часть (*б*) следует из части (*в*), так как совокупное

[1] Формально алгоритм `BFS` эквивалентен версии алгоритма `GenericSearch`, где в каждой итерации цикла `while` последнего алгоритм выбирает приемлемое ребро (v, w), для которого v была обнаружена самой ранней, разрывая связи между приемлемыми ребрами вершины v в соответствии с их порядком в списке смежности этой вершины. Если это звучит слишком сложно, то, как вариант, вы можете проверить, что доказательство утверждения 8.1 дословно также соблюдается для поиска в ширину. В интуитивном плане поиск в ширину обнаруживает вершины, разведывая пути только из s; до тех пор пока он не разведает каждую вершину на пути, следующая вершина на данном пути по-прежнему будет находиться в очереди, ожидая дальнейшего разведывания.

время работы алгоритма `BFS` состоит лишь из времени работы строк 2–8 плюс времени $O(n)$, необходимого для инициализации в строке 1.

Мы можем доказать часть (*в*), непосредственно обратившись к псевдокоду. Инициализация в строке 2 занимает время $O(1)$. В главном цикле `while` алгоритм встречает только n_s вершин, достижимых из s. Поскольку ни одна вершина не разведывается дважды, каждая такая вершина добавляется в конец очереди и удаляется из начала очереди ровно один раз. Каждая из этих операций занимает время $O(1)$ — в этом весь смысл очереди как структуры данных с дисциплиной доступа «первым вошел/первым вышел», и поэтому суммарное количество времени, затрачиваемое в строках 3–4 и 7–8, составляет $O(n_s)$. Каждое из m_s ребер (v, w), достижимых из s, обрабатывается в строке 5 не более двух раз — один раз, когда разведывается v, и один раз, когда разведывается w[1]. Таким образом, суммарное время, затрачиваемое в строках 5–6, составляет $O(m_s)$, а совокупное время работы для строк 2–8 равно $O(m_s + n_s)$. *Ч. т. д.*

8.2.5. Кратчайший путь

Свойства в теореме 8.2 не являются уникальными для поиска в ширину — например, они также соблюдаются для поиска в глубину. Уникальность алгоритма `BFS` состоит в том, что с помощью пары дополнительных строк исходного кода он эффективно вычисляет расстояния кратчайшего пути.

Постановка задачи

В графе G мы используем обозначение $dist(v, w)$ для наименьшего числа ребер в пути из v в w (или $+\infty$, если G не содержит пути из v в w)[2].

---

[1] Если G является ориентированным графом, то при разведывании его хвостовой вершины каждое ребро обрабатывается не более одного раза.

[2] Как обычно, если G является ориентированным, то все ребра пути должны быть пройдены в прямом направлении.

ЗАДАЧА: КРАТЧАЙШИЙ ПУТЬ (ЕДИНИЧНЫЕ ДЛИНЫ РЕБЕР)

Вход: неориентированный или ориентированный граф $G = (V, E)$ и начальная вершина $s \in V$.

Выход: $dist\ (s, v)$ для каждой вершины $v \in V$[1].

Например, если G — это фильмовая сеть и s — вершина, соответствующая Кевину Бейкону, то задача вычисления кратчайших путей как раз и является задачей вычисления числа Бейкона каждого участника (раздел 8.1.1). Базовая задача поиска в графе (раздел 8.1.3) соответствует частному случаю идентификации всех вершин v с расстоянием $dist\ (s, v) \neq +\infty$.

Псевдокод

Для того чтобы вычислить кратчайшие пути, мы добавляем в базовый алгоритм `BFS` две строки кода (приведенные ниже строки 2 и 9); они увеличивают время работы алгоритма на небольшой постоянный множитель. Первая строка инициализирует предварительные оценки расстояний кратчайшего пути вершин — 0 для s и $+\infty$ для других вершин, которые могут быть недостижимы даже из s. Вторая строка выполняется всякий раз, когда вершина w обнаруживается в первый раз, и вычисляет окончательное расстояние кратчайшего пути вершины w как на одно больше, чем у вершины v, которая вызвала обнаружение вершины w.

ДОПОЛНЕННЫЙ ПОИСК В ШИРИНУ (AUGMENTED-BFS)

Вход: граф $G = (V, E)$, представленный в виде списков смежности, и вершина $s \in V$.

[1] Словосочетание «единичные длины ребер» в постановке задачи относится к допущению, что каждое ребро G вносит 1 в длину пути. Глава 9 обобщает алгоритм `BFS` для вычисления кратчайших путей в графах, в которых каждое ребро имеет свою собственную неотрицательную длину.

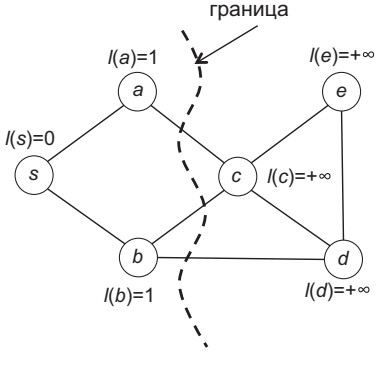
Постусловие: для каждой вершины $v \in V$ значение $l(v)$ равно истинному расстоянию кратчайшего пути $dist(s, v)$.

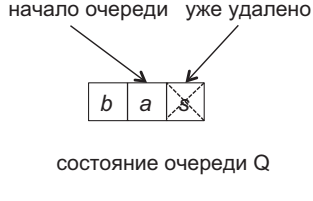
---

1) пометить s как разведанную вершину, все остальные как неразведанные
2) $l(s) := 0$, $l(v) := +\infty$ для каждой $v \neq s$
3) Q := очередь, инициализированная вершиной s
4) **while** Q не является пустой **do**
5) удалить вершину из начала Q, назвать ее v
6) **for** каждое ребро (v, w) в списке смежности вершины v **do**
7) **if** w не разведана **then**
8) пометить w как разведанную
9) $l(w) := l(v) + 1$
10) добавить w в конец Q

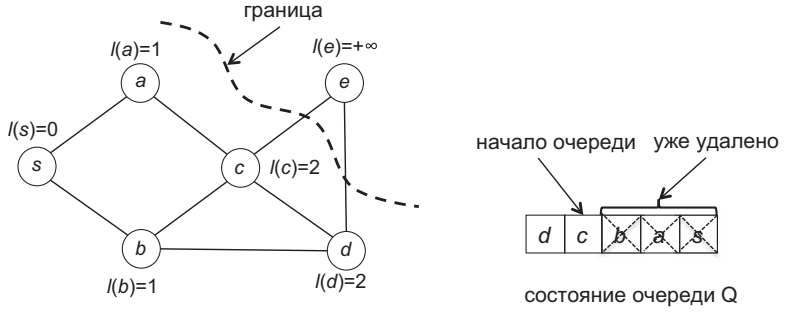
Пример и анализ

В нашем примере выполнения (рис. 8.6) первая итерация цикла `while` обнаруживает вершины a и b. Поскольку s инициировала их обнаружение и $l(s) = 0$, алгоритм модифицирует $l(a)$ и $l(b)$, меняя их значения с $+\infty$ на 1:





Вторая итерация цикла `while` обрабатывает вершину a, что приводит к обнаружению c. Алгоритм модифицирует $l(c)$, меняя ее значение с $+\infty$ на $l(a) + 1$, то есть теперь значение длины равно 2. Схожим образом на третьей итерации $l(d)$ имеет значение $l(b) + 1$, которое также равно 2:



Четвертая итерация обнаруживает финальную вершину e через вершину c и устанавливает $l(e)$ равной значению $l(c) + 1$, которое равно 3. На этом этапе для каждой вершины v $l(v)$ равно истинному расстоянию кратчайшего пути $dist(s, v)$, которое также равно номеру слоя, содержащего v (рис. 8.6). Эти свойства соблюдаются в общем случае, а не только для данного примера.

Теорема 8.3. (Свойства алгоритма `Augmented-BFS`.) *Для каждого неориентированного или ориентированного графа $G = (V, E)$, представленного в виде списков смежности, и для каждой стартовой вершины $s \in V$:*

а) по завершении алгоритма `Augmented-BFS` *для каждой вершины $v \in V$ значение $l(v)$ равно длине $dist(s, v)$ кратчайшего пути из s в v в графе G (либо $+\infty$, если такой путь не существует);*

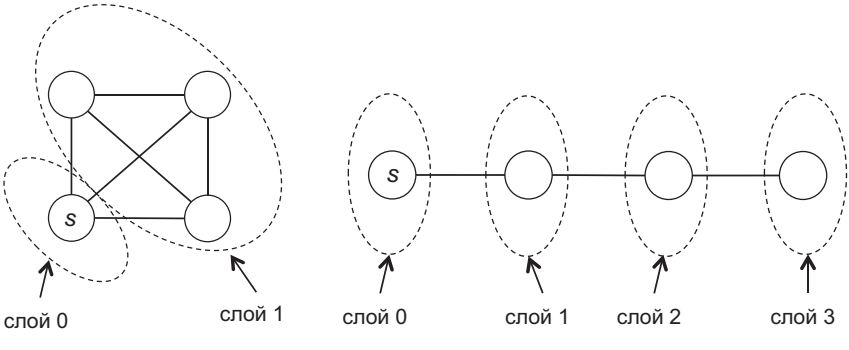
б) время работы алгоритма `Augmented-BFS` *равно $O(m + n)$, где $m = |E|$ и $n = |V|$.*

Поскольку асимптотическое время работы алгоритма `Augmented-BFS` является точно таким же, как и у алгоритма `BFS`, часть (*б*) теоремы 8.3 следует из гарантии времени работы последнего (теорема 8.2 (*б*)). Часть (*а*) вытекает

из двух наблюдений. Во-первых, вершины v, где $dist\ (s, v) = i$, являются как раз вершинами в i-м слое графа — вот почему мы определили слои именно так, а не иначе. Во-вторых, для каждой вершины w слоя i алгоритм `Augmented-BFS` в конечном счете устанавливает $l(w) = i$ (так как w обнаруживается посредством вершины v слоя $(i - 1)$, где $l(v) = i - 1$). Для вершин ни в одном из слоев, то есть не достижимых из s, как $dist(s, v)$, так и $l(v)$ равны $+\infty$[1].

8.2.6. Решение тестового задания 8.1

Правильный ответ: (г). Неориентированный граф с $n \geq 2$ вершинами имеет не менее двух слоев и не более n слоев. При $n \geq 2$ не может быть меньше двух слоев, потому что s является единственной вершиной в слое 0. Полные графы имеют только два слоя (рис. 8.7, *а*). Не может быть больше n слоев, так как слои не пересекаются и содержат по крайней мере одну вершину каждый. Графы путей имеют n слоев (рис. 8.7, *б*).



а) Полный граф *б*) Путь графа

Рис. 8.7. *n*-вершинный граф может иметь от двух до *n* разных слоев

[1] Если вы жаждете более строгого доказательства, то выполните его — в уединении домашней обстановки — по индукции на числе итераций цикла `while`, выполняемых алгоритмом `Augmented-BFS`. В качестве альтернативы теорема 8.3 (*а*) является частным случаем правильности алгоритма кратчайшего пути Дейкстры, как будет доказано в разделе 9.3.

8.3. Вычисление связных компонент

В этом разделе $G = (V, E)$ всегда будет обозначать неориентированный граф. Мы отложим более сложные задачи связности в ориентированных графах до раздела 8.6.

8.3.1. Связные компоненты

Неориентированный граф $G = (V, E)$ естественным образом распадается на части, которые называются связными компонентами (рис. 8.8). Если более формально, то связная компонента — это максимальное подмножество $S \subseteq V$ вершин, так что существует путь из любой вершины в S в любую другую вершину в S[1]. Например, связные компоненты графа на рис. 8.8 равны {1, 3, 5, 7, 9}, {2, 4} и {6, 8, 10}.

Цель этого раздела — применить поиск в ширину для вычисления связных компонент графа за линейное время[2].

---

[1] Еще более формально связные компоненты графа могут быть определены как *классы эквивалентности* подходящего *отношения эквивалентности*. Последние обычно рассматриваются на первом курсе в теме формальных доказательств или по дискретной математике. *Отношение* на множестве X объектов для каждой пары $x, y \in X$ объектов определяет, связаны или нет x и y. (Если да, то мы пишем $x \sim y$.) Для связных компонент соответствующее отношение (на множестве V) равно $v \sim_G w$ тогда и только тогда, когда в G существует путь между v и w. Отношение *эквивалентности* удовлетворяет трем свойствам. Во-первых, оно является *рефлексивным*, имея в виду $x \sim x$ для каждого $x \in X$. (Удовлетворяется отношением $\sim_G$, так как пустой путь соединяет вершину саму с собой.) Во-вторых, оно является *симметричным*, при этом $x \sim y$ тогда и только тогда, когда $y \sim x$. (Удовлетворяется отношением $\sim_G$, так как G является неориентированным.) И наконец, оно является *транзитивным*, имея в виду, что из $x \sim y$ и $y \sim z$ вытекает, что $x \sim z$. (Удовлетворяется отношением $\sim_G$, так как вы можете «склеить» путь между вершинами u и v с путем между вершинами v и w и тем самым получить путь между u и w.) Отношение эквивалентности разбивает множество объектов на классы эквивалентности, причем каждый объект связан со всеми объектами в своем классе и только с ними. Классы эквивалентности отношения $\sim_G$ представляют собой связные компоненты G.

[2] Другие алгоритмы графового поиска, включая поиск в глубину, можно использовать для вычисления связных компонент точно так же.

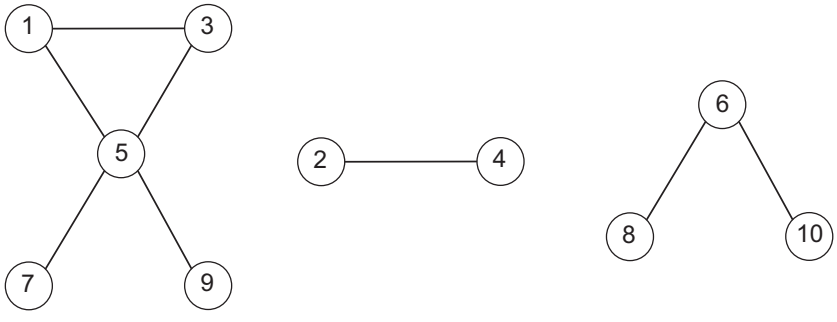


Рис. 8.8. Граф с множеством вершин {1, 2, 3, …, 10} и три связных компоненты

ЗАДАЧА: НЕОРИЕНТИРОВАННЫЕ СВЯЗНЫЕ КОМПОНЕНТЫ

Вход: неориентированный граф $G = (V, E)$.

Цель: определить связные компоненты графа G.

Далее давайте еще раз проверим ваше понимание определения связных компонент.

ТЕСТОВОЕ ЗАДАНИЕ 8.2

Рассмотрим неориентированный граф с n вершинами и m ребрами. Какое соответственно минимальное и максимальное число связных компонент этот граф может иметь?

а) 1 и $n - 1$

б) 1 и n

в) 1 и $\max\{m, n\}$

г) 2 и $\max\{m, n\}$

(Решение и пояснение см. в разделе 8.3.6.)

8.3.2. Применения

Существует несколько причин, почему вас могут заинтересовать связные компоненты графа.

Обнаружение сетевых сбоев. Одним из очевидных применений является проверка того, была ли отключена сеть, такая как дорожная или коммуникационная сеть.

Визуализация данных. Еще одно применение находится в области визуализации графов — если вы пытаетесь нарисовать или иным образом визуализировать граф, то разумно предположить, что вы захотите отобразить разные компоненты раздельно.

Кластеризация. Предположим, у вас есть коллекция объектов, которые имеют для вас значение, при этом каждая пара аннотирована как схожий или несхожий. Например, объектами могут быть документы (просмотренные веб-страницы или новостные репортажи), при этом схожие объекты соответствуют почти повторяющимся документам (возможно, отличающимся только временным штампом или заголовком). Либо объектами могут быть геномы, причем два генома считаются схожими, если малое число изменений может трансформировать одно в другое.

Теперь сформируем неориентированный граф $G = (V, E)$, вершины которого соответствуют объектам, а ребра — парам схожих объектов. Интуитивно каждая связная компонента этого графа представляет собой множество объектов, которые имеют между собой много общего. Так, если объектами являются просмотренные новостные репортажи, то можно ожидать, что вершины связной компоненты будут вариациями того же самого репортажа, сообщенного на других веб-сайтах. Если объектами являются геномы, то связная компонента может соответствовать разным особям, принадлежащим к одному виду.

8.3.3. Алгоритм UCC

Вычисление связных компонент неориентированного графа легко сводится к поиску в ширину (или другим алгоритмам поиска в графе, таким как поиск в глубину). Идея состоит в том, чтобы использовать внешний цикл для

выполнения одного обхода вершин, вызывая алгоритм `BFS` в качестве подпрограммы всякий раз, когда алгоритм встречает вершину, которую он никогда не встречал раньше. Этот внешний цикл гарантирует, что данный алгоритм заглянет на каждую вершину хотя бы один раз. Вершины инициализируются как неразведанные не внутри вызова алгоритма `BFS`, а перед внешним циклом. Данный алгоритм также поддерживает поле $cc(v)$ для каждой вершины v, чтобы помнить, какая компонента связности содержит эту вершину. Идентифицируя каждую вершину массива V с ее положением в массиве вершин, мы можем допустить, что $V = \{1,2,3, ..., n\}$.

UCC

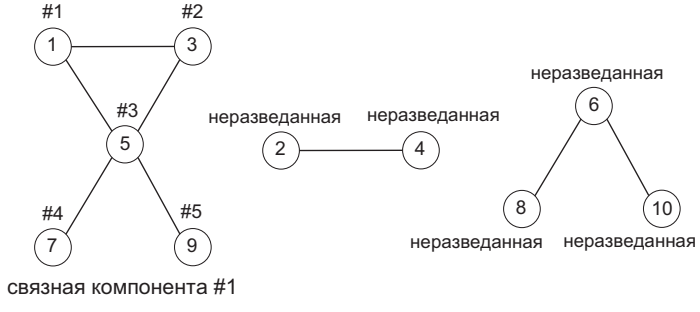
Вход: неориентированный граф $G = (V, E)$, представленный в виде списков смежности, где $V = \{1,2,3, ..., n\}$.

Постусловие: для каждой $u, v \in V$, $cc(u) = cc(v)$ тогда и только тогда, когда u, v находятся в одной и той же связной компоненте.

```
пометить все вершины как неразведанные
numCC := 0
for i := от 1 до n do          // перебрать все вершины
  if i не разведана then       // избежать избыточности
  numCC := numCC + 1           // новая компонента
  // вызвать алгоритм BFS, начиная с i (строки 2-8)
  Q := очередь, инициализированная значением i
  while Q не является пустой do
    удалить вершину из начала Q, назвать ее v
    cc(v) := numCC
    for каждая (v, w) в списке смежности вершины v do
      if w не разведана then
        пометить w как разведанную,
        добавить w в конец Q
```

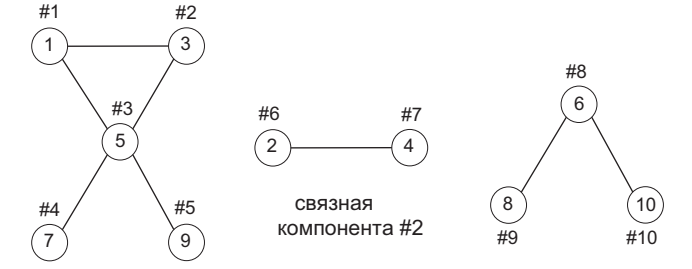
8.3.4. Пример

Проследим выполнение алгоритма UCC на графе (рис. 8.8). Данный алгоритм помечает все вершины как неразведанные и запускает внешний цикл for с вершиной 1. Эта вершина ранее не встречалась, поэтому данный алгоритм вызывает из нее подпрограмму с алгоритмом BFS. Поскольку подпрограмма BFS находит все достижимое из своей стартовой вершины (теорема 8.2 (*a*)), она обнаруживает все вершины в {1, 3, 5, 7, 9} и устанавливает их значения *cc*, равными 1. Один из возможных порядков разведывания имеет следующий вид:



По завершении этого вызова подпрограммы BFS внешний цикл for данного алгоритма продолжает выполнение и рассматривает вершину 2. Эта вершина не была обнаружена при первом вызове подпрограммы BFS, поэтому подпрограмма BFS вызывается снова, на этот раз с вершиной 2 в качестве стартовой вершины. После обнаружения вершин 2 и 4 (и установки их значений *cc* равными 2) этот вызов подпрограммы BFS завершается, и алгоритм UCC возобновляет свой внешний цикл for. Встречал ли этот алгоритм вершину 3 раньше? Да, во время первого вызова подпрограммы BFS. А как насчет вершины 4? Снова да, на этот раз во втором вызове подпрограммы BFS. А вершина 5? Был там и сделал это во время первого вызова подпрограммы BFS. Но как насчет вершины 6? Ни один из предыдущих вызовов подпрограммы BFS не обнаружил эту вершину, поэтому подпрограмма BFS вызывается снова с вершиной 6 в качестве стартовой вершины. Этот третий вызов подпрограммы BFS обнаруживает вершины в {6, 8, 10} и устанавливает значения *cc* равными 3.

Наконец, алгоритм верифицирует, что оставшиеся вершины (7, 8, 9 и 10) уже разведаны, и останавливается.



8.3.5. Правильность и время выполнения

Алгоритм UCC правильно вычисляет связные компоненты неориентированного графа и делает это за линейное время.

Теорема 8.4. (Свойства алгоритма UCC.) *Для каждого неориентированного графа $G = (V, E)$, представленного в виде списков смежности:*

а) по завершении алгоритма UCC *для каждой пары u, v вершин $cc(u) = cc(v)$ тогда и только тогда, когда u и v принадлежат одной и той же связной компоненте графа G;*

б) время работы алгоритма UCC *равно $O(m + n)$, где $m = |E|$ и $n = |V|$.*

Доказательство: в целях доказательства правильности из первого свойства поиска в ширину (теорема 8.2 (*а*)) вытекает, что каждый вызов подпрограммы BFS со стартовой вершиной *i* обнаружит вершины в связной компоненте вершины *i* и не более того. Алгоритм UCC дает этим вершинам общее значение *cc*. Поскольку ни одна вершина не разведывается дважды, каждый вызов подпрограммы BFS идентифицирует новую связную компоненту, причем каждая компонента имеет другое значение *cc*. Внешний цикл `for` гарантирует, что каждая вершина будет посещена хотя бы один раз, поэтому данный алгоритм обнаружит каждую связную компоненту.

Граница времени выполнения следует из нашего уточненного анализа времени выполнения алгоритма BFS (теорема 8.2 (*в*)). Каждый вызов подпрограммы BFS

из вершины i выполняется за время $O(m_i + n_i)$, где m_i и n_i обозначают число ребер и вершин в связной компоненте i соответственно. Поскольку подпрограмма `BFS` вызывается только один раз для каждой связной компоненты и каждая вершина или ребро графа G участвует ровно в одной компоненте, объединенное время выполнения всех вызовов подпрограммы `BFS` равно $O(\Sigma_i\, m_i + \Sigma_i\, n_i) = O(m + n)$. Инициализация и дополнительная служебная работа, выполняемые алгоритмом, требуют только времени $O(n)$, поэтому финальное время выполнения равно $O(m + n)$. *Ч. т. д.*

8.3.6. Решение тестового задания 8.2

Правильный ответ: (б). Граф с одной связной компонентой — это граф, в котором вы можете попасть из любого места в какое-либо другое. Графы путей и полные графы (рис. 8.7) являются двумя такими примерами. С другой стороны, в графе без ребер каждая вершина находится в своей собственной связной компоненте, коих в общей сложности n. Связных компонент не может быть больше n, так как они не пересекаются и каждая содержит хотя бы одну вершину.

8.4. Поиск в глубину

Зачем нужна еще одна стратегия графового поиска? В конце концов, поиск в ширину выглядит потрясающим — он находит все вершины, достижимые из стартовой вершины за линейное время, и может даже по ходу вычислять расстояния кратчайшего пути.

Существует еще одна линейно-временная стратегия графового поиска — *поиск в глубину* (*DFS*), которая сопровождается своим собственным впечатляющим каталогом применений (еще не охваченных стратегией поиска в ширину). Например, мы увидим, как использовать поиск в глубину для линейно-временного вычисления топологического упорядочивания вершин ориентированного ациклического графа, а также связных компонент (соответственно определенных) на ориентированном графе.

8.4.1. Пример

Если поиск в ширину представляет собой стратегию осторожного разведывания графа, то поиск в глубину является ее более решительным родственником, всегда разведывающим, находящимся в самой последней обнаруженной вершине и отступающим только при необходимости (например, разведывая лабиринт). Прежде чем мы опишем полный псевдокод для алгоритма `DFS`, поясним, как он работает, на том же рабочем примере, который использовался в разделе 8.2 (рис. 8.9).

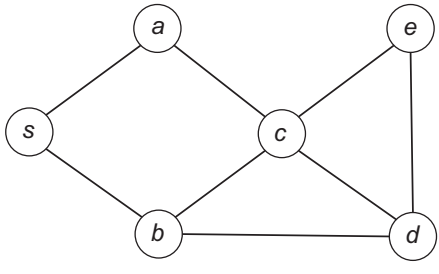
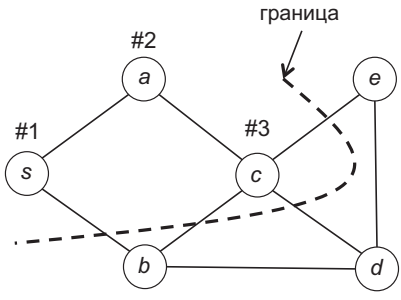
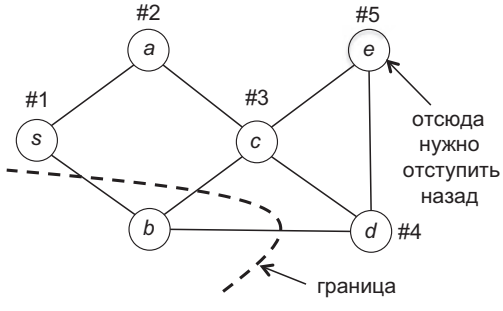


Рис. 8.9. Рабочий пример поиска в глубину

Как и алгоритм `BFS`, алгоритм `DFS` помечает вершину как разведанную в первый раз, когда он ее обнаруживает. Поскольку он начинает свое разведывание в стартовой вершине *s* для графа на рис. 8.9, первая итерация алгоритма `DFS` разведывает ребра (*s*, *a*) и (*s*, *b*) в любом порядке, в котором эти ребра появляются в списке смежности *s*. Предположим, сначала появляется (*s*, *a*), что приводит алгоритм `DFS` к обнаружению вершины *a*, которую он помечает как разведанную. Вторая итерация алгоритма `DFS` — это именно тот этап, где он расходится с алгоритмом `BFS`, — вместо того чтобы рассматривать следующего соседа *b* вершины *s* в слое 1, алгоритм `DFS` немедленно переходит к разведыванию соседей вершины *a*. (Со временем он вернется к разведыванию (*s*, *b*).) Возможно, из вершины *a* он сначала проверит вершину *s* (которая уже помечена как разведанная), а потом обнаружит вершину *c*, из которой он двинется дальше:



Затем алгоритм DFS в некотором порядке следования проверяет соседей вершины *c*, самую последнюю обнаруженную вершину. Ради интереса предположим,что алгоритм DFS следующей обнаруживает вершину *d*, за которой следует *e*:



Из вершины *e* алгоритму DFS идти некуда — оба соседа вершины *e* уже помечены как разведанные. Алгоритм DFS вынужден отступить к предыдущей вершине, а именно к вершине *d*, и возобновить разведывание остальных своих соседей. Из *d* алгоритм DFS обнаружит финальную вершину *b* (возможно, после проверки вершины *c* и обнаружения, что она помечена как разведанная). Находясь в *b*, костяшки домино быстро опрокидываются. Алгоритм DFS обнаруживает, что все соседи вершины *b* уже разведаны, и он должен вернуться к ранее посещенной вершине, которой является вершина *d*. Схожим образом, поскольку все остальные соседи *d* уже помечены как разведанные, алгоритму

DFS необходимо перемотать назад, вернувшись в вершину *c*. Алгоритм DFS затем отступает еще на один шаг (после проверки, что все остальные соседи вершины *c* помечены как разведанные), и затем к вершине *s*. Наконец, он останавливается, после того как проверит оставшегося соседа вершины *c* (то есть *b*) и обнаружит, что она помечена как разведанная.

8.4.2. Псевдокод для алгоритма DFS

Итеративная реализация

Один из способов реализации алгоритма DFS заключается в том, чтобы начать с исходного кода для BFS и внести два изменения: (i) подставить стековую структуру данных (то есть с дисциплиной доступа «последним вошел / первым вышел») вместо очереди (то есть с дисциплиной доступа «первым вошел/первым вышел»); и (ii) отложить проверку того, была ли вершина уже разведана, только после ее удаления из этой структуры данных[1, 2].

DFS (ИТЕРАТИВНАЯ ВЕРСИЯ)

Вход: граф $G = (V, E)$, представленный в виде списков смежности, и вершина $s \in V$.

Постусловие: вершина достижима из *s* тогда и только тогда, когда она помечена как «разведанная».

пометить все вершины как неразведанные

S := стек, инициализированный вершиной *s*

while S не является пустым **do**

---

[1] Стек — это структура данных с дисциплиной доступа «последним вошел / первым вышел», подобная тем стопкам подносов, которые можно увидеть в кафетерии, обычно изучаемая на первом курсе программирования (наряду с очередями, см. сноску на с. 52). Стек поддерживает список объектов, и вы можете добавлять объект в начало списка («push» — затолкнуть) или удалять его из начала списка («pop» — вытолкнуть) за постоянное время.

[2] Будет ли алгоритм вести себя так же, если мы внесем только первое изменение?

удалить (вытолкнуть) вершину v из головы стека S

if v не разведана **then**

пометить v как разведанную

for каждое ребро (v, w) в списке смежности вершины v **do**

добавить (втолкнуть) w в голову стека S

Как обычно, ребра, обрабатываемые в цикле `for`, являются ребрами, инцидентными v (если G является неориентированным графом), либо ребрами, исходящими из v (если G является ориентированным графом).

Например, в графе на рис. 8.9 первая итерация цикла `while` алгоритма `DFS` выталкивает вершину s и помещает ее двух соседей в стек в некотором порядке, скажем, b первой и a второй. Поскольку a была последней помещенной в стек, она является первой, которая будет вытолкнута на второй итерации цикла `while`. Это приводит к тому, что в стек будут помещены вершины s и c, скажем, сначала c. Вершина s выталкивается на следующей итерации; алгоритм ее пропускает, поскольку она уже помечена как разведанная. Затем выталкивается c, и все ее соседи (a, b, d и e) заталкиваются в стек, присоединяясь к первому вхождению b. Если d заталкивается последней, а также b выталкивается перед e, когда d выталкивается на следующей итерации, то восстанавливается порядок разведывания из раздела 8.4.1 (как вы можете убедиться сами).

Рекурсивная реализация

Поиск в глубину также имеет элегантную рекурсивную реализацию[1].

DFS (РЕКУРСИВНАЯ ВЕРСИЯ)

Вход: граф $G = (V, E)$, представленный в виде списков смежности, и вершина $s \in V$.

[1] Я исхожу из того, что вы слышали о рекурсии в рамках вашего опыта в программировании. Рекурсивная процедура — это процедура, которая вызывает саму себя как подпрограмму.

Постусловие: вершина достижима из *s* тогда и только тогда, когда она помечена как «разведанная».

```
// перед внешним вызовом все вершины не разведаны
пометить s как разведанную
for каждое ребро (s, v) в списке смежности вершины s do
  if v не разведана then
    DFS (G, v)
```

В этой реализации все рекурсивные вызовы алгоритма `DFS` имеют доступ к одному и тому же множеству глобальных переменных, которые отслеживают вершины, помеченные как разведанные (причем все вершины изначально не разведаны). Агрессивная природа алгоритма `DFS`, пожалуй, более очевидна именно в этой реализации — алгоритм немедленно выполняет рекурсию на первом неразведанном соседе, которого он находит, и только потом рассматривает оставшихся соседей[1]. По сути дела, явно заданная стековая структура данных в итеративной реализации алгоритма `DFS` симулируется программным стеком рекурсивных вызовов в рекурсивной реализации[2].

8.4.3. Правильность и время выполнения

Поиск в глубину является таким же правильным и таким же невероятно быстрым, как и поиск в ширину, по тем же самым причинам (ср.: теорема 8.2)[3].

---

[1] Как указано, две версии алгоритма `DFS` разведывают ребра в списке смежности вершины в противоположных порядках. (Вы видите, почему?) Если одну из версий модифицировать для итеративного перебора списка смежности вершины в обратном порядке, то итеративная и рекурсивная реализации будут разведывать вершины в одинаковом порядке.

[2] Профессиональный совет: если вашему компьютеру не хватает памяти при выполнении рекурсивной версии алгоритма `DFS` на большом графе, то вы должны либо переключиться на итеративную версию, либо увеличить размер программного стека в вашей среде программирования.

[3] Сокращение «ср.» значит «сравнить с».

Теорема 8.5. (Свойства алгоритма `DFS`.) *Для каждого неориентированного или ориентированного графа $G = (V, E)$, представленного в виде списков смежности, и для каждой стартовой вершины $s \in V$:*

а) по завершении алгоритма `DFS` *вершина $v \in V$ помечается как разведанная тогда и только тогда, когда в G существует путь из s в v;*

б) время работы алгоритма `DFS` *равно $O(m + n)$, где $m = |E|$ и $n = |V|$.*

Часть (*а*) соблюдается, потому что поиск в глубину является частным случаем обобщенного алгоритма графового поиска `GenericSearch` (утверждение 8.1)[1]. Часть (*б*) соблюдается, потому что алгоритм `DFS` проверяет каждое ребро не более чем два раза (один раз из каждой конечной точки) и, поскольку стек поддерживает операции вталкивания и выталкивания за время $O(1)$, он выполняет постоянное число операций в расчете на реберное разведывание (суммарно за время $O(m)$). Инициализация требует времени $O(n)$[2].

8.5. Топологическая сортировка

Поиск в глубину идеально подходит для вычисления топологического упорядочивания[3] ориентированного ациклического графа. Вы наверняка спросите, что это такое и кого это волнует.

---

[1] Формально алгоритм `DFS` эквивалентен версии алгоритма `GenericSearch`, в которой на каждой итерации цикла `while` алгоритм выбирает приемлемое ребро (v, w), для которого вершина v была обнаружена совсем недавно. Связи между приемлемыми ребрами вершины v разрываются в соответствии с их порядком (для рекурсивной версии) либо с их обратным порядком (для итеративной версии) в списке смежности вершины v.

[2] Уточненная граница в теореме 8.2 (*в*) также соблюдается для алгоритма `DFS` (по тем же причинам), а это означает, что алгоритм `DFS` может заменить алгоритм `BFS` в линейно-временном алгоритме `UCC` для вычисления связных компонент из раздела 8.3.

[3] По правилам русского языка упорядочение — это приведение в порядок естественным путем, тогда как упорядочивание — приведение в порядок в результате внешнего воздействия. Кроме того, упорядочение является результатом, тогда как упорядочивание — процессом. — *Примеч. пер.*

8.5.1. Топологические упорядочивания

Представьте, что у вас полно задач, которые нужно выполнить, и существуют *ограничения по предшествованию* — вы не можете начать некоторые задачи до тех пор, пока не завершите другие. Подумайте, например, о курсах университетской программы, которые необходимы в качестве предварительного условия для других. Одним из применений топологических упорядочиваний является построение последовательности операций с целью соблюдения всех ограничений по предшествованию.

ТОПОЛОГИЧЕСКИЕ УПОРЯДОЧИВАНИЯ

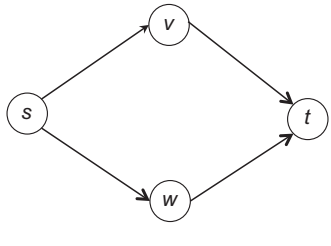
Пусть $G = (V, E)$ равно ориентированному графу. *Топологическим упорядочиванием* графа G является закрепление за $f(v)$ в каждой вершине $v \in V$ отличающегося числа, такого что:

для каждого $(v, w) \in E, f(v) < f(w)$.

Функция f эффективно упорядочивает вершины — от вершины с наименьшим значением f до вершины с наибольшим значением. Данное условие утверждает, что все (ориентированные) ребра графа G должны следовать вперед в упорядочении, при этом метка хвоста ребра должна быть меньше, чем метка его головы.

ТЕСТОВОЕ ЗАДАНИЕ 8.3

Сколько разных топологических упорядочений существует в приведенном ниже графе? Используйте только метки {1, 2, 3, 4}.



а) 0

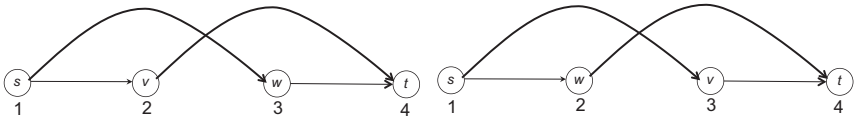
б) 1

в) 2

г) 3

(Решение и пояснение см. в разделе 8.5.7.)

Вы можете визуализировать топологическое упорядочивание, построив вершины в порядке их значений *f*. В топологическом упорядочивании все ребра графа ориентированы слева направо. На рис. 8.10 показаны топологические упорядочивания, выявленные в решении тестового задания 8.3.



а) Одно топологическое упорядочивание... *б*) ...и еще одно

Рис. 8.10. Топологическое упорядочивание эффективно строит вершины графа на прямой, причем все ребра идут слева направо

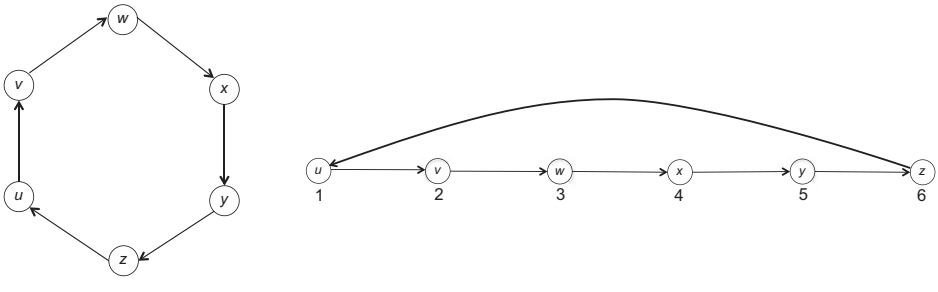
Когда вершины графа представляют операции, а ориентированные ребра представляют ограничения по предшествованию, топологические упорядочивания точно соответствуют разным способам построения последовательности операций при соблюдении ограничений по предшествованию.

8.5.2. Когда есть топологическое упорядочивание?

Все ли графы имеют топологическое упорядочивание? Ни в коем случае. Подумайте о графе, состоящем исключительно из ориентированного цикла (рис. 8.11, *а*). Неважно, какое упорядочивание вершин вы выберете, прохождение по ребрам цикла возвращает вас назад к стартовой точке, что возможно только тогда, когда некоторые ребра идут назад в упорядочении (рис. 8.11, *б*).

В более общем случае невозможно топологически упорядочить вершины графа, содержащего ориентированный цикл. Равным образом невозможно упорядочить множество операций, когда их зависимости зациклены.

К счастью, ориентированные циклы являются единственным препятствием для топологических упорядочиваний. Ориентированный граф без каких-либо ориентированных циклов называется — готовы? — *ориентированным ациклическим графом,* или просто *DAG* (*directed acyclic graph*). Например, граф на рис. 8.10 ориентирован ациклически; граф на рис. 8.11 — нет.



а) Ориентированный цикл *б*) Нетопологическое упорядочение

Рис. 8.11. Топологическое упорядочение может иметь только тот граф, который не имеет ориентированных циклов

Теорема 8.6. (Каждый граф DAG имеет топологическое упорядочение.) *Каждый ориентированный ациклический граф имеет минимум одно топологическое упорядочение.*

Для доказательства этой теоремы нам понадобится следующая лемма об истоковых вершинах. *Истоковая вершина* ориентированного графа — это вершина без входящих ребер. (Схожим образом *стоковая вершина* — это вершина без исходящих ребер.) Например, *s* является единственной истоковой вершиной в графе на рис. 8.10; ориентированный цикл на рис. 8.11 не имеет истоковых вершин.

Лемма 8.7. (Каждый граф DAG имеет исток.) *Каждый ориентированный ациклический граф имеет минимум одну истоковую вершину.*

Лемма 8.7 является истинной, потому что, если вы продолжите следовать по входящим ребрам назад из произвольной вершины ориентированного ациклического графа, вы обязательно достигнете истоковой вершины. (В противном случае вы породите цикл, что невозможно.) См. также рис. 8.12[1].

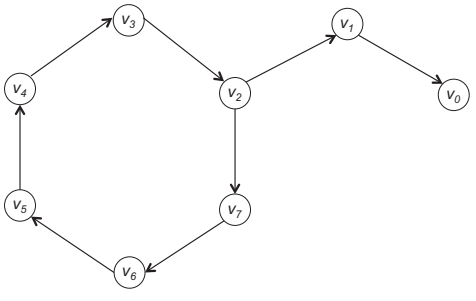


Рис. 8.12. Трассировка входящих ребер назад из текущей вершины не сможет найти истоковую вершину, только если граф содержит ориентированный цикл

Мы можем доказать теорему 8.6, наполняя топологическое упорядочение слева направо извлекаемыми подряд истоковыми вершинами[2].

Доказательство теоремы 8.6: пусть G равно ориентированному ациклическому графу с n вершинами. План состоит в присвоении вершинам значений f

[1] Более формально, выберите вершину v_0 ориентированного ациклического графа G; если это истоковая вершина, то мы закончили. Если нет, то она имеет минимум одно входящее ребро (v_1, v_0). Если v_1 является истоковой вершиной, то мы закончили. В противном случае существует входящее ребро формы (v_2, v_1), и мы можем повторить итерацию снова. После итеративного повтора вплоть до n раз, где n — это число вершин, мы либо находим истоковую вершину, либо создаем последовательность из n ребер (v_n, v_{n-1}), (v_{n-1}, v_{n-2}), ..., (v_1, v_0). Поскольку существует только n вершин, существует по крайней мере одна повторная вершина в последовательности $v_n, v_{n-1}, ..., v_0$. Но если $v_j = v_i$ при $j > i$, то ребра (v_j, v_{j-1}), ..., (v_{i+1}, v_i) образуют ориентированный цикл, противоречащий принятому допущению, что G является ориентированным ациклическим графом. (На рис 8.12 $i = 2$ и $j = 8$.)

[2] В качестве альтернативы: прослеживание не по входящим, а по исходящим ребрам в доказательстве леммы 8.7 показывает, что каждый граф DAG имеет минимум одну стоковую вершину, и мы можем наполнить топологическое упорядочение справа налево извлеченными подряд стоковыми вершинами.

в порядке возрастания от 1 до n. Какая вершина заслужила право иметь единицу в качестве своего значения f? Хорошо бы, чтобы это была истоковая вершина — если вершине с входящим ребром была присвоена первая позиция, то входящее ребро будет предыдущим в упорядочении. Таким образом, пусть v_1 равно истоковой вершине графа G — одна такая существует по лемме 8.7 — и назначим $f(v_1) = 1$. Если существует несколько истоковых вершин, то выберем одну из них произвольно.

Далее, мы получим граф G' из G, удалив вершину v_1 и все ее ребра. Поскольку G является ориентированным ациклическим графом, то таковым является и G' — удаление элементов не может создать новые циклы. Поэтому мы можем рекурсивно вычислить топологическое упорядочение графа G', используя метки $\{2, 3, 4, ..., n\}$, причем каждое ребро в G' следует в упорядочении вперед. (Поскольку каждый рекурсивный вызов выполняется на меньшем графе, рекурсия в конечном итоге остановится.) Только те ребра в G, которые одновременно не находятся в G', являются (исходящими) ребрами вершины v_1; поскольку $f(v_1) = 1$, они также следуют в упорядочении вперед[1]. *Ч. т. д.*

8.5.3. Вычисление топологического упорядочивания

Из теоремы 8.6 следует, что имеет смысл запрашивать топологическое упорядочивание ориентированного графа тогда и только тогда, когда граф является ориентированным ациклическим.

ЗАДАЧА: ТОПОЛОГИЧЕСКАЯ СОРТИРОВКА

Вход: ориентированный ациклический граф $G = (V, E)$.

Выход: топологическое упорядочение вершин графа G.

[1] Если вы предпочитаете формальное доказательство правильности, выполните его в уединении домашней обстановки по индукции на числе вершин.

Доказательства леммы 8.7 и теоремы 8.6 естественным образом приводят к алгоритму. Для n-вершинного ориентированного ациклического графа, представленного в виде списков смежности, первое доказательство дает $O(n)$-временную подпрограмму для нахождения истоковой вершины. Последнее доказательство вычисляет топологическое упорядочивание с n вызовами этой подпрограммы, выхватывая новую истоковую вершину на каждой итерации[1]. Время работы этого алгоритма составляет $O(n^2)$, то есть линейно-временное для самых плотных графов (с $m = \Theta(n^2)$ ребрами), но не для более разреженных графов (где n^2 может быть намного больше m). Далее следует более остроумное решение посредством поиска в глубину, приводящее к линейно-временному ($O(m + n)$) алгоритму[2].

8.5.4. Топологическая сортировка посредством алгоритма DFS

Блестящий способ вычислить топологическое упорядочивание заключается в усилении поиска в глубину за счет двух небольших дополнений. Для простоты мы начнем с рекурсивной реализации алгоритма `DFS` из раздела 8.4. Первое дополнение — это внешний цикл, который делает один проход над вершинами, вызывая алгоритм `DFS` как подпрограмму всякий раз, когда обнаруживается ранее неразведанная вершина. Этим гарантируется, что каждая вершина со временем будет обнаружена, и ей будет присвоена метка. Глобальная переменная `curLabel` отслеживает, где мы находимся в топологическом упорядочивании. Наш алгоритм будет вычислять упорядочивание в обратном порядке (справа налево), поэтому `curLabel` отсчитывает назад от числа вершин до 1.

---

[1] Для графа на рис. 8.10 этот алгоритм может вычислить любое из двух топологических упорядочиваний, в зависимости от того, какая из вершин v, w выбрана в качестве истоковой вершины на второй итерации после удаления s.

[2] При наличии некоторой сообразительности алгоритм, неявно присутствующий в доказательствах леммы 8.7 и теоремы 8.6, также может быть реализован с линейным временем — видите, как это сделать?

TOPOSORT

Вход: ориентированный ациклический граф $G = (V, E)$, представленный в виде списков смежности.

Постусловие: значения f вершин образуют топологическую упорядоченность графа G.

пометить все вершины как неразведанные

curLabel := |*V*| `// отслеживает упорядочивание`

for каждая $v \in V$ **do**

  if v не разведана **then** `// в предыдущем DFS`

    `DFS-Topo` (G, v)

Вторым дополнением мы должны добавить строку кода в алгоритм `DFS`, которая присваивает значение f вершине. Самое подходящее время для этого — сразу после завершения вызова алгоритма `DFS`, инициированного в вершине v.

DFS-TOPO

Вход: граф $G = (V, E)$, представленный в виде списков смежности, и вершина $s \in V$.

Постусловие: каждая вершина, достижимая из s, помечается как «разведанная» и имеет присвоенное ей значение f.

пометить s как разведанную

for каждое ребро (s, v) в исходящем списке смежности s **do**

  if v не разведана **then**

    `DFS-Topo` (G, v)

$f(s) := curLabel$ `// позиция s в упорядочении`

$curLabel := curLabel - 1$ `// двигаться справа налево`

8.5.5. Пример

Предположим, что входным графом является граф из тестового задания 8.3. Алгоритм `TopoSort` инициализирует глобальную переменную `curLabel` числом вершин, равным 4. Внешний цикл алгоритма `TopoSort` перебирает вершины в произвольном порядке; будем считать, что этот порядок равен v, t, s, w. На первой итерации, поскольку вершина v не помечена как разведанная, данный алгоритм вызывает подпрограмму `DFS-Topo` со стартовой вершиной v. Единственным исходящим из v ребром является (v, t), и следующий шаг состоит из рекурсивного вызова подпрограммы `DFS-Topo` со стартовой вершиной t. Этот вызов возвращается немедленно (так как вершина t не имеет исходящих ребер), и на этом этапе $f(t)$ устанавливается равным 4, а переменная `curLabel` уменьшается с 4 до 3. Затем вызов подпрограммы `DFS-Topo` в вершине v завершается (поскольку v не имеет других исходящих ребер), и на этом этапе $f(v)$ устанавливается равным 3, а переменная `curLabel` уменьшается с 3 до 2. На данном этапе алгоритм `TopoSort` возобновляет линейное сканирование вершин в своем внешнем цикле. Следующей вершиной является t; поскольку t уже была помечена как разведанная в первом вызове подпрограммы `DFS-Topo`, алгоритм `TopoSort` ее пропускает. Поскольку следующая вершина (то есть c) еще не была разведана, алгоритм вызывает подпрограмму `DFS-Topo` из s. Из вершины s подпрограмма `DFS-Topo` пропускает вершину v (которая уже была помечена как разведанная) и рекурсивно вызывает подпрограмму `DFS-Topo` в только что обнаруженной вершине w. Вызов в вершине w сразу же завершается (единственное исходящее ребро направлено в ранее разведанную вершину t), и на этом этапе $f(w)$ устанавливается равным 2, а переменная `curLabel` уменьшается с 2 до 1. Наконец, вызов подпрограммы `DFS-Topo` в вершине s завершается, и $f(s)$ устанавливается равным 1. Результирующее топологическое упорядочение является таким же, как на рис. 8.10, *б*.

ТЕСТОВОЕ ЗАДАНИЕ 8.4

Что происходит, когда алгоритм `TopoSort` выполняется на графе с ориентированным циклом?

а) Алгоритм может зациклиться, а может и нет.

б) Алгоритм всегда зацикливается.

в) Алгоритм всегда останавливается, и он сможет вычислить топологическое упорядочивание успешно либо не сможет.

г) Алгоритм всегда останавливается и никогда не вычисляет топологическое упорядочивание.

(Решение и пояснение см. в разделе 8.5.7.)

8.5.6. Правильность и время выполнения

Алгоритм `TopoSort` правильно вычисляет топологическое упорядочивание ориентированного ациклического графа и делает это за линейное время.

Теорема 8.8. (Свойства алгоритма `TopoSort`.) *Для каждого ориентированного ациклического графа $G = (V, E)$, представленного в виде списков смежности:*

а) по завершении алгоритма `TopoSort` *каждой вершине v было присвоено значение f, и эти значения f образуют топологическое упорядочение графа G;*

б) время работы алгоритма `TopoSort` *равно $O(m + n)$, где $m = |E|$ и $n = |V|$.*

Доказательство: алгоритм `TopoSort` работает с линейным временем по обычным причинам. Он разведывает каждое ребро только один раз (из его хвоста) и поэтому выполняет только постоянное число операций для каждой вершины или ребра. Из этого вытекает совокупное время работы $O(m + n)$.

Что касается правильности, то, во-первых, обратите внимание, что подпрограмма `DFS-Topo` будет вызываться из каждой вершины $v \in V$ ровно один раз, когда v встречается впервые, и что за вершиной v закрепляется метка, когда этот вызов завершается. Таким образом, каждая вершина получает метку, и, уменьшая переменную `curLabel` вместе с каждым закреплением метки, алгоритм обеспечивает то, что каждая вершина v получает несовпадающую метку $f(v)$ из множества $\{1, 2, ..., |V|\}$. Для того чтобы понять, почему эти

метки образуют топологическое упорядочение, рассмотрим произвольное ребро (v, w); мы должны доказать, что $f(v) < f(w)$. Существует два случая, в зависимости от того, какую вершину из v, w алгоритм обнаруживает первой[1].

Если вершина v обнаруживается до w, то подпрограмма `DFS-Topo` вызывается со стартовой вершиной v до того, как вершина w была помечена как разведанная. Так как w достижима из v (через ребро (v, w)), этот вызов подпрограммы `DFS-Topo` в конце концов обнаруживает w и рекурсивно вызывает подпрограмму `DFS-Topo` в w. По характеру рекурсивных вызовов с дисциплиной «последним вошел / первым вышел» вызов подпрограммы `DFS-Topo` в w завершается до завершения в v. Поскольку метки присваиваются в убывающем порядке, вершине w присваивается более крупное значение f, чем вершине v, как и требуется.

Во-вторых, предположим, что вершина w обнаружена алгоритмом `TopoSort` до вершины v. Поскольку G является ориентированным ациклическим графом, путь из w назад в v отсутствует; в противном случае объединение такого пути с ребром (v, w) привело бы к ориентированному циклу (рис. 8.13). Таким образом, вызов подпрограммы `DFS-Topo`, начинающийся в w, не может обнаружить v и завершается, когда v остается по-прежнему неразведанной. Еще раз: вызов подпрограммы `DFS-Topo` в w завершается до завершения в v и, следовательно, $f(v) < f(w)$. *Ч. т. д.*

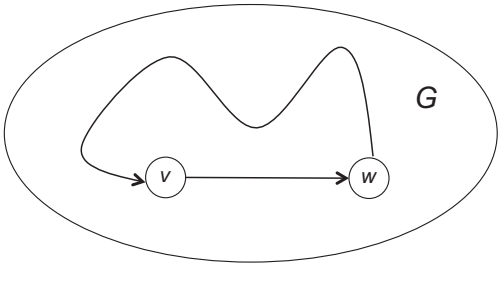


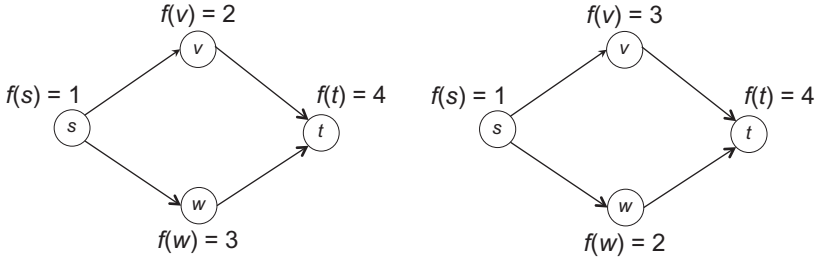
Рис. 8.13. Ориентированный ациклический граф не может содержать ребро (v, w) и путь из w назад в v

[1] Как мы видели в разделе 8.5.5, оба случая являются возможными.

8.5.7. Решения тестовых заданий 8.3–8.4

Решение тестового задания 8.3

Правильный ответ: (в). На рис. 8.14 показаны два разных топологических упорядочивания графа — вы должны убедиться, что они единственные.



а) Топологическое упорядочение *б*) ...и еще одно

Рис. 8.14. Два топологических упорядочения графа из тестового задания 8.3

Решение тестового задания 8.4

Правильный ответ: (г). Алгоритм всегда останавливается: существует только $|V|$ итераций внешнего цикла, и каждая итерация либо ничего не делает, либо вызывает поиск в глубину (с незначительной дополнительной служебной работой). Поиск в глубину всегда останавливается, независимо от того, является ли входной граф ориентированным ациклическим или нет (теорема 8.5), и поэтому алгоритм `TopoSort` делает то же самое. Существует ли шанс, что он остановится с топологическим упорядочиванием? Никаких шансов — топологически отсортировать вершины любого графа с ориентированным циклом невозможно (вспомните раздел 8.5.2).

*8.6. Вычисление сильно связных компонент

Далее мы узнаем еще более интересное применение поиска в глубину, а именно — вычисление сильно связных компонент ориентированного графа[1]. Наш алгоритм будет столь же молниеносным, как и в неориентированном случае (раздел 8.3), хотя и менее простым. Вычисление сильно связных компонент является более сложной задачей, чем топологическая сортировка, и одного прохода поиска в глубину будет недостаточно. Поэтому мы будем использовать два![2]

8.6.1. Определение сильно связных компонент

Что мы вообще подразумеваем под связной компонентой ориентированного графа? Например, сколько связных компонент имеется в графе на рис. 8.15?

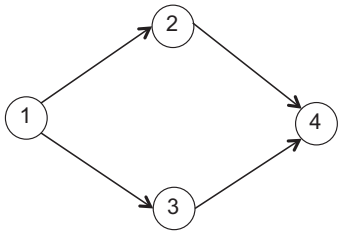


Рис. 8.15. Сколько связных компонент вы можете назвать?

Так и хочется сказать, что этот граф имеет одну связную компоненту — если бы он был физическим объектом, ребра которого соответствовали нитям, связывающим вершины вместе, то мы могли бы его «поднять», и он «висел» бы

---

[1] Отмеченные звездочкой разделы, подобные этому, являются более сложными; при первом чтении их можно пропустить.

[2] На самом деле существует несколько хитроумный способ вычислить сильно связные компоненты ориентированного графа только с одним проходом поиска в глубину; см. статью *«Depth-First Search and Linear Graph Algorithms»* («Алгоритмы поиска в глубину и линейные графовые алгоритмы») *Роберта Э. Тарьяна* (*SIAM Journal on Computing, 1973*).

как одно целое. Но вспомните, что мы определили связные компоненты в неориентированном случае (раздел 8.3) как максимальные области, в пределах которых вы можете добраться из какого-либо места в любое другое. В графе на рис. 8.15 нет никакой возможности двигаться влево, поэтому это не тот случай, когда можно добраться из любой точки в любую другую.

Сильно связная компонента (*strongly connected component*, *SCC*) ориентированного графа является максимальным подмножеством $S \subseteq V$ вершин, так что существует ориентированный путь из любой вершины в S в любую другую вершину в S[1]. Например, сильно связными компонентами графа на рис. 8.16 являются {1, 3, 5}, {11}, {2, 4, 7, 9} и {6, 8, 10}. Внутри каждой

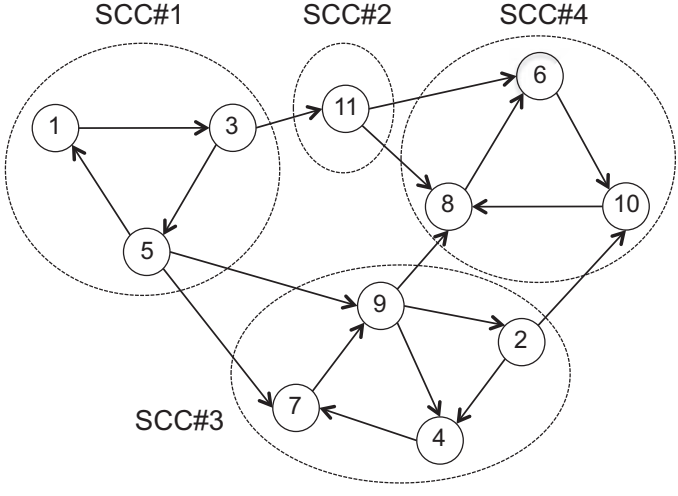


Рис. 8.16. Граф с множеством вершин {1, 2, 3, ..., 11} и четыре сильно связные компоненты

[1] Как и со связными компонентами в неориентированных графах (сноска на с. 61), сильно связные компоненты ориентированного графа G являются как раз классами эквивалентности отношения эквивалентности $\sim_G$, где $v \sim_G w$ тогда и только тогда, когда существуют пути из v в w, а из w в v в G. Доказательство того, что $\sim_G$ является отношением эквивалентности, воспроизводит это в неориентированном случае.

компоненты можно добраться из любого места в любое другое (как вы сами можете убедиться). Каждая компонента является максимальной в отношении этого свойства, так как нет возможности двигаться влево из одной сильно связной компоненты в другую.

Отношения между четырьмя сильно связными компонентами графа на рис. 8.16 воспроизводят те, которые находятся между четырьмя вершинами в графе на рис. 8.15. В целом, если присмотреться, *каждый* ориентированный граф можно рассматривать как ориентированный ациклический, построенный из его сильно связных компонент.

Утверждение 8.9. (Метаграф сильно связных компонент является ориентированным ациклическим графом.) *Пусть $G = (V, E)$ равно ориентированному графу. Определим соответствующий метаграф $H = (X, F)$ с одной метавершиной $x \in X$ на сильно связную компоненту графа G и метаребром (x, y) в F всякий раз, когда существует ребро в G из вершины в сильно связной компоненте, соответствующей x, в вершину в сильно связной компоненте, соответствующей y. Тогда H является ориентированным ациклическим графом.*

Так, ориентированный ациклический граф на рис. 8.15 является метаграфом, соответствующим ориентированному графу на рис. 8.16.

Доказательство утверждения 8.9: если бы метаграф H имел ориентированный цикл с $k \geq 2$ вершинами, то соответствующий цикл предположительно несовпадающих сильно связных компонент $S_1, S_2, \ldots, S_k$ в G сложился бы в одну сильно связную компоненту: вы уже можете свободно перемещаться в пределах каждого из S_i, и цикл тогда позволит перемещаться между любой парой S_i. *Ч. т. д.*

Из утверждения 8.9 вытекает, что на каждый ориентированный граф можно смотреть с двух уровней детализации. При уменьшении масштаба вы фокусируетесь только на (ациклических) связях между его сильно связными компонентами; увеличение масштаба до определенной сильно связной компоненты показывает его мелкозернистую структуру.

ТЕСТОВОЕ ЗАДАНИЕ 8.5

Рассмотрим ориентированный ациклический граф с n вершинами и m ребрами. Каково соответственно минимальное и максимальное число сильно связных компонент, которые мог бы иметь граф?

а) 1 и 1

б) 1 и n

в) 1 и m

г) n и n

(Решение и пояснение см. в разделе 8.6.7.)

8.6.2. Почему поиск в глубину?

Для того чтобы понять, почему графовый поиск способен помочь в вычислении сильно связных компонент, вернемся к графу на рис. 8.16. Предположим, что мы вызываем поиск в глубину (либо поиск в ширину) из вершины 6. Алгоритм найдет все достижимое из 6 и ничего больше, обнаружив {6, 8, 10}, то есть именно одну из сильно связных компонент. Плохим решением будет, если мы вместо этого инициируем графовый поиск из вершины 1, в каковом случае обнаруживаются все вершины (а не только {1, 3, 5}), и мы не получаем никаких сведений о компонентной структуре.

Основной вывод заключается в том, что графовый поиск может выявлять сильно связные компоненты, при условии что вы начинаете с правильного места. Интуитивно мы хотим сначала обнаружить стоковую сильно связную компоненту, то есть сильно связную компоненту без исходящих ребер (подобную *SCC#4* на рис. 8.16), а затем двигаться в обратном направлении. В терминах метаграфа в утверждении 8.9, по всей видимости, мы хотим обнаруживать сильно связные компоненты в обратном топологическом порядке, выхватывая стоковые сильно связные компоненты одну за другой. В разделе 8.5 мы уже видели, что топологические упорядочивания находятся в прямом ведении поиска в глубину, и именно по этой причине наш алгоритм будет использовать

два прохода поиска в глубину. Первый проход вычисляет волшебное упорядочение, в котором вершины будут обрабатываться, и второй подчиняется этому упорядочению с целью обнаружить сильно связные компоненты одну за другой. Эта двухпроходная стратегия называется *алгоритмом Косарайю*[1].

Эффекта ради: вот заблаговременное предупреждение о том, как выглядит алгоритм Косарайю с высоты 9 километров:

КОСАРАЙЮ (ВЫСОКОУРОВНЕВЫЙ)

1. Пусть G^{rev} обозначает входной граф G, в котором направление каждого ребра повернуто в обратную сторону.
2. Вызвать алгоритм `DFS` из каждой вершины графа G^{rev}, обработанной в произвольном порядке, для того чтобы вычислить позицию $f(v)$ для каждой вершины v.
3. Вызвать алгоритм `DFS` из каждой вершины графа G, обработанной от самой высокой до самой низкой позиции, для того чтобы вычислить идентичность сильно связного компонента каждой вершины.

У вас может быть хоть какая-то интуиция в отношении второго и третьего шагов алгоритма Косарайю. Второй шаг, по-видимому, делает что-то подобное алгоритму `TopoSort` из раздела 8.5 с целью обработки сильно связных компонент входного графа на третьем шаге в обратном топологическом порядке. (Предостережение: мы думали об алгоритме `TopoSort` только в графах `DAG`, а здесь у нас общий ориентированный граф.) Третий шаг, как мы надеемся, аналогичен алгоритму `UCC` из раздела 8.3 для неориентированных графов. (Предостережение: в неориентированных графах порядок, в котором вы обрабатываете вершины, не имеет значения; в ориентированных графах,

[1] Данный алгоритм впервые появился в неопубликованной статье *С. Рао Косарайю* (*S. Rao Kosaraju*) в 1978 году. Этот алгоритм также был обнаружен *Мичей Шариром* (*Micha Sharir*), который опубликовал его в статье *«A Strong-Connectivity Algorithm and Its Applications in Data Flow Analysis»* («Алгоритм сильной связности и его применения в анализе потоков данных») (*Computers & Mathematics with Applications, 1981*). Данный алгоритм также иногда называют *алгоритмом Косарайю—Шарира*.

как мы видели, это так. Но как быть с первым шагом? Почему первый проход работает с графом, развернутым в обратную сторону?)

8.6.3. Почему обратный граф?

Давайте сначала рассмотрим более естественную идею вызова алгоритма `TopoSort` из раздела 8.5 на исходном входном графе $G = (V, E)$. Напомним, что этот алгоритм имеет внешний цикл `for`, который делает проход над вершинами G в произвольном порядке; инициирует поиск в глубину при обнаружении еще не разведанной вершины и назначает позицию $f(v)$ вершине v, когда поиск в глубину, инициированный в вершине v, завершается. Позиции назначаются в убывающем порядке, начиная с $|V|$ вниз до 1.

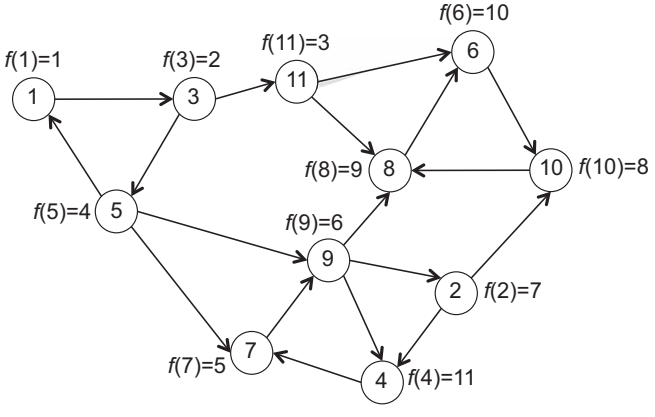
Алгоритм `TopoSort` первоначально был обусловлен случаем с ориентированным ациклическим входным графом, но его можно использовать для вычисления позиций вершин для произвольного ориентированного графа (тестовое задание 8.4). Мы надеемся, что эти позиции вершин каким-то образом будут полезны для быстрого определения хорошей стартовой вершины для нашего второго прохода поиска в глубину, в идеале вершины в стоковой сильно связной компоненте графа G, без исходящих ребер. Основания для оптимизма есть: в случае с ориентированным ациклическим графом G позиции вершин образуют топологическую упорядоченность (теорема 8.8), и вершина в последней позиции должна быть стоковой вершиной графа G без исходящих ребер. (Любые такие ребра будут направлены назад в упорядочении.) Возможно ли, что в случае с общим ориентированным графом G вершина в последней позиции всегда принадлежит стоковой сильно связной компоненте?

Пример

К сожалению, нет. Например, предположим, что мы выполняем алгоритм `TopoSort` на графе рис. 8.16. Предположим, что мы обрабатываем вершины в возрастающем порядке, причем вершина 1 рассматривается первой. (В этом случае все вершины обнаруживаются на первой итерации внешнего цикла.)

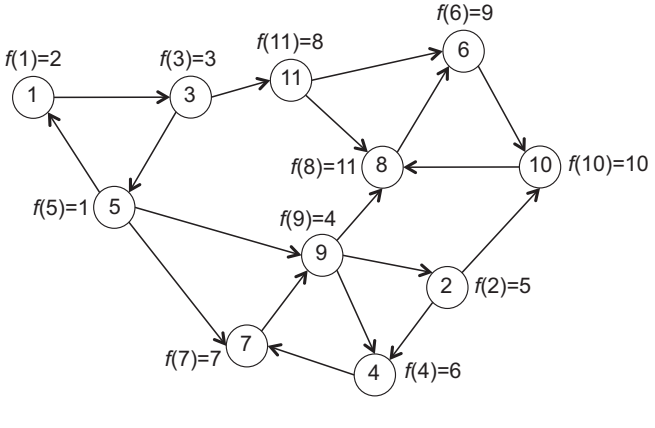
Предположим далее, что поиск в глубину проходит через ребро (3, 5) до ребер (3, 11), (5, 7) до ребер (5, 9), (9, 4) до ребра (9, 2) и (9, 2) до ребра (9, 8).

В этом случае вам следует проверить, что получаются следующие ниже позиции вершин:



Вопреки нашему желанию, вершина в последней позиции (вершина 4) не принадлежит стоковой сильно связной компоненте. Хорошей новостью является тот факт, что вершина в первой позиции (вершина 1) принадлежит истоковой сильно связной компоненте (имея в виду сильно связную компоненту без входящих ребер).

А что, если вместо этого обработать вершины в убывающем порядке? Если поиск в глубину проходит через ребро (11, 6) до ребра (11, 8) и ребро (9, 2) до ребра (9, 4), то (как вы сами можете убедиться) позиции вершин составят:



На этот раз вершина в последней позиции находится в стоковой сильно связной компоненте, но мы знаем, что этого не происходит в общем случае. Более интригует то, что вершина в первой позиции принадлежит истоковой сильно связной компоненте, хотя и другой вершине из этой сильно связной компоненты, нежели в прошлый раз. Может ли это быть верным в общем случае?

Первая вершина находится в истоковой сильно связной компоненте

На самом деле есть кое-что посерьезней: если мы пометим каждую сильно связную компоненту графа G наименьшей позицией одной из ее вершин, то эти метки образуют топологическое упорядочение метаграфа сильно связных компонент, определенного в утверждении 8.9.

Теорема 8.10. (Топологическое упорядочивание сильно связных компонент.) *Пусть G равно ориентированному графу, в котором вершины упорядочены произвольно, и для каждой вершины $v \in V$ пусть $f(v)$ обозначает позицию v, вычисляемую алгоритмом* `TopoSort`. *Пусть S_1, S_2 обозначают две сильно связных компоненты графа G, и пусть G имеет ребро (v, w), где $v \in S_1$ и $w \in S_2$. Тогда*

$$\min_{x \in S_1} f(x) < \min_{x \in S_2} f(y).$$

Доказательство: доказательство будет схожим с доказательством правильности алгоритма `TopoSort` (теорема 8.8, которую сейчас стоит перечитать заново). Пусть S_1, S_2 обозначают две сильно связных компоненты графа G, при этом рассмотрим два случая[1]. В первом случае предположим, что алгоритм `TopoSort` обнаруживает и инициирует поиск в глубину из вершины s компоненты S_1 перед любой вершиной компоненты S_2. Поскольку существует ребро из вершины v в S_1 в вершину w в S_2, и S_1 и S_2 являются сильно связными компонентами, каждая вершина компоненты S_2 является достижимой из s — для того чтобы достичь вершину $y \in S_2$, надо склеить путь $s \rightsquigarrow v$ внутри компоненты S_1, ребро (v, w) и путь $w \rightsquigarrow y$ внутри компоненты S_2. По характеру

[1] Как мы видели в предыдущем примере, оба случая возможны.

рекурсивных вызовов с дисциплиной «последним вошел / первым вышел» поиск в глубину, инициированной в s, не будет завершен до тех пор, пока все вершины компоненты S_2 не будут полностью разведаны. Поскольку позиции вершин назначаются в убывающем порядке, позиция v будет меньше, чем у каждой вершины компоненты S_2.

Во втором случае предположим, что алгоритм `TopoSort` обнаруживает вершину $s \in S_2$ до любой вершины S_1. Поскольку метаграф G является ориентированным ациклическим (утверждение 8.9), ориентированный путь из s в любую вершину компоненты S_1 отсутствует. (Такой путь сложил бы S_1 и S_2 в одну сильно связную компоненту.) Таким образом, поиск в глубину, инициированный в s, завершается после обнаружения всех вершин компоненты S_2 (и, возможно, других вещей) и ни одной из вершин компоненты S_1. В этом случае каждой вершине компоненты S_1 присваивается позиция меньшая, чем у *каждой* вершины компоненты S_2. *Ч. т. д.*

Из теоремы 8.10 вытекает, что вершина в первой позиции всегда располагается в истоковой сильно связной компоненте, как мы и надеялись. Рассмотрим вершину v с $f(v) = 1$, обитающую в сильно связной компоненте S. Если бы S не была бы истоковой сильно связной компонентой с ребром, входящим из другой сильно связной компоненты S', то по теореме 8.10 наименьшая позиция вершины в S' была бы меньше 1, что невозможно.

Подведем итог: после одного прохода поиска в глубину мы можем сразу определить вершину в истоковой сильно связной компоненте. Единственная проблема — мы хотим определить вершину в стоковой сильно связной компоненте. *Решение — сначала развернуть граф в обратную сторону*.

Разворот графа в обратную сторону

ТЕСТОВОЕ ЗАДАНИЕ 8.6

Пусть G равно ориентированному графу, а G^{rev} равно копии графа G, в котором каждое ребро развернуто в обратную сторону. Насколько сильно связные компоненты графов G и G^{rev} связаны между собой? (Выберите все подходящие варианты.)

а) В целом они не связаны между собой.

б) Каждая сильно связная компонента графа G также является сильно связной компонентой графа G^{rev}, и наоборот.

в) Каждая истоковая сильно связная компонента графа G также является истоковой сильно связной компонентой графа G^{rev}.

г) Каждая стоковая сильно связная компонента графа G становится истоковой сильно связной компонентой графа G^{rev}.

(Решение и пояснение см. в разделе 8.6.7.)

Следующее далее следствие переписывает теорему 8.10 для обратного графа, используя решение тестового задания 8.6.

Следствие 8.11. *Пусть G равно ориентированному графу, вершины которого упорядочены произвольно, а для каждой вершины $v \in V$ пусть $f(v)$ обозначает позицию v, вычисляемую алгоритмом TopoSort на обратном графе G^{rev}. Пусть S_1, S_2 обозначают две сильно связных компоненты графа G, и пусть G имеет ребро (v, w), где $v \in S_1$ и $w \in S_2$. Тогда*

$$\min_{x \in S_1} f(x) > \min_{y \in S_2} f(y). \tag{8.1}$$

В частности, вершина в первой позиции располагается в стоковой сильно связной компоненте графа G и является идеальной стартовой точкой для второго прохода поиска в глубину.

8.6.4. Псевдокод для алгоритма Косарайю

Давайте сначала приведем все в порядок: мы выполняем один проход поиска в глубину (посредством алгоритма `TopoSort`) на обратном графе, вычисляющий волшебное упорядочивание, в котором требуется посетить вершины, и второй проход (посредством подпрограммы `DFS-Topo`), для того чтобы обнаружить сильно связные компоненты в обратном топологическом порядке, вынимая их одну за другой, как слои с луковицы.

KOSARAJU

Вход: ориентированный граф $G = (V, E)$, представленный в виде списков смежности, с $V = \{1, 2, 3, ..., n\}$.

Постусловие: для каждой $v, w \in V$, $scc(v) = scc(w)$ тогда и только тогда, когда v, w находятся в одной и той же сильной связной компоненте графа G.

G^{rev} := G, в котором все ребра развернуты в обратную сторону

пометить все вершины G^{rev} как неразведанные

```
// первый проход поиска в глубину
// (вычисляет позиции f(v), волшебную упорядоченность)
TopoSort (G^rev)
// второй проход поиска в глубину
// (находит сильно связные компоненты
// в обратном топологическом порядке)
```

пометить все вершины G как неразведанные

numSCC := 0 `// глобальная переменная`

for каждая $v \in V$, в порядке возрастания $f(v)$ **do**

  if v не разведана **then**

    numSCC := *numSCC* + 1

    `// назначить scc-значения (подробности ниже)`

    `DFS-SCC` (G, v)

Три детали реализации[1]:

1. Наиболее очевидный способ реализации алгоритма — сделать буквальную вторую копию входного графа, в котором все ребра разверну-

[1] Для того чтобы оценить их по-настоящему, лучше всего реализовать алгоритм самостоятельно (см. задачу по программированию 8.10).

ты в обратную сторону, и передать ее в подпрограмму `TopoSort`. Более умная реализация выполняет алгоритм `TopoSort` в обратную сторону на исходном графе путем замены предложения «каждое ребро (s, v) в исходящем списке смежности» в подпрограмме `DFS-Topo` раздела 8.5 на «каждое ребро (v, s) в исходящем списке смежности».

2. Для получения наилучших результатов первый проход поиска в глубину должен экспортировать массив, содержащий вершины (либо указатели на них) в порядке их позиций с целью, чтобы второй проход мог обрабатывать их с помощью простого сканирования массива. Эта работа добавляет в функцию `TopoSort` только постоянные накладные расходы (как вы сами можете убедиться).
3. Подпрограмма `DFS-SCC` такая же, как и `DFS`, с одной дополнительной служебной строкой кода:

DFS-SCC

Вход: ориентированный граф $G = (V, E)$, представленный в виде списков смежности, и вершина $s \in V$.

Постусловие: каждая вершина, достижимая из s, помечается как «разведанная» и имеет присвоенное ей значение *scc*.

пометить s как разведанную

scc(s) := *numSCC* `// приведенная выше глобальная переменная`

for каждое ребро (s, v) в исходящем списке смежности s **do**

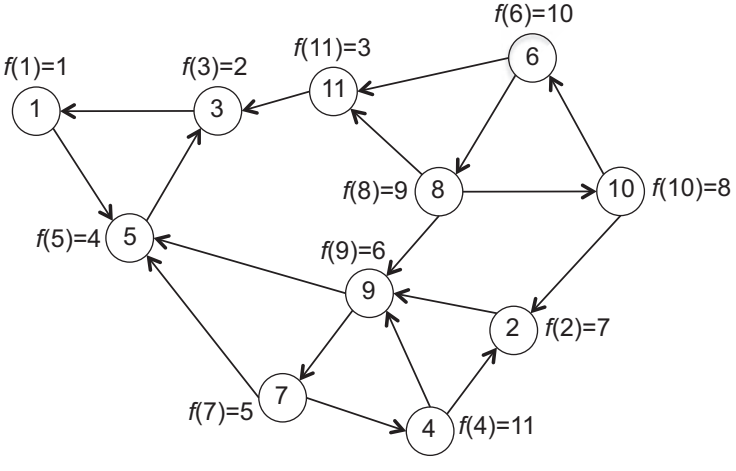
  if v не разведана **then**

    `DFS-SCC` (G, v)

8.6.5. Пример

Давайте проверим на нашем рабочем примере, что мы получаем то, что мы хотим, — что второй проход поиска в глубину обнаруживает сильно связные компоненты в обратном топологическом порядке. Предположим, что граф

на рис. 8.16 является развернутым в обратную сторону графом G^{rev} входного графа. В разделе 8.6.3 мы вычислили два способа, которыми алгоритм `TopoSort` мог бы назначать вершинам этого графа значения f; давайте применим первый. Вот входной (не обратный) граф, в котором вершины аннотированы этими вершинными позициями:



Второй проход перебирает вершины в порядке возрастания их позиции. Тем самым первый вызов подпрограммы `DFS-SCC` инициируется в вершине в первой позиции (которая оказывается вершиной 1); он обнаруживает вершины 1, 3 и 5 и отмечает их как вершины первой сильно связной компоненты. Алгоритм переходит к рассмотрению вершины во второй позиции (вершина 3); она уже была разведана первым вызовом подпрограммы `DFS-SCC` и пропускается. Вершина в третьей позиции (вершина 11) еще не обнаружена и является следующей стартовой точкой для подпрограммы `DFS-SCC`. Единственное исходящее ребро этой вершины направляется в уже разведанную вершину (вершину 3), поэтому 11 является единственным членом второй сильно связной компоненты. Алгоритм пропускает вершину в четвертой позиции (вершина 5 уже разведана) и затем инициирует подпрограмму `DFS-SCC` из вершины 7, то есть вершины в пятой позиции. Этот поиск обнаруживает вершины 2, 4, 7 и 9 (остальные исходящие ребра направлены в уже разведанную вершину 5) и классифицирует их как третью сильно связную компоненту. Алгоритм пропускает вершину 9, а затем вершину 2 и, наконец, вызывает подпрограмму

DFS-SCC из вершины 10, обнаруживая финальную сильно связную компоненту (содержащую вершины 6, 8 и 10).

8.6.6. Правильность и время выполнения

Алгоритм Косарайю является правильным и невероятно быстрым для каждого ориентированного графа, а не только в нашем примере.

Теорема 8.12. (Свойства алгоритма Косарайю.) *Для каждого ориентированного графа, $G = (V, E)$, представленного в виде списков смежности:*

а) по завершении алгоритма Косарайю для каждой пары v, w вершин $scc(v) = scc(w)$ тогда и только тогда, когда v и w принадлежат одной и той же сильно связной компоненте графа G;

б) время работы алгоритма Косарайю равно $O(m + n)$, где $m = |E|$ и $n = |V|$.

Мы уже рассмотрели все ингредиенты, необходимые для доказательства. Данный алгоритм может быть реализован с временем $O(m+n)$, с небольшим скрытым постоянным множителем, по обычным причинам. Каждый из двух этапов поиска в глубину выполняет постоянное число операций в расчете на вершину или ребро, и дополнительная служебная работа увеличивает время выполнения только на постоянный множитель.

Данный алгоритм также правильно вычисляет все сильно связные компоненты: всякий раз, когда он инициирует новый вызов подпрограммы DFS-SCC, данный алгоритм обнаруживает ровно одну новую сильно связную компоненту, которая является стоковой сильно связной компонентой относительно графа еще не разведанных вершин (то есть сильно связной компонентой, в которой все исходящие ребра ведут к уже разведанным вершинам)[1].

[1] В целях более формального доказательства рассмотрим вызов подпрограммы DFS-SCC со стартовой вершиной v, которая принадлежит сильно связной компоненте S. Из следствия 8.11 вытекает, что ориентированные пути из v могут достичь только таких сильно связных компонент, которые содержат по крайней мере одну вершину с присвоенной ей позицией, более ранней, чем у v. Поскольку алгоритм Косарайю обрабатывает вершины в порядке их позиций, все вершины в сильно связных ком-

8.6.7. Решения тестовых заданий 8.5–8.6

Решение тестового задания 8.5

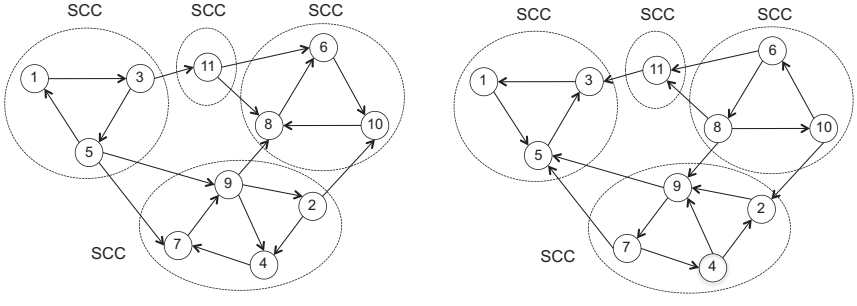
Правильный ответ: (г). В ориентированном ациклическом графе $G = (V, E)$ каждая вершина находится в своей собственной сильно связной компоненте (в общей $n = |V|$ сильно связных компонент). Чтобы увидеть это, зададим топологическое упорядочение графа G (раздел 8.5.1), причем каждой вершине $v \in V$ назначается несовпадающая метка $f(v)$. (Такая существует по теореме 8.6.) Ребра графа G направлены только от меньшего к большему значению f, поэтому для каждой пары $v, w \in V$ вершин в G либо нет пути $v \rightsquigarrow w$ (если $f(v) > f(w)$), либо нет пути $w \rightsquigarrow v$ (если $f(w) > f(v)$). Это не дает двум вершинам обитать в одной и той же сильно связной компоненте.

Решение тестового задания 8.6

Правильные ответы: (б), (г). Две вершины v, w ориентированного графа находятся в той же самой сильно связной компоненте тогда и только тогда, когда существует ориентированный путь P_1 из v в w и ориентированный путь P_2 из w в v. Это свойство соблюдается для v и w в G тогда и только тогда, когда оно соблюдается в G^{rev} — в последнем, с использованием развернутой в обратную сторону версии P_1, для того чтобы добраться из w в v, и развернутой в обратную сторону версии P_2, для того чтобы добраться из v в w. Мы можем заключить, что сильно связные компоненты графов G и G^{rev} являются совершенно одинаковыми. Истоковые сильно связные компоненты графа G (без входящих ребер) становятся стоковыми сильно связными компонентами графа G^{rev} (без исходящих ребер), и стоковые сильно связные компоненты становятся истоковыми сильно

понентах, достижимых из v, уже были разведаны алгоритмом. (Вспомните, что как только алгоритм находит одну вершину из сильно связной компоненты, он находит их все.) Таким образом, ребра, выходящие из S, достигают только уже разведанных вершин. Этот вызов подпрограммы `DFS-SCC` обнаруживает вершины компоненты S и ничего больше, так как для него нет доступных путей для проникновения в другие сильно связные компоненты. Поскольку каждый вызов подпрограммы `DFS-SCC` обнаруживает одну сильно связную компоненту и каждая вершина в конечном счете рассматривается, алгоритм Косарайю правильно идентифицирует все сильно связные компоненты.

связными компонентами. В более общем плане существует ребро из вершины в сильно связной компоненте S_1 в вершину сильно связной компоненты S_2 в G тогда и только тогда, когда существует соответствующее ребро из вершины компоненты S_2 в вершину компоненты S_1 в G^{rev} (рис. 8.17)[1].



а) Исходный граф *б*) Обратный граф

Рис. 8.17. Граф и его разворот в обратную сторону имеют одинаковые сильно связные компоненты

8.7. Структура Всемирной паутины

Теперь вы знаете коллекцию бесплатных графовых примитивов. Если у вас есть графовые данные, то вы можете применить эти быстрые алгоритмы, даже если не уверены, как будете использовать результаты. Например, в случае с ориентированным графом, почему бы не вычислить его сильно связные компоненты, чтобы получить представление о том, как он выглядит? Далее мы разведаем эту идею в огромном и чрезвычайно интересном ориентированном графе — графе Всемирной паутины, или *веб-графе*.

8.7.1. Веб-граф

В веб-графе вершины соответствуют веб-страницам, а ребра — гиперссылкам. Этот граф является ориентированным, его ребро указывает со страницы,

[1] Другими словами, метаграф G^{rev} (утверждение 8.9) просто-напросто является метаграфом G, в котором все ребра развернуты в обратную сторону.

содержащей ссылку, на страницу, куда будет перенаправлен пользователь после клика. Например, моя домашняя страница соответствует вершине в этом графе, а исходящие ребра соответствуют ссылкам на страницы со списком моих книг, моих курсов и так далее. Есть также входящие ребра, соответствующие ссылкам на мою домашнюю страницу, возможно, от моих соавторов или списков преподавателей онлайновых курсов (рис. 8.18).

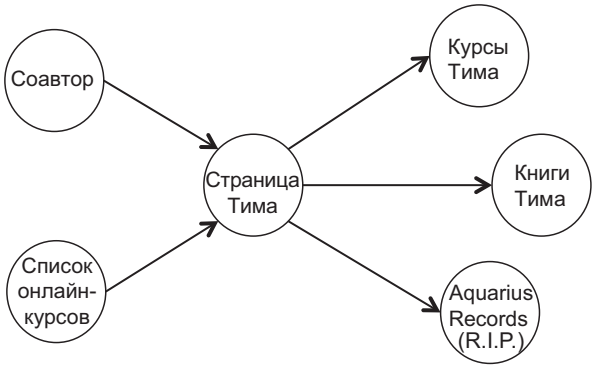


Рис. 8.18. Мизерная часть веб-графа

Истоки Всемирной паутины датируются примерно 90-ми годами, но взрывной рост она испытала около пяти лет спустя. К 2000 году (все еще «каменный век» в интернетовском летоисчислении) веб-граф был уже настолько велик, что не поддавался представлению, и исследователи были очень заинтересованы в понимании его структуры[1]. В этом разделе описывается известное исследование того времени, в котором изучалась структура веб-графа путем вычисления его сильно связных компонент[2]. Граф имел более 200 миллионов

---

[1] Для того чтобы построить этот граф, требуется выполнить обход (большого куска) Паутины, многократно следуя по гиперссылкам, и эта задача сама по себе представляет значительный инженерный подвиг.

[2] Это исследование описано в очень удобочитаемой статье *«Graph Structure in the Web»* («Структура графа в Сети») *Андрея Бродера*, *Рави Кумара*, *Фарзина Магуля*, *Прабхакара Рагхавана*, *Шридхара Раджагопалана*, *Райми Стата*, *Эндрю Томкинса* и *Джанет Винер* (*Computer Networks, 2000*). В то время компания Google только-только появилась, и в данном исследовании использовались данные веб-обходов поисковой системой Alta Vista (которой уже давно не существует).

вершин и 1,5 миллиарда ребер, поэтому линейно-временные алгоритмы были абсолютно необходимы![1]

8.7.2. «Галстук-бабочка»

В исследовании Бродера и соавторов были вычислены сильно связные компоненты веб-графа и приведены объяснения полученных результатов с использованием «галстука-бабочки», изображенного на рис. 8.19. Узел «галстука-бабочки» является крупнейшим сильно связным компонентом графа, содержащим примерно 28 % его вершин. Эта сильно связная компонента называется гигантской вполне заслуженно, так как следующая по величине сильно связная компонента была на два порядка меньше[2]. Гигантская сильно связная компонента может быть истолкована как ядро сети, каждая страница которого доступна с любой другой страницы по последовательности гиперссылок.

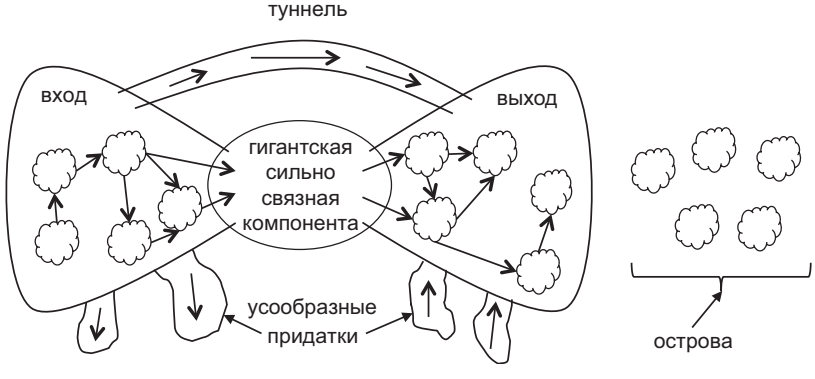


Рис. 8.19. Визуализация веб-графа в виде «галстука-бабочки». Примерно одинаковое число веб-страниц принадлежит гигантской сильно связной компоненте, входу, выходу и остальной части графа

---

[1] Данное исследование предвосхищает современные платформы обработки массивных данных, такие как MapReduce и Hadoop, и в то время это был отпугивающий размер вводных данных.

[2] Помните: чтобы сложить две сильно связных компоненты в одну требуется лишь одно ребро в каждом направлении. Интуитивно было бы довольно странно, если бы было две массивных сильно связных компоненты без проходящих между ними, по крайней мере в одном направлении, ребер.

Более мелкие сильно связные компоненты можно отнести к нескольким категориям. Из некоторых можно добраться до гигантской сильно связной компоненты (но не наоборот); это левая (входная) часть «галстука-бабочки». Например, в этой части появится вновь созданная веб-страница со ссылкой на какую-либо страницу в гигантской сильно связной компоненте. Аналогичным образом выходная часть представляет собой все сильно связные компоненты, достижимые из гигантской сильно связной компоненты, но не наоборот. Одним из примеров сильно связной компоненты в этой части является корпоративный веб-сайт, для которого политика компании предписывает, чтобы все гиперссылки с его страниц оставались в пределах сайта. Есть и другие странные вещи: «туннели», проходящие от входа к выходу, минуя гигантскую сильно связную компоненту; «усы», которые доступны из входа или которые могут достигать выхода (но не принадлежащие гигантской сильно связной компоненте), и «острова» веб-страниц, которые не могут достигать или быть достигнуты почти из любой другой части Сети.

8.7.3. Основные выводы

Возможно, самым удивительным открытием данного исследования является то, что все основные части веб-графа: гигантская сильно связная компонента, входная часть, выходная часть и странные вещи имеют примерно одинаковый размер (≈ 24–28 % вершин каждая). До этого исследования многие ожидали, что гигантская сильно связная компонента будет намного больше, чем просто 28 % Паутины. Второй интересный вывод заключается в том, что гигантская сильно связная компонента внутренне обильно связана: она имеет примерно 56 миллионов веб-страниц, но для перехода от одной страницы к другой нужно, как правило, перейти не менее чем по 20 гиперссылкам[1]. Остальная часть веб-графа связана слабее, где для перехода от одной вершины к другой часто необходимы длинные пути.

Вы были бы правы, если бы задались вопросом, являются ли какие-либо из этих находок пережитком ныне доисторического снимка веб-графа, который

[1] Наличие коротких вездесущих путей также известно под названием «свойства малого мира», которое тесно связано с популярной фразой «шесть степеней разделения» (см. *Теория шести рукопожатий*).

был задействован в эксперименте. Хотя с течением времени точные цифры менялись, по мере того как веб-граф рос и эволюционировал, более поздние дополнительные исследования с переоценкой его структуры свидетельствуют о том, что качественные выводы Бродера и его соавторов остаются точными[1].

ВЫВОДЫ

- ✶ Поиск в ширину (BFS) разведывает граф осторожно, послойно.
- ✶ Поиск в ширину может быть реализован в линейном времени с использованием структуры данных, именуемой очередью.
- ✶ Поиск в ширину может использоваться для вычисления длин кратчайших путей между стартовой вершиной и всеми другими вершинами за линейное время.
- ✶ Связная компонента неориентированного графа является максимальным подмножеством вершин, так что между каждой парой его вершин существует путь.
- ✶ Эффективный алгоритм графового поиска, такой как поиск в ширину, может использоваться для вычисления связных компонент неориентированного графа за линейное время.
- ✶ Поиск в глубину (DFS) разведывает граф агрессивно, отступая назад только при необходимости.
- ✶ Поиск в глубину может быть реализован в линейном времени с использованием структуры данных, именуемой стеком (либо с использованием рекурсии).

[1] Относительно веб-графа и других информационных сетей продолжается много интересных исследований; например, в отношении того, как веб-граф эволюционирует с течением времени, по динамике распространения информации через такой граф и в отношении того, как идентифицировать «сообщества» или другую значимую мелкозернистую структуру. В большинстве этих исследований невероятно быстрые графовые примитивы играют решающую роль. В качестве введения в эти темы рекомендуем ознакомиться с учебником *«Networks, Crowds, and Markets: Reasoning About a Highly Connected World»* («Сети, толпы и рынки: рассуждения о сильно связном мире») *Дэвида Исли* и *Джона Клейнберга* (*Cambridge University Press, 2010*).

- ✶ Топологическое упорядочивание ориентированного графа назначает вершинам разные числа, причем каждое ребро направлено от меньшего числа к большему.
- ✶ Ориентированный граф имеет топологическое упорядочение тогда и только тогда, когда он является ориентированным ациклическим графом.
- ✶ Поиск в глубину может быть использован для вычисления топологического упорядочивания ориентированного ациклического графа за линейное время.
- ✶ Сильно связная компонента ориентированного графа является максимальным подмножеством вершин, так что существует ориентированный путь из любой вершины в множестве в любую другую вершину в множестве.
- ✶ Поиск в глубину может использоваться для вычисления сильно связных компонент ориентированного графа за линейное время.
- ✶ В веб-графе гигантская сильно связная компонента содержит примерно 28 % вершин и внутренне обильно связана.

Задача 8.1. Какое из следующих утверждений соблюдается? Как обычно, n и m обозначают число вершин и ребер графа соответственно. (Выберите все подходящие варианты.)

а) Поиск в ширину может использоваться для вычисления связных компонент неориентированного графа за время $O(m + n)$.

б) Поиск в ширину может использоваться для вычисления длин кратчайших путей из стартовой вершины до любой другой вершины за время $O(m + n)$, где «кратчайший» означает наименьшее число ребер.

в) Поиск в глубину может использоваться для вычисления сильно связных компонент ориентированного графа за время $O(m + n)$.

г) Поиск в глубину может использоваться для вычисления топологического упорядочивания ориентированного ациклического графа за время $O(m + n)$.

Задача 8.2. Каково время выполнения поиска в глубину как функции n и m (числа вершин и ребер), если входной граф представлен матрицей смежности (а не списками смежности)? Вы можете исходить из того, что граф не имеет параллельных ребер.

а) $\Theta(m + n)$

б) $\Theta(m + n \log n)$

в) $\Theta(n^2)$

г) $\Theta(m \times n)$

Задача 8.3. В данной задаче исследуется связь между двумя определениями расстояний между графами. В этой задаче рассматриваются только неориентированные и связные графы. *Диаметр* графа — это максимальное кратчайшее расстояние между v и w над всеми вариантами вершин v и w[1]. Далее, для вершины v, обозначим через $l(v)$ максимальное кратчайшее расстояние между v и w над всеми вершинами w. *Радиус* графа — это минимальное значение $l(v)$ над всеми вариантами вершины v.

Какое из следующих неравенств, связывающих радиус r с диаметром d, содержится в каждом неориентированном связном графе? (Выберите все подходящие варианты.)

а) $r \leq d/2$

б) $r \leq d$

в) $r \geq d/2$

г) $r \geq d$

Задача 8.4. Когда ориентированный граф имеет уникальное топологическое упорядочение?

а) Когда он является ориентированным ациклическим.

б) Когда он имеет уникальный цикл.

в) Всякий раз, когда он содержит ориентированный путь, который посещает каждую вершину ровно один раз.

г) Ни один из представленных вариантов не является правильным.

---

[1] Напомним, что кратчайшее расстояние между v и w — это наименьшее число ребер в пути v-w.

Задача 8.5. Рассмотрите выполнение алгоритма `TopoSort` (раздел 8.5) на ориентированном графе G, который не является ориентированным ациклическим. Данный алгоритм не будет вычислять топологическое упорядочивание (поскольку оно отсутствует). Вычисляет ли он упорядочивание, минимизирующее число ребер, которые направлены назад (рис. 8.20)? (Выберите все подходящие варианты.)

а) Алгоритм `TopoSort` всегда вычисляет упорядочивание вершин, которое минимизирует число обратных ребер.

б) Алгоритм `TopoSort` никогда не вычисляет упорядочивание вершин, которое минимизирует число обратных ребер.

в) Существуют примеры, в которых алгоритм `TopoSort` вычисляет упорядочивание вершин, которое минимизирует число обратных ребер, а также примеры, в которых он этого не делает.

г) Алгоритм `TopoSort` вычисляет упорядочивание вершин, которое минимизирует число обратных ребер тогда и только тогда, когда входной граф представляет собой ориентированный цикл.

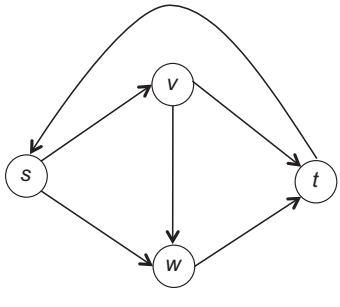


Рис. 8.20. Граф без топологического упорядочивания. В упорядочивании s, v, w, t единственным обратным ребром является (t, s)

Задача 8.6. Если добавить одно новое ребро в ориентированный граф G, то число сильно связных компонент... (Выберите все подходящие варианты.)

а) ...может или не может остаться тем же самым (в зависимости от G и нового ребра).

б) ...не может уменьшиться.

в) ...не может увеличиться.

г) ...не может уменьшиться более чем на 1.

Задача 8.7. Вспомните алгоритм `Kosaraju` из раздела 8.6, который использует два прохода поиска в глубину для вычисления сильно связных компонент ориентированного графа. Какие из следующих утверждений являются истинными? (Выберите все подходящие варианты.)

а) Алгоритм оставался бы правильным, если бы он использовал поиск в ширину вместо поиска в глубину в обоих своих проходах.

б) Алгоритм остался бы правильным, если бы мы использовали поиск в ширину вместо поиска в глубину в первом проходе.

в) Алгоритм остался бы правильным, если бы мы использовали поиск в ширину вместо поиска в глубину во втором проходе.

г) Алгоритм не является правильным, если только он не использует поиск в глубину в обоих своих проходах.

Задача 8.8. Напомним, что в алгоритме `Kosaraju` первый проход поиска в глубину работает на развернутой в обратную сторону версии входного графа, и второй проход — на исходном входном графе. Какие из следующих утверждений являются истинными? (Выберите все подходящие варианты.)

а) Алгоритм остался бы правильным, если бы в первом проходе ему присвоили позиции вершин в возрастающем (а не убывающем) порядке, а во втором проходе рассматривали вершины в убывающем (а не возрастающем) порядке позиций вершин.

б) Алгоритм остался бы правильным, если бы он использовал исходный входной граф в своем первом проходе и обратный граф в своем втором проходе.

в) Алгоритм остался бы правильным, если бы он использовал исходный входной граф в обоих проходах, при условии что в первом проходе он присвоил позиции вершин в возрастающем (а не убывающем) порядке.

г) Алгоритм остался бы правильным, если бы он использовал исходный входной граф в обоих проходах, при условии что во втором проходе он рассматривал вершины в порядке убывания (а не увеличения) положения вершин.

Задача повышенной сложности

Задача 8.9. В задаче *2SAT*[1] задано множество выражений (дизъюнктивных одночленов), каждое из которых является дизъюнкцией (логическим «или») двух литералов. (Литерал — это булева переменная или отрицание булевой переменной.) Вы хотите присвоить каждой переменной значение «истина» либо «ложь», для того чтобы все выражения были удовлетворены по крайней мере одним истинным литералом в каждом выражении. Например, если входные данные содержат три выражения $x_1 \vee x_2$, $\neg x_1 \vee x_3$ и $\neg x_2 \vee \neg x_3$, то один из способов удовлетворить все из них — установить x_1 и x_3 равными «истина» и x_2 равным «ложь»[2]. Из семи других возможных присваиваний истинностных значений только одно удовлетворяет всем трем выражениям.

Разработайте алгоритм, который определяет, имеет ли данный экземпляр задачи *2SAT* хотя бы одно удовлетворяющее присваивание. (Ваш алгоритм отвечает только за принятие решения о том, существует или нет удовлетворяющее его присваивание; само такое присваивание демонстрировать не обязательно.) Ваш алгоритм должен выполняться за время $O(m + n)$, где m и n — это число выражений и переменных соответственно.

[Подсказка: покажите, как решить задачу, вычисляя сильно связные компоненты надлежащим образом определенного ориентированного графа.]

Задача по программированию

Задача 8.10. Реализуйте на своем любимом языке программирования алгоритм `Kosaraju` из раздела 8.6 и используйте его для вычисления размеров пяти самых больших сильно связных компонент разных ориентированных графов. Вы можете реализовать итеративную версию поиска в глубину, рекурсивную версию (см. сноску на с. 43) либо и то и другое. (Обратитесь к сайту www.algorithmsilluminated.org для получения тестовых случаев и совокупностей данных для задач.)

---

[1] Относительно задач *2SAT* см. http://neerc.ifmo.ru/wiki/index.php?title=2SAT. — *Примеч. пер.*

[2] Символ «$\vee$» обозначает логическую операцию «или», а «$\neg$» — отрицание булевой переменной.

9

Алгоритм кратчайшего пути Дейкстры

Мы добрались до еще одного из величайших хитов computer science: алгоритма кратчайшего пути Дейкстры[1]. Этот алгоритм работает в любом ориентированном графе с неотрицательными длинами ребер и вычисляет длины кратчайших путей из стартовой вершины до всех остальных вершин. После формального определения задачи (раздел 9.1) мы даем описание данного алгоритма (раздел 9.2), доказательство его правильности (раздел 9.3) и приводим его простую реализацию (раздел 9.4). В следующей главе мы увидим невероятно быструю реализацию данного алгоритма, в котором используется преимущество структуры данных, именуемой кучей.

9.1. Задача о кратчайшем пути с единственным истоком

9.1.1. Определение задачи

Алгоритм Дейкстры решает задачу о кратчайшем пути с единственным истоком[2].

ЗАДАЧА: КОРОТКИЕ ПУТИ С ЕДИНСТВЕННЫМ ИСТОКОМ

Вход: ориентированный граф $G = (V, E)$, стартовая вершина $s \in V$ и неотрицательная длина l_e для каждого ребра $e \in E$.

Выход: $dist\ (s, v)$ для каждой вершины $v \in V$.

[1] Обнаружен Эдсгером В. Дейкстрой в 1956 году («примерно за двадцать минут», по его словам в интервью много лет спустя). Несколько других исследователей независимо обнаружили аналогичные алгоритмы в конце 1950 годов.

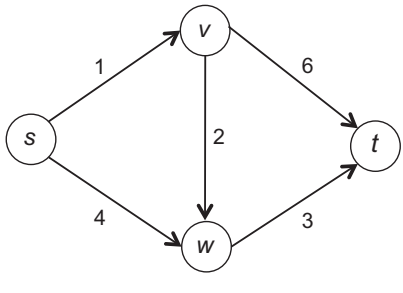
[2] Термин «исток» в названии задачи относится к заданной стартовой вершине. Мы уже использовали термин «истоковая вершина» для обозначения вершины ориентированного графа без входящих ребер (раздел 8.5.2). Для того чтобы остаться последовательными с нашей терминологией из главы 8, мы будем придерживаться термина «стартовая вершина».

Напомним, что форма записи *dist*(*s*, *v*) обозначает длину кратчайшего пути из *s* к *v*. (Если пути из *s* в *v* нет, то *dist*(*s*, *v*) равно +∞.) Под длиной пути мы подразумеваем сумму длин его ребер. Так, в графе, в котором каждое ребро имеет длину 1, длина пути — это количество ребер в нем. Кратчайший путь из вершины *v* до вершины *w* — это путь с минимальной длиной (среди всех путей *v*-*w*).

Например, если граф представляет дорожную сеть, а длина каждого ребра представляет ожидаемое время в пути от одного конца до другого, то задача о кратчайшем пути с единственным истоком представляет собой задачу вычисления времени в пути из исходной точки (стартовой вершины) во все возможные финальные вершины.

ТЕСТОВОЕ ЗАДАНИЕ 9.1

Рассмотрим следующий входной граф для задачи о кратчайшем пути с единственным истоком, со стартовой вершиной *s* и в котором каждое ребро помечено его длиной:



Каковы расстояния кратчайшего пути соответственно до *s*, *v*, *w* и *t*?

а) 0, 1, 2, 3

б) 0, 1, 3, 6

в) 0, 1, 4, 6

г) 0, 1, 4, 7

(Решение и пояснение см. в разделе 9.1.4.)

9.1.2. Несколько допущений

Для конкретности в этой главе мы исходим из допущения, что входной граф является ориентированным. Алгоритм Дейкстры одинаково хорошо применим к неориентированным графам после небольших косметических изменений (как вы сами можете убедиться).

Другое наше допущение является значительным и прозвучало оно уже в постановке задачи: мы предполагаем, что длина каждого ребра является неотрицательной. Во многих приложениях, таких как вычисление маршрутов движения, длины ребер автоматически являются неотрицательными (за исключением машины времени), и беспокоиться не о чем. Но помните, что пути в графе могут представлять абстрактные последовательности решений. Возможно, вы захотите вычислить прибыльную последовательность финансовых транзакций, которая включает в себя как покупку, так и продажу. Эта задача соответствует нахождению кратчайшего пути в графе с длинами ребер как положительными, так и отрицательными. Вам *не* следует использовать алгоритм Дейкстры в приложениях с отрицательной длиной ребер; см. также раздел 9.3.1[1].

9.1.3. Почему не поиск в ширину?

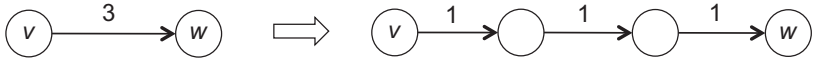
В разделе 8.2 мы увидели, что одним из превалирующих применений поиска в ширину является вычисление расстояний кратчайшего пути из стартовой вершины. Зачем нужен еще один алгоритм кратчайшего пути?

Напомним, что поиск в ширину вычисляет минимальное число *ребер* в пути от стартовой вершины до любой другой вершины. Это частный случай задачи о кратчайшем пути с единственным истоком, в которой каждое ребро имеет длину 1. В тестовом задании 9.1 мы увидели, что при общих неотрицательных длинах ребер кратчайший путь не обязательно должен быть путем с наименьшим числом ребер. Многие применения кратчайших путей, такие как вычисление маршрутов движения или последовательности финансовых транзакций, с неизбежностью предусматривают наличие ребер разной длины.

---

[1] В *третьей части* нашей серии мы узнаем об эффективных алгоритмах для более общей задачи о кратчайшем пути с единственным истоком, в которой разрешены отрицательные длины ребер, включая знаменитый алгоритм Беллмана—Форда.

Но постойте, возразите вы, а действительно ли общая проблема так отличается от этого особого случая? Разве мы не можем просто подумать о ребре с большей длиной как о пути из l ребер, каждое из которых имеет длину 1?:



И действительно, нет принципиальной разницы между ребром с положительной интегральной длиной l и путем из l ребер длины 1. В принципе, вы можете решить задачу о кратчайшем пути с единственным истоком, развернув ребра в путь из ребер длины 1 и применив к расширенному графу поиск в ширину.

Выше приведен пример *редукции* одной задачи к другой — в данном случае задачи о кратчайшем пути с единственным истоком и с положительными целочисленными длинами ребер к частному случаю данной задачи, в которой каждое ребро имеет длину 1.

Главная проблема с этой редукцией заключается в том, что она приводит к взрывному росту размер графа. Взрывной рост не так уж плох, если все длины ребер являются малыми целыми числами, но в реальных приложениях это не всегда так. Длина ребра может быть даже намного больше, чем число вершин и ребер в исходном графе! Поиск в ширину будет выполняться во времени, линейном по размеру развернутого графа, но это не обязательно будет близко к линейному времени по размеру исходного графа. Алгоритм Дейкстры можно рассматривать как гладкое моделирование поиска в ширину на расширенном графе, работая только с исходным входным графом почти за линейное время.

О РЕДУКЦИЯХ

Задача А сводится (или *редуцируется*) к задаче Б, если алгоритм, решающий задачу Б, можно легко перевести в алгоритм, решающий задачу А. Например, задача вычисления медианного элемента массива сводится к задаче сортировки массива. Редукции являются одним из наиболее важных понятий в изучении алгоритмов и их ограничений, и они могут иметь большую практическую пользу.

Вы всегда должны быть в поиске редукций. Всякий раз, когда вы сталкиваетесь с, казалось бы, новой задачей, всегда спрашивайте: является ли эта задача замаскированной версией той, решение которой вы уже знаете?

9.1.4. Решение тестового задания 9.1

Правильный ответ: (б). Нетрудно догадаться, что кратчайшее расстояние из вершины s в саму себя равно 0 и из s в v равно 1. Вершина w — интереснее. Один путь s-w является прямым ребром (s, w), которое имеет длину 4. Но использование большего числа ребер может уменьшить общую длину: путь $s \to v \to w$ имеет длину всего 1 + 2 = 3 и является кратчайшим путем s-w. Схожим образом каждый из путей с двумя переходами из s в t имеет длину 7, в то время как зигзагообразный путь имеет длину всего 1 + 2 + 3 = 6.

9.2. Алгоритм Дейкстры

9.2.1. Псевдокод

Высокоуровневая структура алгоритма Дейкстры напоминает структуру наших алгоритмов графового поиска[1]. Каждая итерация ее главного цикла обрабатывает одну новую вершину. Изощренность алгоритма заключается в его умном правиле отбора следующей вершины для обработки — еще не обработанной вершины, которая выглядит самой близко расположенной к начальной вершине. Следующий элегантный псевдокод проясняет эту идею.

DIJKSTRA

Вход: ориентированный граф $G = (V, E)$, представленный в виде списков смежности, вершина $s \in V$, длина $l_e \geq 0$ для каждого $e \in E$.

Постусловие: для каждой вершины v значение $len(v)$ равно истинному кратчайшему расстоянию $dist(s, v)$.

```
// инициализация
```

1) $X := \{s\}$

2) $len(s) := 0$, $len(v) := +\infty$ для каждого $v \neq s$

```
// главный цикл
```

3) **while** существует ребро (v, w), где $v \in X$, $w \notin X$ **do**

[1] Когда все ребра имеют длину 1, это эквивалентно поиску в ширину (как вы можете убедиться сами).

4) (v^*, w^*) := такое ребро, которое минимизирует $len(v) + l_{vw}$

5) добавить w^* в X

6) $len(w^*) := len(v^*) + l_{v^*w^*}$

Множество X содержит вершины, с которыми алгоритм уже имел дело. Изначально X содержит только стартовую вершину (и, конечно, $len(s) = 0$), и множество разбухает как тесто, до тех пор пока оно не охватит все вершины, достижимые из s. Данный алгоритм присваивает конечное значение *len*-значению вершины в то же время, когда он добавляет вершину в X. Каждая итерация главного цикла наращивает X на одну новую вершину, голову некоторого ребра (v, w), переходящего из X в $V - X$ (рис. 9.1). (Если такого ребра нет, то алгоритм останавливается, причем $len(v) = +\infty$ для всех $v \notin X$.) Таких ребер может быть много; алгоритм `Dijkstra` выбирает одно (v^*, w^*), которое минимизирует *дейкстрову балльную оценку*, определяемую как

$$len(v) + l_{vw}. \tag{9.1}$$

Обратите внимание, что дейкстровы балльные оценки определяются на *ребрах* — вершина $w \notin X$ может быть головой многих разных ребер, переходящих из X в $V - X$, и эти ребра обычно имеют разные дейкстровы оценки.

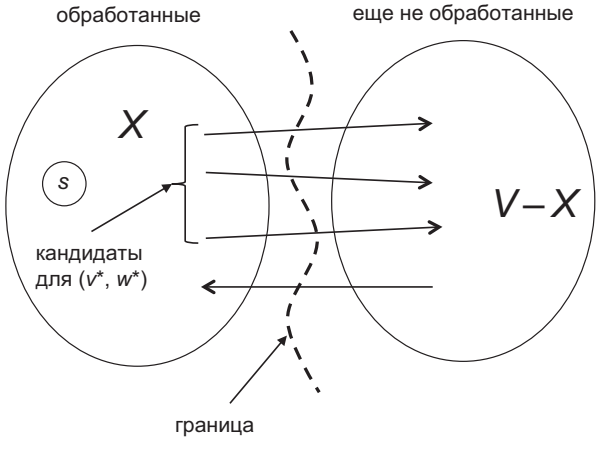
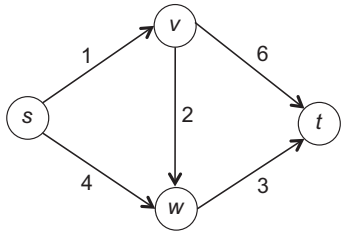


Рис. 9.1. Каждая итерация алгоритма Дейкстры обрабатывает одну новую вершину, голову ребра, переходящего из X в $V - X$

Вы можете связать дейкстрову оценку для ребра (v, w), где $v \in X$ и $w \notin X$, с гипотезой, что кратчайший путь из s в w состоит из кратчайшего пути из s в v (который, хочется верить, имеет длину $len(v)$) с ребром (v, w) (которое имеет длину l_{vw}), присоединенным в конце. Таким образом, алгоритм `Dijkstra` решает добавить еще не обработанную вершину, которая выглядит самой близкой к s в соответствии с уже вычисленными расстояниями кратчайшего пути и длинами ребер, переходящих из X в $V - X$. При добавлении w^* в X алгоритм назначает $len(w^*)$ его гипотетическому кратчайшему расстоянию из s, то есть дейкстровой оценке $len(v^*) + l_{v^*w^*}$ ребра (v^*, w^*). Магия алгоритма Дейкстры, формализованная в приведенной ниже теореме 9.1, заключается в том, что эта гипотеза гарантированно истинна, даже если алгоритм до этого рассмотрел только крошечную часть графа[1].

9.2.2. Пример

Попробуем алгоритм `Dijkstra` на примере из тестового задания 9.1:



Изначально множество X содержит только s, и $len(s) = 0$. На первой итерации главного цикла существует два ребра, переходящих из X в $V - X$ (и, следовательно, приемлемые для роли (v^*, w^*)) ребра (s, v) и (s, w). Дейкстровы оценки (определенные в п. 9.1) для этих двух ребер равны $len(s) + l_{sv} = 0 + 1 = 1$

[1] Для того чтобы вычислить сами кратчайшие пути (а не только их длины), свяжите указатель *предшественник* (v) с каждой вершиной $v \in V$. Когда ребро (v^*, w^*) выбирается в итерации главного цикла `while` (строки 4–6), назначьте указатель предшественник (w^*) вершине v^*, то есть вершине, ответственной за выбор w^*. Для того чтобы восстановить кратчайший путь из s в вершину v после завершения алгоритма, следуйте по указателям предшественника назад от v, до тех пор пока не достигнете s.

и $len(s) + l_{sw} = 0 + 4 = 4$. Поскольку первое ребро имеет меньшую оценку, его голова v добавляется в X, и $len(v)$ назначется дейкстровой оценке ребра (s, v), которая равна 1. На второй итерации с $X = \{s, v\}$ имеется три ребра для роли (v^*, w^*): (s, w), (v, w) и (v, t). Их дейкстровы оценки равны $0 + 4 = 4$, $1 + 2 = 3$ и $1 + 6 = 7$. Поскольку (v, w) имеет наименьшую дейкстрову оценку, w втягивается в X, и длине $len(w)$ назначается значение 3 (дейкстрова оценка ребра (v, w)). Мы уже знаем, какая вершина добавляется в X в финальной итерации (единственная еще не обработанная вершина t), но нам все равно нужно определить ребро, которое приводит к ее добавлению (вычислить $len(t)$). Поскольку ребра (v, t) и (w, t) имеют дейкстровы оценки соответственно $1 + 6 = 7$ и $3 + 3 = 6$, длина $len(t)$ устанавливается равной более низкой оценке 6. Множество X теперь содержит все вершины, поэтому никакие ребра не переходят из X в $V - X$, и алгоритм останавливается. Значения $len(s) = 0$, $len(v) = 1$, $len(w) = 3$ и $len(t) = 6$ соответствуют истинным кратчайшим расстояниям, которые мы определили в тестовом задании 9.1.

Конечно, тот факт, что алгоритм правильно работает на конкретном примере, *не* означает, что он является правильным в общем случае![1] По сути дела, алгоритм `Dijkstra` *не* обязательно вычисляет правильные расстояния кратчайшего пути, когда ребра могут иметь отрицательные длины (раздел 9.3.1). Вы должны изначально скептически относиться к алгоритму `Dijkstra` и требовать доказательства того, что, по крайней мере, в графах с неотрицательными реберными длинами он правильно решает задачу о кратчайшем пути с единственным истоком.

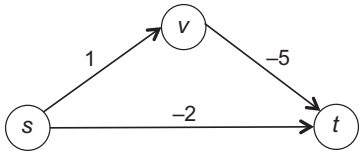
*9.3. Почему алгоритм Дейкстры правилен?

9.3.1. Фиктивная редукция

Вы, возможно, задаетесь вопросом, почему так важно, имеют или нет ребра отрицательные реберные длины. Разве мы не можем просто сделать так, чтобы все реберные длины были неотрицательными, добавив большое число к длине каждого ребра?

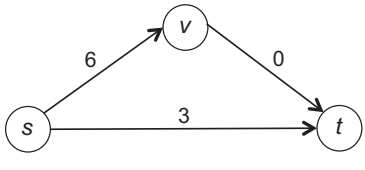
[1] Даже сломанные аналоговые часы верны два раза в сутки.

Это отличный вопрос — вы всегда должны быть начеку, пытаясь редуцировать задачи, решения которых вы уже знаете. Увы, таким образом вы не можете свести задачу о кратчайшем пути с единственным истоком с общими реберными длинами к частному случаю неотрицательных реберных длин. Проблема в том, что разные пути из одной вершины в другую могут иметь разное число ребер. Если добавить некоторое число к длине каждого ребра, то длины разных путей могут увеличиться на разные величины, и кратчайший путь в новом графе может отличаться от исходного графа. Вот простой пример:



Существует два пути из s в t: прямой путь (который имеет длину −2) и путь с двумя переходами $s \rightarrow v \rightarrow t$ (который имеет длину $1 + (-5) = -4$). Последний имеет меньшую (то есть более отрицательную) длину и является кратчайшим путем s-t.

Для того чтобы граф заимел неотрицательные реберные длины, мы могли бы добавить 5 к длине каждого ребра:



Кратчайший путь из s в t поменялся и теперь является прямым ребром s-t (который имеет длину 3 — лучше, чем альтернатива из 6). Выполнение алгоритма кратчайшего пути на преобразованном графе не даст правильного ответа для исходного графа.

9.3.2. Плохой пример алгоритма Дейкстры

Что произойдет, если мы попробуем выполнить алгоритм `Dijkstra` непосредственно на графе с отрицательными реберными длинами, как на приведенном

выше графе? Как всегда, изначально $X = \{s\}$ и $len(s) = 0$, и все это прекрасно. Однако на первой итерации главного цикла алгоритм вычисляет дейкстровы оценки ребер $\{s, v\}$ и (s, t), то есть $len(s) + l_{sv} = 0 + 1 = 1$ и $len(s) + l_{st} = 0 + (-2) = -2$. Последнее ребро имеет меньшую оценку, и поэтому алгоритм добавляет вершину t в X и назначает $len(t)$ оценке -2. Как мы уже отмечали, фактический кратчайший путь из s в t (путь $s \to v \to t$) имеет длину -4. Мы заключаем, что алгоритм `Dijkstra` не обязательно вычисляет правильные расстояния кратчайшего пути при наличии отрицательных реберных длин.

9.3.3. Правильность с неотрицательными реберными длинами

Доказательства правильности могут казаться довольно занудными. Вот почему я часто их «затушевываю» для алгоритмов, в отношении которых студенты, как правило, имеют сильное и точное интуитивное понимание. Алгоритм Дейкстры отличается. Во-первых, тот факт, что он не работает на чрезвычайно простых графах с отрицательными реберными длинами (раздел 9.3.1), должен заставить вас нервничать. Во-вторых, дейкстрова оценка (9.1) может показаться загадочной или даже необоснованной — что в ней такого важного? Из-за этих сомнений и потому, что это такой фундаментальный алгоритм, мы специально потратим время на то, чтобы тщательно доказать его правильность (в графах с неотрицательными реберными длинами).

Теорема 9.1. (Правильность алгоритма `Dijkstra`.) *Для каждого ориентированного графа $G = (V, E)$, каждой стартовой вершины s и каждого варианта неотрицательных реберных длин по завершении алгоритма* `Dijkstra` *$len(v) = dist(s, v)$ для каждой вершины $v \in V$.*

Индукционное отклонение

План состоит в том, чтобы обосновать расстояния кратчайшего пути, вычисляемые алгоритмом `Dijkstra` одно за другим, по индукции на числе итераций его главного цикла. Напомним, что доказательства по индукции подчиняются довольно жесткому шаблону, цель которого — зафиксировать, что логическое утверждение $P(k)$ соблюдается для любого положительного целого числа k. В доказательстве теоремы 9.1 мы определим $P(k)$ следующим формальным

суждением: для k-й вершины v, добавленной в множество X в алгоритме `Dijkstra`, $len(v) = dist(s, v)$.

Аналогично рекурсивному алгоритму, доказательство по индукции состоит из двух частей: *базового случая* и *индукционного шага*, так называемого индукционного перехода. Базовый случай прямо доказывает, что $P(1)$ является истинным. На индукционном шаге вы принимаете допущение, что все $P(1)$, ..., $P(k - 1)$ являются истинными — это называется индукционном гипотезой, — и используете это допущение, для того чтобы доказать, что $P(k)$, следовательно, тоже является истинным. Если вы доказываете и базовый случай, и индукционный шаг, то $P(k)$ действительно является истинным для любого положительного целого числа k. $P(1)$ является истинным по базовому случаю, и применение индукционного шага несколько раз подряд показывает, что $P(k)$ является истинным для сколь угодно больших значений k.

О ЧТЕНИИ ДОКАЗАТЕЛЬСТВ

Математические аргументы выводят заключения из допущений. При чтении доказательства всегда следует убедиться, что вы понимаете, каким образом каждое допущение используется в аргументе, и почему аргумент нарушается при отсутствии каждого допущения.

Имея это в виду, внимательно следите за ролью, которую играют в доказательстве теоремы 9.1 два ключевых допущения: что длины ребер являются неотрицательными и что алгоритм всегда выбирает ребро с наименьшей дейкстровой оценкой. Любое предпринятое доказательство теоремы 9.1, в котором не удается использовать оба этих допущения, автоматически является ошибочным.

Доказательство теоремы 9.1

Мы будем действовать по индукции, с $P(k)$, то есть логическим утверждением, что алгоритм `Dijkstra` правильно вычисляет расстояние кратчайшего пути k-й вершины, добавленной в множество X. В базовом случае ($k = 1$) мы знаем, что первая вершина, добавленная в X, является стартовой вершиной s. Алгоритм `Dijkstra` назначает 0 длине $len(s)$. Поскольку каждое ребро имеет неотрица-

тельную длину, кратчайший путь из вершины s в саму себя является пустым путем с длиной 0. Следовательно, $len(s) = 0 = dist(s, s)$, что доказывает $P(1)$.

На индукционном шаге выберем $k > 1$ и примем допущение, что все $P(1)$, ..., $P(k-1)$ являются истинными: что $len(v) = dist(s, v)$ для первых $k-1$ вершин v, добавленных алгоритмом `Dijkstra` в X. Пусть w^* обозначает k-ю вершину, добавленную в X, и пусть (v^*, w^*) обозначает ребро, выбранное на соответствующей итерации (где обязательно v^* уже находится в X). Алгоритм назначает $len(w^*)$ дейкстровой оценке этого ребра, а именно $len(v^*) + l_{v^*w^*}$. Мы надеемся, что это значение является тем же самым, что и истинное расстояние кратчайшего пути $dist(s, w^*)$, но так ли это?

Мы утверждаем в двух частях, что это так. Прежде всего, давайте докажем, что истинное расстояние $dist(s, w^*)$ может быть только меньше, чем предположение алгоритма $len(w^*)$, где $dist(s, w^*) \leq len(w^*)$. Поскольку v^* уже находилась в X, когда было выбрано ребро (v^*, w^*), эта вершина была одной из первых $k-1$ вершин, добавленных в X. По индукционной гипотезе алгоритм `Dijkstra` правильно вычислил расстояние кратчайшего пути вершины v^*: $len(v^*) = dist(s, v^*)$. В частности, существует путь P из s в v^* с длиной ровно $len(v^*)$. В результате прикрепления ребра (v^*, w^*) в конце P получается путь P^* из s в w^* с длиной $len(v^*) + l_{v^*w^*} = len(w^*)$ (рис. 9.2). Длина кратчайшего пути s-w^* не превышает длину потенциально возможного пути P^*, поэтому $dist(s, w^*)$ составляет не больше $len(w^*)$.

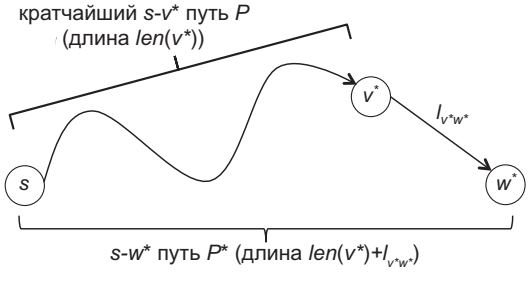


Рис. 9.2. Прикрепление ребра (v^*, w^*) в конце кратчайшего s-v^*-пути P производит s-w^*-путь P^* длины $len(v^*) + l_{v^*w^*}$

Теперь рассмотрим обратное неравенство, констатирующее, что $dist(s, w^*) \geq len(w^*)$ (и поэтому $len(w^*) = dist(s, w^*)$, как того бы хотелось). Другими

словами, давайте покажем, что путь P^* на рис. 9.2 действительно является кратчайшим путем s-w^* — длина каждого конкурирующего пути s-w^* составляет не меньше $len(w^*)$.

Зафиксируем конкурирующий s-w^*-путь P'. Мы очень мало знаем о P'. Вместе с тем мы все-таки имеем представление, что он берет начало в s и заканчивается в w^* и что в начале этой итерации вершина s, а не w^*, принадлежала множеству X. Поскольку путь P' начинается в X и заканчивается за пределами X, он пересекает границу между X и $V - X$ не менее одного раза (рис. 9.3); пусть (y, z) обозначает первое ребро P', которое пересекает эту границу (причем $y \in X$ и $z \notin X$)[1].

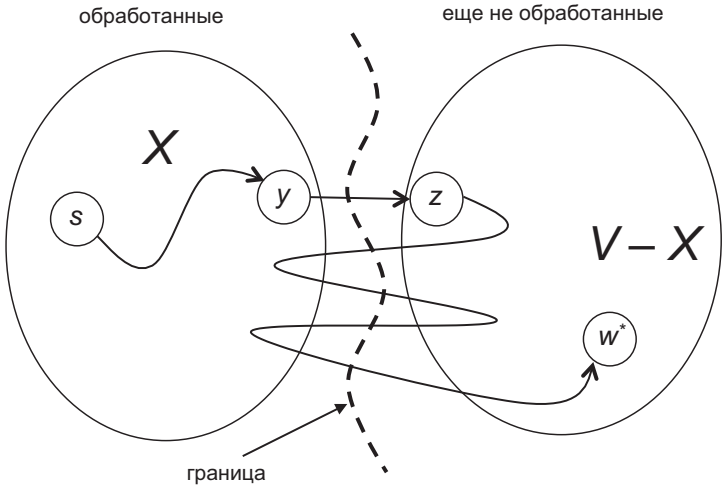
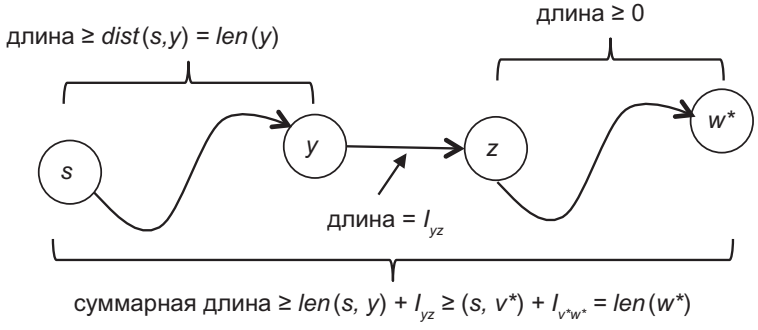


Рис. 9.3. Каждый путь s-w^* пересекает не менее одного раза из X в $V - X$

Для того чтобы доказать, что длина пути P' составляет не менее $len(w^*)$, мы рассматриваем три его части раздельно: первоначальную часть P', которая проходит из s в y, ребро (y, z) и финальную часть, которая проходит из z в w^*. Первая часть не может быть короче кратчайшего пути из s в y, поэтому его длина составляет не менее $dist(s, y)$. Длина ребра (y, z) равна l_{yz}. Мы не знаем

[1] Не нужно беспокоиться, если $y = s$ или $z = w^*$ — этот аргумент работает нормально, в чем вы можете убедиться сами.

многого о финальной части пути, затерянной среди вершин, которые алгоритм еще не просматривал. Но мы выяснили (поскольку все длины ребер являются неотрицательными!), что его суммарная длина составляет не меньше нуля:



Объединяя наши нижние границы длины трех частей пути P', мы имеем

$$\text{длина пути } P' \geq \underbrace{dist(s,y)}_{\text{подпуть } s-y} + \underbrace{l_{yz}}_{\text{ребро } (y,\,z)} + \underbrace{0}_{\text{подпуть } z-w^*}. \tag{9.2}$$

Последний пункт на повестке — связать нашу нижнюю границу длины в (9.2) с балльными дейкстровыми оценками, которые направляют решения алгоритма. Поскольку $y \in X$, эта вершина была одной из первых $k - 1$ вершин, добавленных в X, и из индукционной гипотезы вытекает, что алгоритм правильно вычислил расстояние своего кратчайшего пути: $dist(s, y) = len(y)$. Таким образом, неравенство (9.2) транслируется в

$$\text{длина пути } P' \geq \underbrace{len(y) + l_{yz}}_{\text{дейкстрова балльная оценка ребра } (y,\,z)}. \tag{9.3}$$

Правая сторона неравенства — это именно дейкстрова оценка ребра (y, z). Поскольку алгоритм всегда выбирает ребро с наименьшей дейкстровой оценкой, и поскольку в этой итерации он предпочел ребро (v^*, w^*) ребру (y, z), первое из двух имеет еще меньшую дейкстрову оценку: $len(v^*) + l_{v^*w^*} \leq len(y) + l_{yz}$. Постановка неравенства 9.3 даст нам желаемое:

$$\text{длина пути } P' \geq \underbrace{len(v^*) + l_{v^*w^*}}_{\text{дейкстрова балльная оценка ребра } (v^*,w^*)} = len(w^*).$$

Этим завершается вторая часть индукционного шага, и мы заключаем, что $len(v) = dist(s, v)$ для каждой вершины v, которая когда-либо будет добавлена в множество X.

Финальным аккордом станет рассмотрение вершины v, которая ни разу не добавлялась в X. Когда алгоритм закончил работу, $len(v) = +\infty$, и ни одно ребро не переходит из X в $V - X$. Это означает, что во входном графе не существует пути из s в v — такой путь в какой-то момент должен был бы пересечь эту границу — и, следовательно, мы также имеем $dist(s, v) = +\infty$. Мы заключаем, что алгоритм останавливается с $len(v) = dist(s, v)$ для каждой вершины v, независимо от того, была ли когда-либо добавлена вершина v в X. Это завершает наше доказательство! *Ч. т. д.*

9.4. Реализация и время выполнения

Алгоритм кратчайшего пути Дейкстры напоминает наши линейно-временные алгоритмы графового поиска в главе 8. Основная причина, по которой поиск в ширину и в глубину выполняется за линейное время (теоремы 8.2 и 8.5), заключается в том, что на решение, какую вершину разведывать дальше, они тратят только постоянное количество времени (удаляя вершину из начала очереди или стека). Вызывает обеспокоенность, что каждая итерация алгоритма Дейкстры должна идентифицировать ребро, пересекающее границу с наименьшей дейкстровой оценкой. Можем ли мы реализовать алгоритм, который работал бы за линейное время?

ТЕСТОВОЕ ЗАДАНИЕ 9.2

Какое из следующих времен выполнения лучше всего описывает простую реализацию алгоритма Дейкстры для графов, представленных в виде списков смежности? Как обычно, n и m обозначают соответственно число вершин и ребер входного графа.

а) $O(m + n)$

б) $O(m \log n)$

в) $O(n^2)$

г) $O(mn)$

(Решение и пояснение см. ниже.)

Правильный ответ: (г). Простая реализация отслеживает то, какие вершины находятся в X, связывая булеву переменную с каждой вершиной. Каждая итерация выполняет исчерпывающий поиск по всем ребрам, вычисляет дейкстрову оценку для каждого ребра с хвостом в X и головой вне X (за постоянное время на ребро) и возвращает пересекающее ребро с наименьшей оценкой (или правильно идентифицирует, что никакие пересекающие ребра не существуют). После не более чем $n - 1$ итераций алгоритм `Dijkstra` исчерпывает все новые вершины, которые добавляются в его множество X. Поскольку число итераций равно $O(n)$ и каждая занимает время $O(m)$, совокупное время выполнения равняется $O(mn)$.

Утверждение 9.2. (Время выполнения алгоритма `Dijkstra` (простого).) *Для каждого ориентированного графа $G = (V, E)$, каждой стартовой вершины s и каждого варианта неотрицательных реберных длин простая реализация алгоритма* `Dijkstra` *выполняется за время $O(mn)$, где $m = |E|$ и $n = |V|$.*

Время выполнения простой реализации хоть и является хорошим, но недостаточно. Реализация будет замечательно работать для графов, в которых число вершин исчисляется сотнями или несколькими тысячами, но заглохнет на значительно более крупных графах. Можем ли мы добиться лучшего? Поистине священным Граалем в проектировании алгоритмов является линейно-временной алгоритм (или близкий к нему), и это то, чего мы хотим для задачи о кратчайшем пути с единственным истоком. Такой алгоритм мог бы обрабатывать графы с миллионами вершин на ноутбуке.

Нам не нужен более быстрый *алгоритм*, который достигал бы почти линейно-временного решения задачи. Нам просто нужна более качественная *реализация* алгоритма Дейкстры. Структуры данных (очереди и стеки) сыграли решающую роль в наших линейно-временных реализациях поиска в ширину и в глубину; схожим образом алгоритм Дейкстры может быть реализован почти в линейном времени с помощью правильной структуры

данных для облегчения повторных вычислений минимума в его главном цикле. Эта структура данных называется *кучей* и является предметом следующей главы.

ВЫВОДЫ

- ✶ В задаче о кратчайшем пути с единственным истоком входные данные состоят из графа, стартовой вершины и длины каждого ребра. Цель состоит в том, чтобы вычислить длину кратчайшего пути из стартовой вершины до всех остальных вершин.
- ✶ Алгоритм Дейкстры обрабатывает вершины одну за другой, всегда выбирая еще не обработанную вершину, которая ближе остальных к стартовой вершине.
- ✶ Индукционная аргументация доказывает, что алгоритм Дейкстры правильно решает задачу о кратчайшем пути с единственным истоком, когда входной граф имеет только неотрицательные длины ребер.
- ✶ Алгоритм Дейкстры не обязательно правильно решает задачу о кратчайшем пути с единственным истоком, когда некоторые ребра входного графа имеют отрицательные длины.
- ✶ Простая реализация алгоритма Дейкстры выполняется за время $O(mn)$, где m и n обозначают число ребер и вершин входного графа соответственно.

Задачи на закрепление материала

Задача 9.1. Рассмотрим ориентированный граф G с несовпадающими и неотрицательными длинами ребер. Пусть s равно стартовой вершине, и t равно финальной вершине, и будем считать, что G имеет по крайней мере один путь s-t. Какие из следующих утверждений являются истинными? (Выберите все подходящие варианты.)

а) Кратчайший (то есть с минимальной длиной) путь s-t может иметь вплоть до $n - 1$ ребер, где n — число вершин.

б) Существует кратчайший путь *s*-*t* без повторяющихся вершин (то есть без петель).

в) Кратчайший путь *s*-*t* должен включать ребро минимальной длины в графе *G*.

г) Кратчайший путь *s*-*t* должен исключать ребро максимальной длины в графе *G*.

Задача 9.2. Рассмотрим ориентированный граф *G* со стартовой вершиной *s*, финальной вершиной *t* и неотрицательными длинами ребер. При каких условиях гарантируется уникальность кратчайшего пути *s*-*t*?

а) Когда все длины ребер являются несовпадающими целыми положительными числами.

б) Когда все длины ребер являются несовпадающими степенями двойки.

в) Если длины всех ребер являются несовпадающими положительными числами и граф *G* не содержит ориентированных циклов.

г) Ни один из представленных вариантов не является правильным.

Задача 9.3. Рассмотрим ориентированный граф *G* с неотрицательными длинами ребер и двумя несовпадающими вершинами, *s* и *t*. Пусть *P* обозначает кратчайший путь из *s* в *t*. Если мы добавим 10 к длине каждого ребра в графе, то: (выберите все, что применимо.)

а) *P* определенно остается кратчайшим путем *s*-*t*.

б) *P* определенно не остается кратчайшим путем *s*-*t*.

в) *P* может или не может оставаться кратчайшим путем *s*-*t* (в зависимости от графа).

г) Если *P* имеет только одно ребро, то *P* определенно остается кратчайшим путем *s*-*t*.

Задача 9.4. Рассмотрим ориентированный граф *G* и стартовую вершину *s* со следующими свойствами: ни одно ребро не входит в стартовую вершину *s*; ребра, выходящие из *s*, имеют произвольную (возможно, отрицательную) длину; все остальные реберные длины являются неотрицательными. Правильно ли алгоритм Дейкстры решает задачу о кратчайшем пути с единственным истоком в этом случае? (Выберите все подходящие варианты.)

а) Да, для всех таких входов.

б) Никогда, ибо таких входов нет.

в) И то и другое (в зависимости от конкретного выбора G, s и длин ребер).

г) Только если мы добавим допущение, что G не содержит ориентированных циклов с отрицательной суммарной длиной.

Задача 9.5. Рассмотрим ориентированный граф G и стартовую вершину s. Предположим, что G имеет отрицательные длины ребер, но не имеет отрицательных циклов, то есть G не имеет ориентированного цикла, в котором сумма его длин ребер является отрицательной. Предположим, что вы выполняете алгоритм Дейкстры на этих входных данных. Какие из следующих утверждений являются истинными? (Выберите все подходящие варианты.)

а) Алгоритм Дейкстры может зациклиться.

б) Невозможно выполнить алгоритм Дейкстры на графе с отрицательными длинами ребер.

в) Алгоритм Дейкстры всегда останавливается, но в некоторых случаях расстояния кратчайшего пути, которые он вычисляет, не будут правильными.

г) Алгоритм Дейкстры всегда останавливается, и в некоторых случаях расстояния кратчайшего пути, которые он вычисляет, будут правильными.

Задача 9.6. В продолжение предыдущей задачи теперь предположим, что входной граф G содержит отрицательный цикл, а также путь из стартовой вершины s в этот цикл. Предположим, вы выполняете алгоритм Дейкстры на этих входных данных. Какие из следующих утверждений являются истинными? (Выберите все подходящие варианты.)

а) Алгоритм Дейкстры может зациклиться.

б) Невозможно выполнить алгоритм Дейкстры на графе с отрицательным циклом.

в) Алгоритм Дейкстры всегда останавливается, но в некоторых случаях расстояния кратчайшего пути, которые он вычисляет, не будут правильными.

г) Алгоритм Дейкстры всегда останавливается, и в некоторых случаях расстояния кратчайшего пути, которые он вычисляет, будут правильными.

Задача повышенной сложности

Задача 9.7. Рассмотрим ориентированный граф $G = (V, E)$ с неотрицательными длинами ребер и стартовой вершиной s. Определим узкое место пути как максимальную длину одного из его ребер (в отличие от суммы длин его ребер). Покажите, как изменить алгоритм Дейкстры так, чтобы вычислить для каждой вершины $v \in V$ наименьшее узкое место любого пути s-v. Ваш алгоритм должен работать за время $O(mn)$, где m и n обозначают число ребер и вершин соответственно.

Задача по программированию

Задача 9.8. Реализуйте на любимом языке программирования алгоритм `Dijkstra` из раздела 9.2 и используйте его для решения задачи о кратчайшем пути с единственным истоком в разных ориентированных графах. С учетом простой реализации в этой главе каков размер самой большой задачи, которую вы можете решить за пять минут или меньшее время? (Обратитесь к сайту www.algorithmsilluminated.org для получения тестовых случаев и совокупностей данных для задач.)

10 Куча

Остальные три главы этой книги посвящены трем наиболее важным и повсеместным структурам данных — кучам, деревьям поиска и хеш-таблицам. Цель состоит в том, чтобы изучить операции, которые поддерживаются этими структурами данных (вместе с их временами выполнения), развить с помощью примеров применений свою интуицию о том, какие структуры данных полезны для каких видов задач, и, возможно, узнать немного о том, как они реализуются под капотом[1]. Мы начинаем с *кучи* — структуры данных, которая способствует быстрым вычислениям минимума либо максимума.

10.1. Структуры данных: краткий обзор

10.1.1. Выбор правильной структуры данных

Структуры данных используются почти в каждой основной части программного обеспечения, поэтому знание того, когда и как их применять, является важным навыком для серьезного программиста. Смысл структуры данных заключается в организации данных таким образом, чтобы вы могли получать к ним доступ быстро и с пользой. Вы уже видели несколько примеров. *Очередь*, используемая в нашей линейно-временной реализации поиска в ширину (раздел 8.2), последовательно организует данные так, что удаление объекта в ее начало или добавление объекта в ее конце занимает постоянное время. *Стек* в качестве структуры данных, которая имела решающее значение в нашей итеративной реализации поиска в глубину (раздел 8.4), позволяет удалять объект или добавлять объект в его начало за постоянное время.

Существует еще много других структур данных — в этой книжной серии мы увидим кучи, бинарные (или двоичные) деревья поиска, хеш-таблицы, фильтры Блума и (в *третьей части*) систему непересекающихся множеств (union-find). Почему такой непонятный перечень? Потому что *разные структуры данных поддерживают разные множества операций, что делает их хорошо пригодными для разных типов задач программирования*. Например, поиск в ширину и поиск в глубину имеют разные потребности, что приводит

[1] Некоторые программисты резервируют словосочетание «*структура данных*» для конкретной реализации и ссылаются на список поддерживаемых операций как на *абстрактный тип данных*.

к двум разным структурам данных. Наша быстрая реализация алгоритма кратчайшего пути Дейкстры (в разделе 10.4) имеет свои собственные отличающиеся потребности, требующие более сложной структуры данных, именуемой кучей.

Каковы плюсы и минусы разных структур данных и как выбрать, какую из них использовать в программе? Как правило, чем больше операций поддерживает структура данных, тем медленнее операции и тем больше объем занимаемого пространства. Следующая цитата, часто приписываемая Альберту Эйнштейну, будет здесь уместной:

> «Все следует упрощать до тех пор, пока это возможно, но не более того».

При реализации программы важно тщательно продумывать, какие именно операции вы будете использовать снова и снова. Например, интересует ли вас только отслеживание того, какие объекты хранятся в структуре данных, или вы также хотите, чтобы они были упорядочены определенным образом? После того как вы поймете потребности вашей программы, вы сможете придерживаться принципа экономности и выбрать структуру данных, которая поддерживает все нужные операции, кроме лишних.

ПРИНЦИП ЭКОНОМИИ

Выберите простейшую структуру данных, которая поддерживает все операции, требуемые приложением.

10.1.2. Переход на следующий уровень

Каков ваш текущий и желаемый уровень осведомленности о структурах данных?

Уровень 0 «Что такое структура данных?»

> Уровень 0 представляет собой полное дилетантство — тот, кто никогда не слышал о структуре данных и не знает, что умная организация ваших данных может значительно улучшить время работы программы.

Уровень 1 «Слышал много хорошего о хеш-таблицах»

Уровень 1 представляет собой поверхностную осведомленность, как если бы, оказавшись в кругу специалистов на вечеринке, вы поддержали бы разговор на уровне основных структур данных[1]. Вы слышали о нескольких основных структурах, таких как деревья поиска и хеш-таблицы, и, возможно, знаете о некоторых из их поддерживаемых операций, но чувствовали бы себя неуверенно, пытаясь использовать их в программе или собеседовании.

Уровень 2 «Для этой задачи требуется куча»

С уровнем 2 у нас уже что-то получается. Это тот, кто обладает солидной грамотностью в отношении основных структур данных, комфортно использует их в качестве клиента в своих собственных программах и имеет хорошее представление о том, какие структуры данных для каких типов задач программирования подходят.

Уровень 3 «Использую только те структуры данных, что написал сам»

Уровень 3, самый продвинутый уровень, относится к высококлассным программистам и специалистам в области computer science, которые не довольствуются просто использованием существующих реализаций структуры данных в качестве клиента. На этом уровне у вас есть детальное понимание сути основных структур данных и того, как они реализуются.

Самое большое решающее расширение возможностей проистекает из достижения уровня 2. Большинство программистов в какой-то момент должны стать образованными клиентами основных структур данных, таких как кучи, деревья поиска и хеш-таблицы. Основная цель глав 10–12 — поднять ваши знания до этой планки с использованием этих структур данных, уделяя особое внимание операциям, которые они поддерживают, и их каноническим применениям. Все эти структуры данных легко доступны в стандартных библиотеках большинства современных языков программирования, которые только и ждут того, чтобы их ловко развертывали в своих собственных программах.

У продвинутых программистов иногда возникает потребность в реализации упрощенной версии одной из этих структур данных с нуля. Каждая из

[1] Как всегда, речь о достаточно занудных фуршетных вечеринках!

последующих глав включает по крайней мере один расширенный раздел, посвященный типичным реализациям этих структур данных. Эти разделы предназначены для тех из вас, кто хочет достичь максимальной вершины.

10.2. Поддерживаемые операции

Куча — это структура данных, которая отслеживает эволюционирующее множество объектов с *ключами* и может быстро идентифицировать объект с наименьшим ключом[1]. Например, объекты могут соответствовать записям о сотрудниках с ключами, равными их идентификационным номерам. Они могут быть ребрами графа, в котором ключи соответствуют их длинам. Либо они могут соответствовать запланированным на будущее событиям, причем каждый ключ указывает на время, в которое событие произойдет[2].

10.2.1. Вставка и извлечение минимума

Самые главные вещи, которые следует помнить о любой структуре данных, — это операции, которые она поддерживает, и время, необходимое для каждой из них. Две самые важные операции, поддерживаемые кучами, это операции Вставить и Извлечь минимум[3].

КУЧИ: ОСНОВНЫЕ ОПЕРАЦИИ

Вставить: с учетом H и нового объекта x добавить x в H.

Извлечь минимум: с учетом кучи H удалить и вернуть из H объект с наименьшим ключом (либо указатель на него).

---

[1] Не следует путать с кучей в качестве динамической *памяти*, то есть частью памяти программы, зарезервированной для динамического выделения.

[2] Ключи часто выражены числами, но могут принадлежать любому полностью упорядоченному множеству — важно то, что для каждой пары неравных ключей один меньше другого.

[3] Структуры данных, поддерживающие эти операции, также называются приоритизированными очередями, или *очередями с приоритетом*.

Например, если для добавления объектов с ключами 12, 7, 29 и 15 в пустую кучу четыре раза вызвать метод Вставить, то операция Извлечь минимум вернет объект с ключом 7. Ключи не обязательно должны быть несовпадающими; если в куче имеется несколько объектов с наименьшим ключом, операция Извлечь минимум возвращает произвольный такой объект.

Было бы легко поддерживать только операцию Вставить, повторно присоединяя новые объекты в конец массива или связного списка (за постоянное время). Загвоздка заключается в том, что операция Извлечь минимум требует линейно-временного исчерпывающего поиска по всем объектам. Ясно также, каким образом поддерживать только операцию Извлечь минимум — отсортировать исходное множество n объектов по ключу заранее раз и навсегда (с использованием времени $O(n \log n)$ на предобработку), а затем последовательно вызывать операцию Извлечь минимум для выхватывания объектов по одному с начала отсортированного списка (каждая за постоянное время). Загвоздка здесь в том, что любая простая реализация операции Вставить требует линейного времени (как вы сами можете убедиться). Хитрость кроется в разработке структуры данных, которая позволяет *обеим* операциям выполняться очень быстро. Именно в этом заключается смысл существования кучи.

Стандартные реализации куч, как та, которая схематично описана в разделе 10.5, гарантируют следующее.

Теорема 10.1. (Время выполнения основных операций кучи.) *В куче с n объектами операции* Вставить *и* Извлечь минимум *выполняются за время* $O(\log n)$.

В качестве бонуса в типичных реализациях константа, скрытая за формой записи O-большое, очень мала, и почти нет лишних пространственных накладных расходов.

Существует также вариант кучи, который поддерживает операции Вставить и Извлечь минимум за время $O(\log n)$, где n — это число объектов. Один из способов реализации этого варианта состоит в смене направления всех неравенств в реализации из раздела 10.5. Второй способ — использовать стандартную кучу, но выполнять операцию HE с ключами объектов перед их вставкой (в результате эффективно преобразовывая операцию Извлечь минимум в операцию Извлечь максимум). Ни один из вариантов кучи не под-

держивает операции Извлечь минимум и Извлечь максимум одновременно за $O(\log n)$ — вы должны выбрать, какую из них вы хотите[1].

10.2.2. Дополнительные операции

Кучи могут также поддержать ряд менее существенных операций.

КУЧИ: ДОПОЛНИТЕЛЬНЫЕ ОПЕРАЦИИ

Найти минимум: с учетом кучи H вернуть объект с наименьшим ключом (либо указателем на него).

Объединить в кучу: с учетом объектов $x_1, ..., x_n$ создать кучу, содержащую эти объекты.

Удалить: с учетом кучи H и указателя на объект x в H удалить x из H.

Вы можете сымитировать операцию Найти минимум, вызвав операцию Извлечь минимум, а затем применив Вставить к результату (за время $O(\log n)$ по теореме 10.1), но типичная реализация кучи может избегать этого обходного решения и поддерживать операцию Найти минимум непосредственно за время $O(1)$. Вы могли бы реализовать операцию Объединить в кучу, вставляя n объекты по одному в пустую кучу (за суммарное время $O(n \log n)$ по теореме 10.1), однако существует блестящий способ добавлять объекты в пустую кучу одним пакетом за суммарное время $O(n)$. Наконец, кучи могут также поддерживать удаление произвольных объектов, а не только объектов с наименьшим ключом за время $O(\log n)$ (см. также программный проект 10.8).

Теорема 10.2. (Время выполнения дополнительных кучевых операций.) *В куче с n объектами операции* Найти минимум, Объединить в кучу *и* Удалить *выполняются за время $O(1)$, $O(n)$ и $O(\log n)$ соответственно.*

[1] Если вы хотите иметь обе, то можете использовать одну кучу каждого типа (см. также раздел 10.3.3) либо обновите свое представление данных до сбалансированного бинарного дерева поиска (см. главу 11).

Обобщим и приведем итоговый перечень показателей для куч.

Таблица 10.1. Кучи: поддерживаемые операции и время их выполнения, где *n* обозначает текущее число объектов, хранящихся в куче

Операция	Время выполнения
Вставить	$O(\log n)$
Извлечь минимум	$O(\log n)$
Найти минимум	$O(1)$
Объединить в кучу	$O(n)$
Удалить	$O(\log n)$

КОГДА ИСПОЛЬЗОВАТЬ КУЧУ

Если вашему приложению требуются быстрые вычисления минимума (либо максимума) для динамически изменяющегося множества объектов, куча обычно является предпочтительной структурой данных.

10.3. Применения

Следующий пункт на повестке — пройтись по нескольким примерам применения и почувствовать все преимущества кучи. Общей темой этих применений является замена вычислений минимума, наивно реализованных с использованием (линейно-временного) исчерпывающего поиска, последовательностью (логарифмически-временных) операций Извлечь минимум из кучи. Всякий раз, когда вы видите алгоритм или программу с большим количеством вычислений минимума или максимума методом полного перебора, у вас в голове должно щелкнуть: *для них требуется куча!*

10.3.1. Применение: сортировка

В нашем первом применении давайте вернемся к праматери всех вычислительных задач — *сортировке*.

ЗАДАЧА: СОРТИРОВКА

Вход: массив из n чисел в произвольном порядке.

Выход: массив тех же чисел, отсортированных от наименьшего к наибольшему.

Например, при заданном входном массиве

5	4	1	8	7	2	6	3

желаемый выходной массив будет

1	2	3	4	5	6	7	8

Пожалуй, самым простым алгоритмом сортировки является `SelectionSort`, или сортировка методом выбора. Этот алгоритм выполняет линейное сканирование входного массива с целью идентификации минимального элемента, обменивает его с первым элементом массива, выполняет второе сканирование оставшихся $n - 1$ элементов с целью идентификации и перестановкой на второе место второго наименьшего элемента и так далее. Каждая проверка занимает время, пропорциональное числу оставшихся элементов, поэтому совокупное время работы равно $\Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$[1]. Поскольку каждая итерация алгоритма `SelectionSort` вычисляет минимальный элемент с помощью исчерпывающего поиска, данный алгоритм поистине взывает к куче! Идея проста: вставить все элементы входного массива в кучу и заполнять выходной массив слева направо последовательно извлеченными минимальными элементами. Первое извлечение производит самый малый элемент; второе — самый малый оставшийся элемент (второй наименьший в совокупности); и так далее.

[1] Сумма $\sum_{i=1}^{n} i$ равна не более n^2 (она содержит n слагаемых, которых не более чем n) и не менее $n^2/4$ (она имеет $n/2$ слагаемых, которых в общей сложности не менее $n/2$).

HEAPSORT (КУЧЕВАЯ СОРТИРОВКА)

Вход: массив A из n несовпадающих целых чисел.

Выход: массив B с теми же целыми числами, отсортированными от наименьшего к наибольшему.

H := пустая куча

for i = 1 to n **do**

  Вставить $A[i]$ в H

for i = 1 to n **do**

  $B[i]$:= Извлечь минимум из H

ТЕСТОВОЕ ЗАДАНИЕ 10.1

Каково время работы алгоритма `HeapSort` как функции от длины n входного массива?

а) $O(n)$

б) $O(n \log n)$

в) $O(n^2)$

г) $O(n^2 \log n)$

(Решение и пояснение см. ниже.)

Правильный ответ: (б). Работа, выполняемая алгоритмом `HeapSort`, сводится к $2n$ операциям на куче, содержащей не более n объектов[1]. Поскольку теорема 10.1 гарантирует, что каждая кучевая операция требует время $O(\log n)$, совокупное время выполнения равно $O(n \log n)$.

[1] Еще более оптимальная реализация заменит первый цикл одной операцией Объединить в кучу, которая выполняется за время $O(n)$. Однако второй цикл по-прежнему требует времени $O(n \log n)$.

Теорема 10.3. (Время работы алгоритма `HeapSort`.) *Для каждого входного массива длины $n \geq 1$ время работы алгоритма* `HeapSort` *равно* $O(n \log n)$.

Давайте сделаем шаг назад и оценим, что сейчас произошло. Мы начали с наименее изобретательного алгоритма сортировки из возможных, квадратично-временного алгоритма сортировки выбором `SelectionSort`. Мы распознали шаблон повторяющихся вычислений минимума, подставили кучевую структуру данных и — ба-бах! — выскочил алгоритм сортировки за время $O(n \log n)$[1]. Это отличное время работы для алгоритма сортировки — оно даже является оптимальным, вплоть до постоянных множителей, среди алгоритмов сортировки на основе сравнения[2]. Одним из побочных продуктов этого наблюдения является доказательство того, что нет никакого основанного на сравнении способа реализовать операции Вставить и Извлечь минимум за время более оптимальное, чем логарифмическое: такое решение дало бы управляемый сравнениями алгоритм сортировки лучше, чем $O(n \log n)$, и мы знаем, что это невозможно.

10.3.2. Применение: событийный менеджер

Наше второе применение, хотя и немного очевидное, является и каноническим, и практическим. Представьте, что вам поручили написать программу,

---

[1] Для наглядности мы описали алгоритм `HeapSort` с использованием отдельных входных и выходных массивов, но его можно реализовать прямо на месте, практически без дополнительной памяти. Такая реализация прямо на месте является сверхпрактичным алгоритмом и в большинстве приложений почти так же быстра, как и быстрая сортировка `QuickSort`.

[2] Напомним из раздела 5.6 книги «Совершенный алгоритм. Основы», что управляемый сравнениями алгоритм сортировки обращается к входному массиву только посредством сравнений между парами элементов и никогда напрямую не обращается к значению элемента. «Универсальные» алгоритмы сортировки, которые не делают допущений о сортируемых элементах, обязательно основаны на сравнении. Примеры включают сортировку выбором `SelectionSort`, сортировку вставкой `InsertionSort`, кучевую сортировку `Heapsort` и быструю сортировку `QuickSort`. Не-примеры включают блочную сортировку `BucketSort`, сортировку подсчетом `CountingSort` и поразрядную сортировку `RadixSort`. Теорема 5.5 из *первой части* показывает, что ни один управляемый сравнениями алгоритм сортировки не имеет асимптотического времени работы для худшего случая лучше, чем $\Theta(n \log n)$.

которая выполняет симуляцию физического мира. Возможно, вы участвуете в разработке баскетбольной видеоигры. В качестве симуляции вы должны следить за различными событиями и тем, когда они должны произойти: когда игрок попадает мячом под определенным углом и с определенной скоростью, когда мяч в результате попадает в заднюю часть обода, когда два игрока одновременно соперничают за отскок, когда один из этих игроков фолит на другом, совершая толчок сзади, и так далее.

Симуляция должна постоянно определять, что произойдет дальше. Это сводится к повторным вычислениям минимума на множестве запланированных событий, поэтому в вашей голове должно щелкнуть: данная задача требует кучи! Если события хранятся в куче с ключами, равными их запланированному времени, операция Извлечь минимум предоставит вам следующее событие на блюдечке с голубой каемочкой за логарифмическое время. Новые события могут вставляться в кучу по мере их возникновения (опять же, за логарифмическое время).

10.3.3. Применение: поддержка медианы

В качестве менее очевидного применения куч рассмотрим задачу *о поддержке медианы*. Вам предлагается последовательность чисел, одно за другим; для простоты предположим, что они не совпадают. Всякий раз, когда вы получаете новое число, ваша обязанность — ответить медианным элементом всех чисел, которые вы встречали до сих пор[1]. Таким образом, увидев первые 11 чисел, вы должны ответить шестым наименьшим из тех, которые вы встречали; после 12 — шестым или седьмым наименьшим; после 13 — седьмым наименьшим, и так далее.

Один из подходов к этой задаче, который должен казаться излишним, заключается в пересчете медианы с нуля на каждой итерации. Мы видели в главе 6, как вычислять медиану массива длиной n за время $O(n)$, поэтому данное решение требует $O(i)$ времени в каждом раунде i. В качестве альтернативы мы могли бы содержать все встречавшиеся ранее элементы в отсортированном

[1] Напомним, что *медиана* множества чисел является его срединным элементом. В массиве с нечетной длиной $2k - 1$ медиана является k-порядковой статистикой (то есть k-м наименьшим элементом). В массиве с четной длиной $2k$ k-порядковая и $(k + 1)$-порядковая статистики считаются медианными элементами.

массиве, благодаря чему легко вычислить медианный элемент за постоянное время. Недостатком этого является то, что обновление отсортированного массива при поступлении нового числа может потребовать линейного времени. Можем ли мы добиться лучшего?

Используя кучи, мы можем решить задачу о поддержке медианы всего за *логарифмическое* время за раунд. Я предлагаю в этом месте отложить книгу и потратить несколько минут на размышления о том, как это можно сделать.

Ключевая идея в том, чтобы поддерживать две кучи H_1 и H_2, при этом удовлетворяя два инварианта[1]. Первый инвариант заключается в том, что H_1 и H_2 являются *сбалансированными*, то есть каждая из них содержит одинаковое число элементов (после четного раунда) либо что одна содержит ровно на один элемент больше, чем другая (после нечетного раунда). Второй инвариант состоит в том, что H_1 и H_2 являются *упорядоченными*, то есть каждый элемент в H_1 меньше, чем каждый элемент в H_2. Например, если до этого цифры были 1, 2, 3, 4, 5, тогда куча H_1 хранит 1 и 2, и куча H_2 хранит 4 и 5; медианный элемент 3 может входить в любую из них как максимальный элемент кучи H_1 или минимальный элемент кучи H_2. Если нам встречались числа 1, 2, 3, 4, 5, 6, то первые три цифры находятся в H_1, а вторые три — в H_2; и максимальный элемент H_1, и минимальный элемент H_2 являются медианными элементами. Однако замысловатость вот в чем: H_2 будет стандартной кучей с поддержкой операций Вставить и Извлечь минимум, тогда как H_1 будет максимальным вариантом, описанным в разделе 10.2.1, с поддержкой операций Вставить и Извлечь максимум. Благодаря этому мы можем извлечь медианный элемент с помощью одной кучевой операции, будь то в H_1 или в H_2.

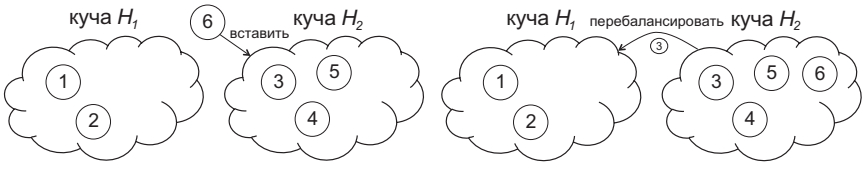
Мы все еще должны объяснить, как обновлять кучи H_1 и H_2 всякий раз, когда прибывает новый элемент, так, чтобы они оставались сбалансированными и упорядоченными. Для того чтобы выяснить, куда вставить новый элемент x, не нарушая упорядоченность куч, достаточно вычислить максимальный элемент y в H_1 и минимальный элемент z в H_2[2]. Если x меньше y, то он должен уйти в H_1;

---

[1] *Инвариант алгоритма* — это свойство, которое всегда истинно в предписанных точках его исполнения (как и в конце каждой итерации цикла).

[2] Это может быть сделано за логарифмическое время путем извлечения и повторной вставки этих двух элементов. Более оптимальным решением является использование операций Найти минимум и Найти максимум, которые выполняются за постоянное время (см. раздел 10.2.2).

если он больше z, то должен уйти в H_2; если он находится между ними, то может уйти в любую из них. Остаются ли кучи H_1 и H_2 сбалансированными даже после вставки x? Да, за исключением одного случая: в четном раунде $2k$, если x вставляется в бóльшую кучу (с k элементами), то эта куча будет содержать $k+1$ элементов, в то время как другая содержит только $k-1$ элементов (рис. 10.1, *а*). Но этот дисбаланс легко исправить: извлечь максимальный либо минимальный элемент из H_1 либо H_2 соответственно (в зависимости от того, какая из них содержит больше элементов) и вставить этот элемент в другую кучу (рис. 10.1, *б*). Две кучи остаются упорядоченными (как вы сами можете убедиться), а теперь и сбалансированными. Каждый раунд в данном решении используется постоянное число кучевых операций за время работы $O(\log i)$ в раунде i.



а) Вставка может вызвать дисбаланс *б*) Перебалансировка

Рис. 10.1. При вставке нового элемента куча H_2 содержит на два элемента больше, чем куча H_1, наименьший элемент в H_2 извлекается и повторно вставляется в H_1 для восстановления баланса

10.4. Ускорение алгоритма Дейкстры

Нашим финальным и самым сложным применением куч будет почти линейная реализация алгоритма Дейкстры для задачи кратчайшего пути с единственным истоком (глава 9). Это приложение наглядно иллюстрирует прекрасное взаимодействие между проектированием алгоритмов и проектированием структур данных.

10.4.1. Почему именно кучи?

В предложении 9.2 мы увидели, что простая реализация алгоритма Дейкстры требует времени $O(mn)$, где m — число ребер и n — число вершин. Эта реализация достаточно быстра для обработки графов среднего размера (с тысячами вершин и ребер), но не больших графов (с миллионами вершин и ребер).

Можем ли мы добиться лучшего? Кучи позволяют построить ослепительно быструю, почти линейно-временную реализацию алгоритма Дейкстры.

Теорема 10.4. (Время выполнения алгоритма `Dijkstra` (на основе кучи).) *Для каждого ориентированного графа $G = (V, E)$, каждой стартовой вершины s и каждого варианта неотрицательных длин ребер кучевая реализация алгоритма* `Dijkstra` *выполняется за время $O((m + n) \log n)$, где $m = |E|$ и $n = |V|$.*

Время $O((m + n) \log n)$, хотя и не такое быстрое, как наши линейно-временные алгоритмы графового поиска, по-прежнему является фантастическим временем работы, сравнимым с нашими лучшими алгоритмами сортировки, и является достаточно хорошим, для того чтобы квалифицироваться как бесплатный примитив.

Вспомним, как работает алгоритм Дейкстры (раздел 9.2). Данный алгоритм поддерживает подмножество $X \subseteq V$ вершин, к которым он уже вычислил расстояния кратчайшего пути. На каждой итерации он определяет ребро, пересекающее границу (причем хвост находится в X, а голова находится в $V - X$) с минимальной дейкстровой балльной оценкой, где дейкстрова оценка такого ребра (v, w) — это (уже вычисленное) расстояние кратчайшего пути $len(v)$ из стартовой вершины в v плюс длина l_{vw} ребра. Другими словами, каждая итерация главного цикла делает вычисление минимума на дейкстровых оценках ребер, которые пересекают границу. Для выполнения этих вычислений минимума в простой реализации используется исчерпывающий поиск. Поскольку ускорение вычислений минимума с линейного времени до логарифмического является смыслом существования куч, в этот момент в вашей голове должно щелкнуть: в алгоритме Дейкстры требуется куча!

10.4.2. План

Что мы должны хранить в куче и какими должны быть их ключи? Ваша первая мысль может заключаться в том, чтобы хранить ребра входного графа в куче с целью замены вычислений минимума (на ребрах) в простой реализации на вызовы операции Извлечь минимум. Эту идею можно заставить работать, но более блестящая и быстрая реализация будет хранить *вершины* в куче. Эта идея может вас удивить, так как дейкстровы оценки определены для ребер,

а не для вершин. С другой стороны, нас заботили дейкстровы реберные оценки только потому, что они направляли нас к вершине, которая стояла следующей в ряду на обработку. Можем ли мы применить кучу, чтобы сразу перейти к делу и непосредственно вычислить эту вершину?

Конкретный план состоит в том, чтобы хранить еще необработанные вершины ($V - X$ в псевдокоде алгоритма `Dijkstra`) в куче, при этом поддерживая нижеизложенный инвариант.

ИНВАРИАНТ

Ключом вершины $w \in V - X$ является минимальная дейкстрова оценка ребра с хвостом $v \in X$ и головой w, либо $+\infty$, если такое ребро не существует.

То есть мы хотим, чтобы уравнение

$$key(w) = \min_{(v,w) \in E: v \in X} \underbrace{len(v) + l_{vw}}_{\text{дейкстрова балльная оценка}} \tag{10.1}$$

всегда соблюдалось для каждого $w \in V - X$, где $len(v)$ обозначает расстояние кратчайшего пути v, вычисленное на более ранней итерации алгоритма (рис. 10.2).

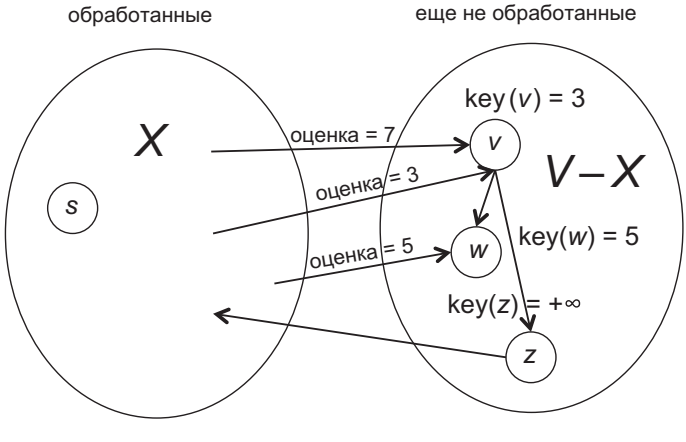


Рис. 10.2. Ключ вершины $w \in V - X$ определяется как минимальная дейкстрова оценка ребра с головой w и хвостом в X

Что же тут происходит? Представьте, что мы используем турнир с выбыванием в два раунда с целью определить ребро (v, w), где $v \in X$ и $w \notin X$, с минимальной дейкстровой оценкой. Первый раунд состоит из локального турнира для каждой вершины $w \in V - X$, где участниками являются ребра (v, w), где $v \in X$, и головой w, и победителем первого раунда является участник с наименьшей дейкстровой оценкой (если таковая имеется). Победители первого раунда (не более одного на вершину $w \in V - X$) переходят во второй раунд, и чемпионом финального раунда является победитель первого раунда с наименьшей дейкстровой оценкой. Этот чемпион является тем же самым ребром, которое было бы идентифицировано исчерпывающим поиском.

Значением ключа (10.1) вершины $w \in V - X$ является именно выигрышная дейкстрова оценка в локальном турнире в w, поэтому наш инвариант эффективно реализует все соревнования первого раунда. Затем извлечение вершины с минимальным ключом реализует второй раунд турнира и возвращает на блюдце с голубой каемочкой следующую вершину для обработки, а именно голову пересекающего ребра с наименьшей дейкстровой оценкой. Дело в том, что до тех пор, пока мы поддерживаем наш инвариант, мы можем реализовать каждую итерацию алгоритма Дейкстры с помощью одной кучевой операции.

Псевдокод выглядит следующим образом[1]:

DIJKSTRA (НА ОСНОВЕ КУЧИ, ЧАСТЬ 1)

Вход: ориентированный граф $G = (V, E)$, представленный в виде списков смежности, вершина $s \in V$, длина $l_e \geq 0$ для каждого $e \in E$.

Постусловие: для каждой вершины v значение $len(v)$ равно истинному кратчайшему расстоянию $dist(s, v)$.

```
// Инициализация
```

1) X := пустое множество, H := пустая куча

---

[1] Инициализация множества X обрабатываемых вершин пустым множеством, а не стартовой вершиной приводит к более чистому псевдокоду (ср.: раздел 9.2.1). Первая итерация главного цикла гарантированно извлекает стартовую вершину (понимаете, почему?), которая является первой вершиной, добавляемой в X.

2) $key(s) := 0$

3) **for** каждая $v \neq s$ **do**

4) $key(v) := +\infty$

5) **for** каждая $v \in V$ **do**

6) Вставить v в H `// либо использовать операцию`
`// «Объединить в кучу»`

`// Главный цикл`

7) **while** H является непустой **do**

8) w^* := Извлечь минимум (H)

9) добавить w^* в X

10) $len(w^*) := key(w^*)$

`// обновить кучу для поддержания инварианта`

11) `(будет продолжено)`

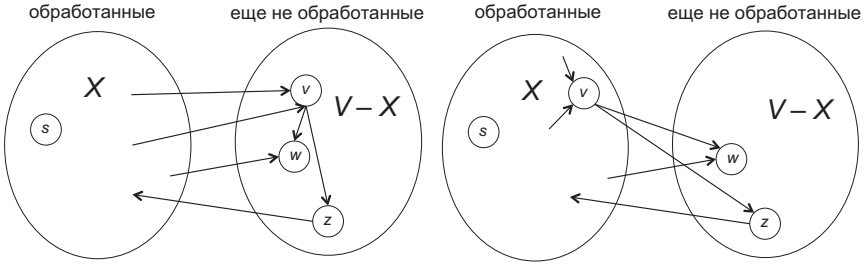
Но сколько работы потребуется для поддержания инварианта?

10.4.3. Поддержание инварианта

Теперь пришло время расплатиться за музыку. Мы наслаждались плодами нашего инварианта, который сводит каждое вычисление минимума, требуемое алгоритмом Дейкстры, до одной кучевой операции. Взамен мы должны объяснить, как поддерживать его без чрезмерной работы.

Каждая итерация алгоритма перемещает одну вершину v из $V - X$ в X, в результате изменяя границу (рис. 10.3). Ребра из вершин в множестве X в вершину v втягиваются в X и больше не пересекают границу. Более проблематично то, что ребра из v в другие вершины множества $V - X$ больше не располагаются полностью в $V - X$ и вместо этого переходят из X в $V - X$. Почему это представляет проблему? Потому что наш инвариант (10.1) настаивает на том, что для каждой вершины $w \in V - X$ ключ w равен наименьшей дейкстровой оценке пересекающего ребра, заканчивающегося в w. Новые пересекающие ребра означают новых кандидатов на наименьшую дейкстрову оценку, поэтому

правая часть (10.1) может уменьшиться для некоторых вершин w. Например, в первый раз, когда вершина v с ребром $(v, w) \in E$ втягивается в X, это выражение падает с $+\infty$ до конечного числа (а именно, $len\,(v) + l_{vw}$).



а) До *б*) После

Рис. 10.3. Когда новая вершина *v* перемещается из $V - X$ в X, ребра, выходящие из *v*, могут стать пересекающими ребрами

Всякий раз, когда мы извлекаем вершину w^* из кучи, перемещая ее из $V - X$ в X, нам может потребоваться уменьшить ключ некоторых вершин, оставшихся в $V - X$, с тем чтобы отразить новые пересекающие ребра. Поскольку все новые пересекающие ребра исходят из w^*, нам нужно только перебрать список исходящих ребер w^* и проверить вершины $y \in V - X$ с ребром (w^*, y). Для каждой такой вершины y существует два кандидата на победителя первого раунда в локальном турнире вершины y: либо он такой же, как и раньше, либо это новый участник (w^*, y). Таким образом, новое значение ключа вершины y должно быть либо его старым значением, либо дейкстровой оценкой $len\,(w^*) + l_{w^*y}$ нового пересекающего ребра, в зависимости от того, какое из них меньше.

Как уменьшить ключ объекта в куче? Один простой способ — удалить его, используя операцию Удалить, описанную в разделе 10.2.2, обновить его ключ и использовать операцию Вставить, чтобы добавить его обратно в кучу[1]. Этим завершается реализация алгоритма `Dijkstra` на основе кучи.

---

[1] Некоторые реализации кучи экспортируют операцию Уменьшить ключ, выполняемую за время $O(\log n)$ для кучи из n объектов. В этом случае требуется только одна кучевая операция.

DIJKSTRA (НА ОСНОВЕ КУЧИ, ЧАСТЬ 2)

```
    // обновить кучу для поддержания инварианта
```

12) **for** каждое ребро (w^*, y) **do**

13) Удалить y из H

14) $key(y) := \min\{key(y), len(w^*) + l_{w^*y}\}$

15) Вставить y в H

10.4.4. Время выполнения

Почти вся работа, выполняемая кучевой реализацией алгоритма `Dijkstra`, состоит из кучевых операций (как вы сами можете убедиться). Каждая из этих операций занимает время $O(\log n)$, где n — это число вершин. (Куча никогда не содержит более $n - 1$ объектов.)

Сколько кучевых операций выполняет данный алгоритм? Существует $n - 1$ операций в каждой из строк 6 и 8 — по одной на вершину, отличную от стартовой вершины s. Как насчет строк 13 и 15?

ТЕСТОВОЕ ЗАДАНИЕ 10.2

Сколько раз алгоритм `Dijkstra` выполняет строки 13 и 15? Выберите наименьшую применимую границу. (Как обычно, n и m обозначают соответственно число вершин и ребер.)

а) $O(n)$

б) $O(m)$

в) $O(n^2)$

г) $O(mn)$

(Решение и пояснение см. ниже.)

Правильный ответ: (б). Строки 13 и 15 могут выглядеть немного пугающе. За одну итерацию главного цикла эти две строки могут быть выполнены $n - 1$ раз — один раз на исходящее ребро w^*. Существует $n - 1$ итераций,

которые, кажется, приводят к квадратичному числу операций с кучей. Эта оценка точна для плотных графов, но в целом мы можем добиться лучшего. Причина? Давайте сделаем ответственными за эти кучевые операции ребра, а не вершины. Каждое ребро (v, w) графа имеет не более одного появления в строке 12 — когда v впервые извлекается из кучи и перемещается из $V - X$ в X[1]. Таким образом, строки 13 и 15 выполняются не более одного раза на ребро, давая в общей сложности $2m$ операций, где m — это количество ребер.

Тестовое задание 10.2 показывает, что кучевая реализация алгоритма `Dijkstra` использует $O(m + n)$ кучевых операций, каждая из которых занимает время $O(\log n)$. Совокупное время работы составляет $O((m + n) \log n)$, как обещано теоремой 10.4. *Ч. т. д.*

*10.5. Детали реализации

Давайте выведем ваше понимание кучи на следующий уровень, описав то, как вы бы ее реализовали с нуля. Мы сосредоточимся на двух основных операциях — Вставить и Извлечь минимум — и на том, как обеспечить выполнение обеих за логарифмическое время.

10.5.1. Кучи в виде деревьев

Существует два способа визуализации объектов в куче: в виде дерева (лучше всего подходит для изображений и экспозиции) или в виде массива (лучше всего подходит для реализации). Давайте начнем с деревьев.

Кучи можно рассматривать как корневое бинарное дерево, где каждый узел имеет 0, 1 или 2 детей, или потомков, — когда каждый уровень максимально заполнен. Если число хранимых объектов на один меньше степени 2, то каждый уровень является полным (рис. 10.4, *а* и *б*). Когда число объектов находится между двумя такими числами, то единственный неполный уровень является последним, который заполняется слева направо (рис. 10.4, *с*)[2].

---

[1] Если w извлекается перед v, то ребро (v, w) никогда не появляется.

[2] По какой-то непонятной причине специалисты в области computer science считают, что деревья растут вниз.

Куча так управляет объектами, ассоциированными с ключами, чтобы соблюдалось следующее свойство кучи.

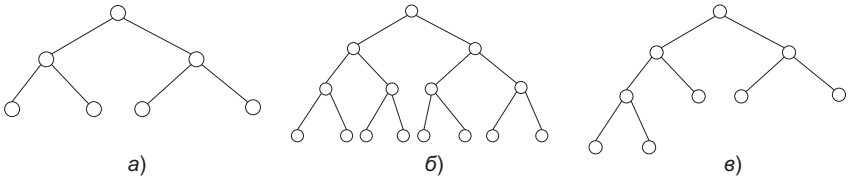
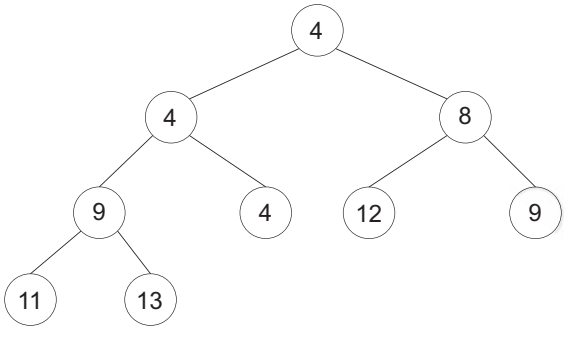


Рис. 10.4. Полные бинарные деревья с 7, 15 и 9 узлами

СВОЙСТВО КУЧИ

Для каждого объекта *x* ключ объекта *x* меньше или равен ключам его потомков.

Повторяющиеся ключи разрешены. Например, ниже приведена допустимая куча, содержащая девять объектов[1]:



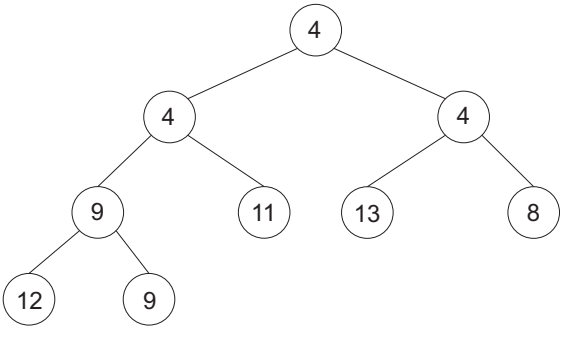
Для каждой пары «родитель — потомок» ключ родителя не больше ключа потомка[2].

---

[1] Когда мы рисуем кучу, то показываем только ключи объектов. Не забывайте, что куча на самом деле хранит объекты (или указатели на объекты). Каждый объект ассоциирован с ключом и, возможно, с большим количеством других данных.

[2] Итеративное применение свойства кучи к потомкам объекта, потомкам его потомков и т. д. показывает, что ключ каждого объекта меньше или равен ключу *всех* его пря-

Существует несколько способов упорядочения объектов так, чтобы свойство кучи соблюдалось. Приведем еще одну кучу с таким же множеством ключей:



У обеих куч в корне имеется ключ «4», который также (ограничен) является самым малым из всех ключей. Это не случайность: поскольку по мере того, как вы будете выполнять обход кучи вверх, ключи будут только уменьшаться, при этом корневой ключ является настолько малым, насколько это может получиться. Это должно звучать обнадеживающе, учитывая, что смысл существования кучи — быстрые вычисления минимума.

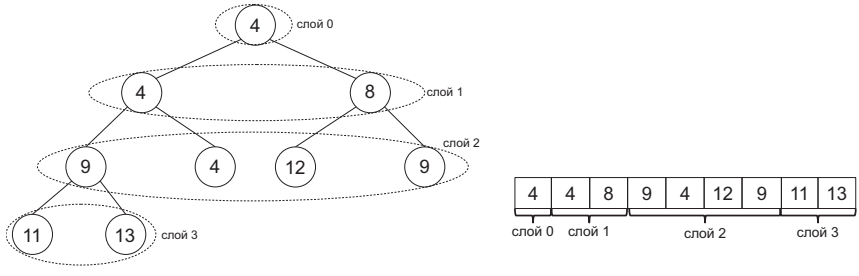
10.5.2. Кучи в виде массива

Мысленно мы визуализируем кучу как дерево, но в реализации мы используем массив с длиной, равной максимальному числу объектов, которые мы намерены хранить. Первый элемент массива соответствует корню дерева, последующие два элемента — следующему уровню дерева (в том же порядке), и так далее (рис. 10.5).

Отношения «родитель — потомок» в дереве хорошо преобразуются в массив (табл. 10.2). Если такие позиции помечены 1, 2, ..., n, где n — это число объектов, то потомки объекта в позиции i соответствуют объектам в позициях $2i$ и $2i + 1$ (если таковые имеются). Например, на рис. 10.5 потомками корня (в позиции 1) являются следующие два объекта (в позициях 2 и 3),

---

мых потомков. В приведенном выше примере показано, что свойство кучи ничего не подразумевает об относительном порядке ключей в разных поддеревьях — так же, как это происходит в реальных семейных древах!



а) Представление в виде дерева

б) Представление в виде массива

Рис. 10.5. Отображение древовидного представления кучи на его представление в виде массива

потомками 8 (в позиции 3) являются объекты в позициях 6 и 7 и так далее. Двигаясь в обратную сторону, для некорневого объекта (в позиции $i \geq 2$) родителем i является объект в позиции $\lfloor i/2 \rfloor$[1]. Так, на рис. 10.5 родителем последнего объекта (в позиции 9) является объект в позиции $\lfloor 9/2 \rfloor = 4$.

Таблица 10.2. Взаимосвязи между позицией $i \in \{1, 2, 3, \ldots, n\}$ объекта в куче и позициями его родителя, левого потомка и правого потомка, где n обозначает число объектов в куче

Позиция родителя	$\lfloor i/2 \rfloor$ (при условии, что $i \geq 2$)
Позиция левого потомка	$2i$ (при условии, что $2i \leq n$)
Позиция правого потомка	$2i + 1$ (при условии, что $2i + 1 \leq n$)

Такие простые формулы для перехода от потомка к его родителю и обратно существуют потому, что мы используем только полные бинарные деревья[2]. Нет необходимости хранить дерево явным образом; следовательно, кучевая структура данных имеет минимальные пространственные накладные расходы[3].

---

[1] Форма записи $\lfloor x \rfloor$ обозначает функцию «floor», которая округляет свой аргумент вниз до ближайшего целого числа.

[2] В качестве бонуса в языках низкого уровня можно умножать или делить на 2 до смешного быстро, используя трюки с битовыми сдвигами.

[3] Напротив, деревья поиска (глава 11) не обязательно должны быть полными; они требуют дополнительного пространства для хранения явных указателей из каждого узла в узлы его потомков.

10.5.3. Реализация операции «Вставить» со временем O (log n)

Мы проиллюстрируем реализацию операций Вставить и Извлечь минимум на примере, а не на псевдокоде[1]. Задача состоит в том, чтобы поддерживать дерево полным и поддерживать свойство кучи после добавления или удаления объекта. Мы будем следовать одинаковой схеме для обеих операций:

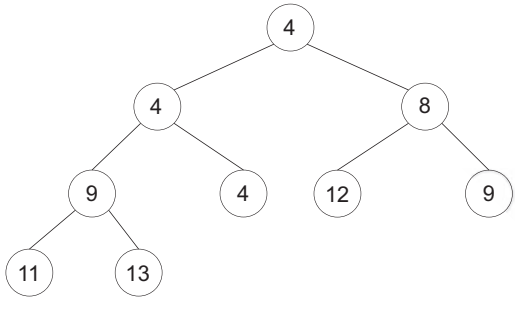
1. Поддерживать дерево полным самым очевидным способом из возможных.
2. Играя в игру «Поймай крота», систематически подавлять любые выскакивающие нарушения свойства кучи.

В частности, вспомните операцию Вставить:

> с учетом кучи H и нового объекта x добавить x в H.

После добавления объекта x в кучу H куча H должна по-прежнему соответствовать полному бинарному дереву (с одним узлом больше, чем раньше), которое удовлетворяет свойству кучи. Операция должна занимать время $O(\log n)$, где n — это число объектов в куче.

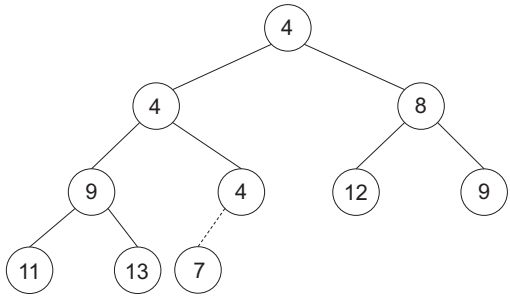
Начнем с нашего примера:



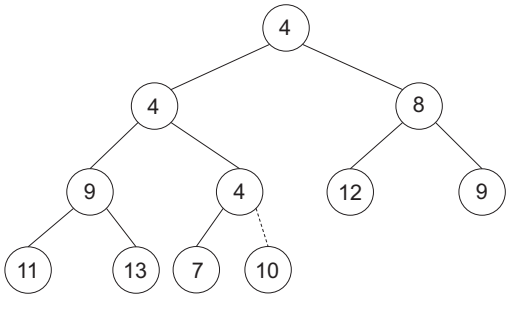
При вставке нового объекта наиболее очевидный способ поддерживать дерево полным состоит в присоединении нового объекта в конец массива либо

[1] Мы продолжим изображать кучи в виде деревьев, но не забывайте, что они хранятся в виде массивов. Когда мы говорим о переходе от узла к потомку или его родителю, мы имеем в виду применение простых индексных формул в табл. 10.2.

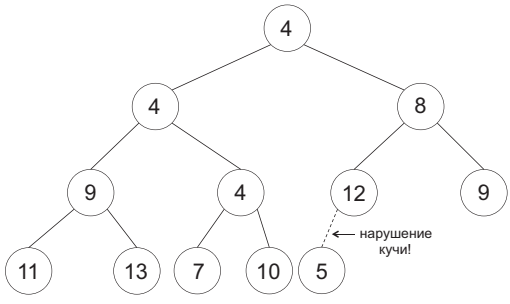
равным образом в последний уровень дерева. (Если последний уровень уже заполнен, то объект станет первым на новом уровне.) При условии, что реализация отслеживает число *n* объектов (что легко сделать), этот шаг занимает постоянное время. Например, если вставить объект с ключом 7 в нашем рабочем примере, то мы получим:



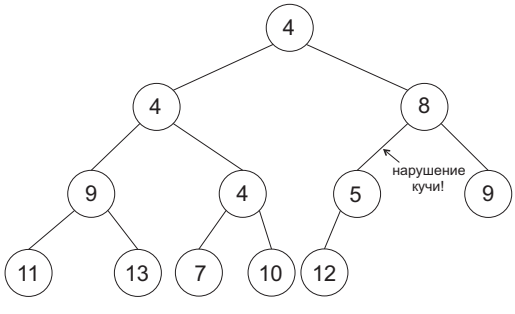
Мы имеем полное бинарное дерево, но соблюдается ли свойство кучи? Существует лишь одно место, где оно может быть нарушено, — новая пара «родитель — потомок» (4 и 7). В этом случае нам повезло, и новая пара не нарушает свойство кучи. Если наша следующая вставка является объектом с ключом 10, то нам снова повезет, и мы сразу получим допустимую кучу:



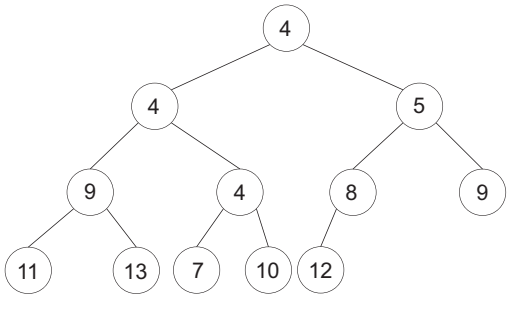
Но предположим, что мы вставляем объект с ключом 5. После присоединения его дальше в конец наше дерево будет выглядеть так:



Теперь у нас появилась проблема: новая пара «родитель — потомок» (12 и 5) нарушает свойство кучи. И что нам с этим делать? Мы можем, по крайней мере, решить эту проблему локально, переставив местами два узла в нарушающей паре:



Это исправляет нарушение пары «родитель — потомок». Однако мы еще не вышли из трудного положения, так как нарушение кучи переместилось вверх, к 8 и 5. Поэтому мы делаем это снова и переставляем местами узлы в нарушающей паре, в результате получая:



Это явным образом исправляет нарушающую пару «родитель — потомок». Мы убедились, что такая перестановка может проталкивать нарушение свойства кучи вверх, но здесь этого не происходит — 4 и 5 уже находятся в правильном порядке. Возможно, вы беспокоитесь, что перестановка также может проталкивать нарушение вниз. Но этого не случилось — 8 и 12 уже находятся в правильном порядке. После восстановления свойства кучи вставка завершена.

В общем случае операция Вставить присоединяет новый объект в конец кучи и многократно переставляет местами узлы нарушающей пары[1]. В любой момент времени существует не более одной нарушающей пары «родитель — потомок» — пары, в которой новый объект является потомком[2]. Каждая перестановка проталкивает нарушающую пару «родитель — потомок» вверх на один уровень в дереве. Этот процесс не может продолжаться вечно — если новый объект попадает в корень, то у него нет родителя, и не может существовать нарушающей пары «родитель — потомок».

ВСТАВИТЬ

1. Прикрепить новый объект в конец кучи и увеличить размер кучи.
2. Многократно переставлять местами новый объект с родителем, до тех пор пока свойство кучи не будет восстановлено.

[1] Такая подпрограмма перестановки местами в англоязычной литературе известна под многими именами, в том числе `Bubble-Up` (всплыть вверх), `Sift-Up` (просеять вверх), `Heapify-Up` (объединить, распространяя вверх) и многими другими.

[2] Между новым объектом и его потомками ни разу не происходит нарушений кучи. Изначально у него нет потомков, и после перестановки его потомки состоят из замененного им узла (который имеет более крупный ключ, так как в противном случае мы бы не выполняли перестановку) и предыдущего потомка того узла (который по свойству кучи может иметь только еще более крупный ключ). Каждая пара «родитель — потомок», не включая новый объект, появлялась в исходной куче и, следовательно, не нарушает свойство кучи. Например, после двух перестановок местами в нашем примере 8 и 12 снова находятся в связи «родитель — потомок», как и в исходной куче.

Поскольку куча является полным бинарным деревом, она имеет $\approx \log_2 n$ уровней, где n — это число объектов в куче. Число перестановок местами не превышает число уровней, и на перестановку требуется только постоянный объем работы. Мы заключаем, что в худшем случае время выполнения операции Вставить равно $O(\log n)$, как того и хотелось.

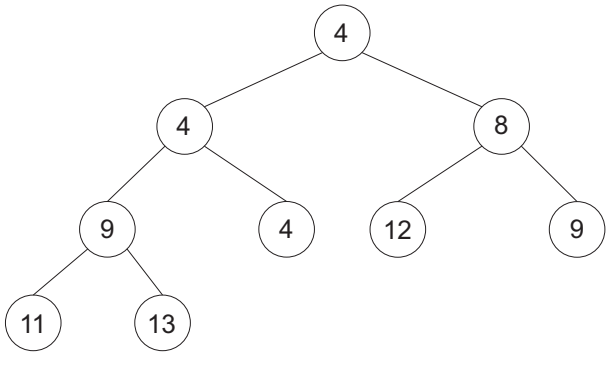
10.5.4. Реализация операции «Извлечь минимум» со временем O (log n)

Вспомните операцию Извлечь минимум:

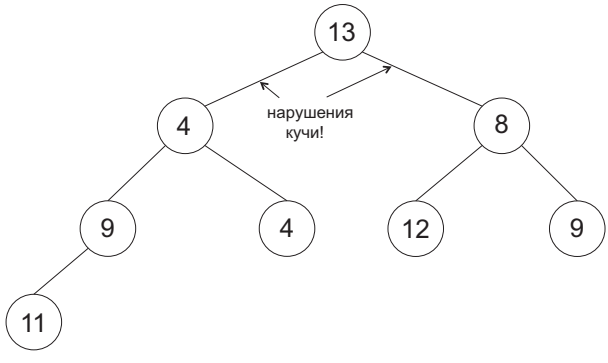
> с учетом кучи H удалить и вернуть из H объект с наименьшим ключом.

Корень кучи гарантированно будет таким объектом. Задача заключается в восстановлении полного бинарного дерева и свойств кучи после удаления корня кучи.

Мы снова поддерживаем дерево полным самым очевидным способом. Как и операция Вставить в обратную сторону, мы знаем, что последний узел дерева должен идти куда-то в другое место. Но куда? Поскольку мы все равно извлекаем корень, давайте перезапишем старый корневой узел тем, что раньше было последним узлом. Например, начиная с кучи



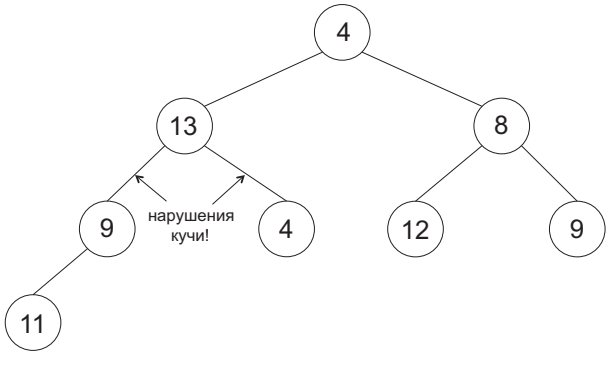
результирующее дерево выглядит следующим образом:



Хорошей новостью является то, что мы восстановили свойство полного бинарного дерева.

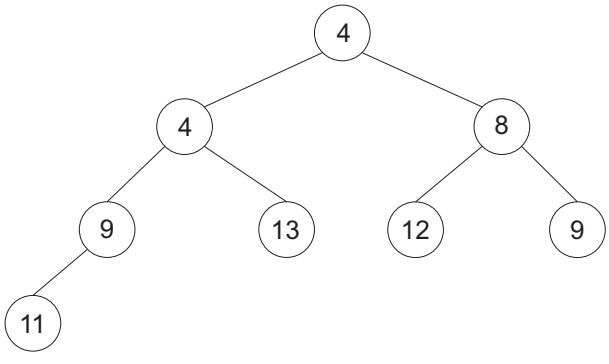
Плохая новость заключается в том, что массовое содействие, предоставленное объекту с ключом 13, создало две нарушающие пары «родитель — потомок» (13 и 4 и 13 и 8). Нужны ли нам две перестановки, чтобы их исправить?

Основная идея состоит в том, чтобы переставить местами корневой узел и *меньшего* из его двух потомков:



Никаких нарушений кучи, связанных с корнем, больше нет — новый корневой узел меньше обоих узлов: узла, который был им заменен (вот почему мы вы-

полнили перестановку местами), и его другого потомка (так как мы поменяли меньшего потомка)[1]. Нарушения кучи мигрируют вниз, снова вовлекая объект с ключом 13 и его двух (новых) потомков. Поэтому мы делаем это снова и переставляем местами 13 и его меньшего потомка:



Свойство кучи наконец восстановлено, и теперь извлечение завершено.

В общем случае операция Извлечь минимум перемещает последний объект кучи в корневой узел (перезаписывая предыдущий корень) и повторно переставляет местами этот объект с его меньшим потомком[2]. В любой момент времени существует не более двух нарушающих пар «родитель — потомок» — двух пар, в которых ранее последний объект является родителем[3]. Поскольку каждая перестановка проталкивает этот объект вниз на один уровень дерева, этот процесс не может продолжаться вечно — он останавливается, как только новый объект принадлежит последнему уровню, если не раньше.

---

[1] Перестановке 13 и 8 не удастся избавить левое поддерево от нарушений кучи (с нарушающей парой 8 и 4), позволяя болезни распространиться на правое поддерево (с нарушающей парой 13 и 12 и парой 13 и 9).

[2] Эта подпрограмма перестановки местами, среди прочего, в англоязычной литературе называется `Bubble-Down` (всплытие вниз).

[3] Каждая пара «родитель — потомок», не включающая этот ранее последний объект, появлялась в исходной куче и, следовательно, не нарушает свойство кучи. Также нет никаких нарушений, связанных с этим объектом и его родителем, — изначально у него не было родителя, и впоследствии он, двигаясь вниз, меняется местами с объектами, которые имеют меньшие ключи.

ИЗВЛЕЧЬ МИНИМУМ

1. Перезаписать корень последним объектом *x* в куче и уменьшить размер кучи.
2. Многократно переставлять местами *x* и его меньшего потомка, до тех пор пока свойство кучи не будет восстановлено.

Число перестановок не превышает число уровней, и на каждую перестановку требуется только постоянный объем работы. Поскольку существует $\approx \log_2 n$ уровней, мы заключаем, что время выполнения операции Извлечь минимум в худшем случае равно $O(\log n)$, где n — это число объектов в куче.

ВЫВОДЫ

- Существует целый ряд отличных друг от друга структур данных, каждая из которых оптимизирована для разных множеств операций.
- Принцип экономии рекомендует выбирать простейшую структуру данных, которая поддерживает все операции, требуемые вашим приложением.
- Если ваше приложение требует быстрых вычислений минимума (или максимума) на эволюционирующем множестве объектов, то куча обычно является предпочтительной структурой данных.
- Две наиболее важные кучевые операции, Вставить и Извлечь минимум, выполняются за время $O(\log n)$, где n — это число объектов.
- Кучи также поддерживают операции Найти минимум за время $O(1)$, Удалить за время $O(\log n)$ и Объединить в кучу за время $O(n)$.
- Алгоритм `HeapSort` использует кучу для сортировки массива длины n за время $O(n \log n)$.
- Кучи могут быть использованы для реализации алгоритма кратчайшего пути Дейкстры за время $O((m + n) \log n)$, где m и n обозначают число ребер и вершин графа соответственно.

- ✶ Кучи могут быть визуализированы как полные бинарные деревья, но реализованы в виде массивов.
- ✶ Свойство кучи указывает на то, что ключ каждого объекта меньше или равен ключам его потомков.
- ✶ Операции Вставить и Извлечь минимум реализуются путем поддержания дерева полным наиболее очевидным способом из возможных и систематического подавления любых нарушений свойства кучи.

Задачи на закрепление материала

Задача 10.1. Какой из следующих шаблонов в компьютерной программе предполагает, что кучевая структура данных может обеспечить значительное ускорение? (Отметьте все подходящие варианты.)

а) Повторные операции поиска.

б) Многократные вычисления минимума.

в) Многократные вычисления максимума.

г) Ни один из вариантов.

Задача 10.2. Предположим, что вы реализовали функциональность очереди с приоритетом (то есть операции Вставить и Извлечь минимум) с использованием массива, отсортированного от наибольшего к наименьшему. Каково время работы операций Вставить и Извлечь минимум соответственно в худшем случае? Будем считать, что у вас имеется достаточно большой массив, который позволяет разместить все вставки.

а) $\Theta(1)$ и $\Theta(n)$

б) $\Theta(n)$ и $\Theta(1)$

в) $\Theta(\log n)$ и $\Theta(1)$

г) $\Theta(n)$ и $\Theta(n)$

Задача 10.3. Предположим, что функциональность очереди с приоритетом (то есть операции Вставить и Извлечь минимум) реализована с помощью

неотсортированного массива. Каково время выполнения операций Вставить и Извлечь минимум соответственно в худшем случае? Будем считать, что у вас имеется достаточно большой массив, который позволяет разместить все вставки.

а) $\Theta(1)$ и $\Theta(n)$

б) $\Theta(n)$ и $\Theta(1)$

в) $\Theta(1)$ и $\Theta(\log n)$

г) $\Theta(n)$ и $\Theta(n)$

Задача 10.4. Дана куча с n объектами. Какие из следующих задач можно решить с помощью операций Вставить и Извлечь минимум с временем $O(1)$ и дополнительной работы с временем $O(1)$? (Выберите все подходящие варианты.)

а) Найти объект, хранящийся в куче с пятым наименьшим ключом.

б) Найти объект, хранящийся в куче с максимальным ключом.

в) Найти объект, хранящийся в куче с медианным ключом.

г) Ничего из перечисленного выше.

Задачи повышенной сложности

Задача 10.5. В продолжение задачи 9.7 покажите, как модифицировать реализацию кучевого алгоритма Дейкстры, для того чтобы вычислить для каждой вершины $v \in V$ наименьшее узкое место пути s-v. Ваш алгоритм должен работать за время $O((m + n) \log n)$, где m и n обозначают число ребер и вершин соответственно.

Задача 10.6 (сложная). Мы можем достичь большего. Предположим, что граф является неориентированным. Дайте линейно-временной алгоритм (то есть работающий за время $O(m + n)$) для вычисления пути с минимальным узким местом между двумя заданными вершинами.

[Подсказка: пригодится линейно-временной алгоритм из первой части. В рекурсии цель состоит в том, чтобы сократить размер входных данных вдвое за линейное время.]

Задача 10.7 (сложная). Что делать, если граф является ориентированным? Можете ли вы вычислить путь с минимальным узким местом между двумя заданными вершинами менее чем за время $O((m + n) \log n)$?[1]

Задача по программированию

Задача 10.8. Реализуйте на своем любимом языке программирования кучевую версию алгоритма Дейкстры из раздела 10.4 и используйте ее для решения задачи о кратчайшем пути с единственным истоком в разных ориентированных графах. В случае с этой кучевой реализации каков размер самой большой задачи, которую можно решить за пять минут или меньший временной срок? (Обратитесь к сайту www.algorithmsilluminated.org для тестовых случаев и совокупностей данных задач.)

[Подсказка: для этого потребуется операция Удалить, которая может побудить вас реализовать собственную кучевую структуру данных с нуля. Для того чтобы удалить объект из кучи в заданной позиции, следуйте высокоуровневому подходу операций Вставить и Извлечь минимум, по мере необходимости используя подпрограммы всплытия вверх `Bubble-Up` или всплытия вниз `Bubble-Down` с целью устранения нарушений свойства кучи. Вам также нужно будет отслеживать, какая вершина находится в какой позиции кучи, возможно, применяя хеш-таблицы (глава 12).]

[1] Для глубокого погружения в эту задачу см. работу *«Algorithms for Two Bottleneck Optimization Problems»* («Алгоритмы для двух оптимизационных задач с узкими местами») *Гарольда Х. Гэбоу и Роберта Тарьяна* (*Journal of Algorithms, 1988*).

11 Дерево поиска

Дерево поиска, как и куча, представляет собой структуру данных для хранения эволюционирующего множества объектов, ассоциированных с ключами (и, возможно, с большим количеством других данных). Она поддерживает полное упорядочивание хранимых объектов и может поддерживать более богатое множество операций, чем куча, за счет увеличенного пространства и, в случае некоторых операций, несколько более медленного времени выполнения. Мы начнем с ответа на вопрос «что» (то есть поддерживаемые операции) и только после этого перейдем к ответу на вопросы «почему» (применения) и «как» (дополнительные детали реализации).

11.1. Отсортированные массивы

Неплохая идея представлять дерево поиска как динамическую версию отсортированного массива — оно может делать то же, что и отсортированный массив, а также способно быстро производить операции вставки и удаления.

11.1.1. Отсортированные массивы: поддерживаемые операции

С отсортированным массивом вы много чего можете.

ОТСОРТИРОВАННЫЕ МАССИВЫ: ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ

Отыскать: для ключа k вернуть указатель на объект в структуре данных с ключом k (либо сообщить, что такого объекта не существует).

Минимум (Максимум): вернуть указатель на объект в структуре данных с наименьшим (либо, соответственно, наибольшим) ключом.

Предшественник (Преемник): по указателю на объект в структуре данных вернуть указатель на объект со следующим наименьшим (соответственно следующим наибольшим) ключом. Если данный объект имеет минимальный (соответственно, максимальный) ключ, то сообщить «пусто» (или «none»).

Вывести в отсортированном порядке: вывести объекты в структуре данных по одному в порядке их ключей.

Выбрать: по заданному числу i между 1 и числом объектов вернуть указатель на объект в структуре данных с i-м наименьшим ключом.

Взять ранг: по заданному ключу k вернуть число объектов в структуре данных с ключом не выше k.

Рассмотрим на следующем примере, как реализовать каждую из этих операций:

3	6	10	11	17	23	30	36

- В операции Отыскать используется двоичный поиск: сначала проверить, имеет ли объект в срединной позиции массива нужный ключ. Если это так, то вернуть его. Если нет, то выполнить рекурсию в левой половине (если ключ срединного объекта — слишком большой) либо в правой половине (если он — слишком малый)[1]. Например, чтобы выполнить поиск ключа 8 в приведенном выше массиве, двоичный поиск проверит четвертый объект (с ключом 11); выполнит рекурсию на левой половине (объекты с ключами 3, 6 и 10); проверит второй объект (с ключом 6); выполнит рекурсию на правой половине оставшегося массива (объект с ключом 10); заключит, что правильная позиция объекта с ключом 8 будет между вторым и третьим объектами; и сообщит «пусто». Так как каждый рекурсивный вызов сокращает размер массива в 2 раза, существует не более $\log_2 n$ рекурсивных вызовов, где n — это длина массива. Поскольку каждый рекурсивный вызов выполняет постоянный объем работы, данная операция выполняется за время $O(\log n)$.
- Операции Минимум и Максимум легко реализовать со временем $O(1)$: вернуть указатель на первый или последний объект в массиве соответственно.

[1] Читателям постарше он наверняка напомнит поиск телефонного номера в телефонной книге. Если вы еще не просматривали исходный код этого алгоритма, то загляните в свою любимую вводную книгу либо учебник по программированию.

- Для того чтобы реализовать операции Предшественник или Преемник, следует применить операцию Отыскать, которая восстановит позицию данного объекта в отсортированном массиве, и вернуть объект в предыдущей или следующей позиции соответственно. Эти операции выполняются так же быстро, как и поиск — за время $O(\log n)$, где n — это длина массива.
- Операция Вывести в отсортированном порядке тривиальна для линейно-временной реализации с отсортированным массивом: выполнить один проход вперед-назад по массиву, выводя каждый объект по очереди.
- Операция Выбрать легко реализуется за постоянное время: по заданному индексу i вернуть объект в i-й позиции массива.
- Операция Взять ранг похожа на обратную операцию Выбрать и может быть реализована схожим образом, что и операция Отыскать: если бинарный поиск находит объект с ключом k в i-й позиции массива либо если он обнаруживает, что k находится между ключами объектов в i-й и $(i + 1)$-й позициях, то правильным ответом будет i[1].

Подводя итог, приведем окончательный перечень показателей для отсортированных массивов.

Таблица 11.1. Отсортированные массивы: поддерживаемые операции и время их выполнения, где n обозначает текущее число объектов, хранящихся в массиве

Операция	Время выполнения
Отыскать	$O(\log n)$
Минимум	$O(1)$
Максимум	$O(1)$
Предшественник	$O(\log n)$
Преемник	$O(\log n)$
Вывести в отсортированном порядке	$O(n)$
Выбрать	$O(1)$
Взять ранг	$O(\log n)$

[1] Это упрощенное описание исходит из того, что повторяющиеся ключи отсутствуют. Какие изменения необходимы для размещения нескольких объектов с одним ключом?

11.1.2. Неподдерживаемые операции

Нужно ли мечтать о чем-то большем? В случае со *статическим* набором данных, который не изменяется со временем, этот список поддерживаемых операций впечатляет. Однако многие реальные приложения *динамичны*, а множество релевантных объектов с течением времени эволюционирует. К примеру, сотрудники приходят и уходят, и структура данных, в которой хранятся их записи, должна оставаться актуальной. По этой причине нас также интересуют вставки и удаления.

ОТСОРТИРОВАННЫЕ МАССИВЫ: НЕПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ

Вставить: с учетом нового объекта x добавить x в структуру данных.

Удалить: для ключа k удалить объект с ключом k из структуры данных, если он существует[1].

Эти две операции вполне можно реализовать с отсортированным массивом, но они тягостно медленные — вставка или удаление элемента при сохранении свойства отсортированного массива требует линейного времени в самом неблагоприятном случае. Существует ли альтернативная структура данных, которая повторяет все функциональные возможности отсортированного массива, одновременно соответствуя логарифмически-временной результативности кучи для операций Вставить и Удалить?

[1] Внимательный читатель, возможно, заметил, что эта спецификация операции Удалить (которая на входе принимает ключ) отличается от той же для куч (которая на входе принимает указатель на объект). Причина заключается в том, что кучи не поддерживают быстрый поиск. В отсортированном массиве (а также в деревьях поиска и хеш-таблицах) указатель на объект легко восстанавливается по его ключу (через операцию Отыскать).

11.2. Деревья поиска: поддерживаемые операции

Смысл дерева поиска заключается в поддержке всех операций, поддерживаемых отсортированным массивом, а также вставок и удалений. Все операции, кроме Вывести в отсортированном порядке, выполняются за время $O(\log n)$, где n — это число объектов в дереве поиска. Операция Вывести в отсортированном порядке выполняется за время $O(n)$, и лучшего просто не бывает (так как она должна выводить n объектов).

Приведем перечень показателей для деревьев поиска в сравнении с отсортированными массивами.

Таблица 11.2. Сбалансированные деревья поиска и отсортированные массивы: поддерживаемые операции и их время выполнения, где n обозначает текущее число объектов, хранящихся в структуре данных

Операция	Отсортированный массив	Сбалансированное дерево поиска
Отыскать	$O(\log n)$	$O(\log n)$
Минимум	$O(1)$	$O(\log n)$
Максимум	$O(1)$	$O(\log n)$
Предшественник	$O(\log n)$	$O(\log n)$
Преемник	$O(\log n)$	$O(\log n)$
Вывести в отсортированном порядке	$O(n)$	$O(n)$
Выбрать	$O(1)$	$O(\log n)$
Взять ранг	$O(\log n)$	$O(\log n)$
Вставить	$O(n)$	$O(\log n)$
Удалить	$O(n)$	$O(\log n)$

Важный нюанс: время выполнения в табл. 11.2 достигаются *сбалансированным* деревом поиска, то есть более сложной версией стандартного бинарного

дерева поиска[1], описанного в разделе 11.3. Несбалансированное дерево поиска не гарантирует такое время выполнения[2].

> **КОГДА ИСПОЛЬЗОВАТЬ СБАЛАНСИРОВАННОЕ ДЕРЕВО ПОИСКА**
>
> Если ваше приложение нуждается в поддержке упорядоченного представления динамически изменяющегося множества объектов, то сбалансированное дерево поиска (или структура данных на его основе[3]) обычно является предпочтительной структурой данных[4].

Вспомните принцип экономии — выбирать простейшую структуру данных, которая поддерживает все операции, требуемые вашим приложением. Если вам нужно поддерживать только упорядоченное представление статической совокупности данных (без вставок и удалений), используйте вместо сбалансированного дерева поиска отсортированный массив; первое было бы излишним. Если набор данных является динамическим, но вас интересуют только быстрые операции взятия минимума (либо максимума), то вместо сбалансированного дерева поиска используйте кучу. Эти более простые структуры данных делают меньше, чем сбалансированное дерево поиска, но то, что они

---

[1] Бинарное дерево поиска (binary search tree) нередко еще называется отсортированным бинарным деревом и является корневым бинарным деревом. См. https://en.wikipedia.org/wiki/Binary_search_tree. — *Примеч. пер.*

[2] Предварительный обзор разделов 11.3 и 11.4: как правило, операции поиска по дереву выполняются во времени, пропорциональном высоте дерева, имея в виду самый длинный путь от корня дерева до одного из его листьев. В бинарном дереве с n узлами высота может быть где угодно — от $\approx \log_2 n$ (если дерево идеально сбалансировано) до $n - 1$ (если узлы образуют единую цепочку). Сбалансированные деревья поиска выполняют скромный объем дополнительной работы, тем самым гарантируя, что высота всегда равна $O(\log n)$; эта гарантия высоты затем приводит к границам времени выполнения, приведенным в табл. 11.2.

[3] Например, класс `TreeMap` в Java и шаблон класса `map` в стандартной библиотеке шаблонов C++ построены поверх сбалансированных деревьев поиска.

[4] Хорошим местом, где можно увидеть сбалансированные деревья поиска в естественных условиях, является ядро Linux. Например, они используются для управления планированием процессов и отслеживания объема виртуальной памяти каждого процесса.

делают, удается им лучше — быстрее (на постоянный или логарифмический множитель) и с меньшим пространством (на постоянный множитель)[1].

*11.3. Детали реализации

Этот раздел предоставляет высокоуровневое описание типичной реализации (не обязательно сбалансированного) бинарного дерева поиска. Раздел 11.4 затрагивает некоторые дополнительные идеи, необходимые для сбалансированных деревьев поиска[2].

11.3.1. Свойство дерева поиска

В бинарном дереве поиска каждый узел соответствует объекту (с ключом) и имеет три ассоциированных с ним указателя: указатель на родителя, указатель на левого потомка и указатель на правого потомка. Любой из этих указателей может быть пустым, то есть иметь значение null, указывающее на отсутствие родителя или потомка. Левое поддерево узла *x* содержит узлы, доступные из *x* через его указатель на левого потомка, и схожим образом для правого поддерева. Определяющим *свойством дерева поиска* является следующее:

СВОЙСТВО ДЕРЕВА ПОИСКА

1. Для каждого объекта *x* объекты в левом поддереве объекта *x* имеют ключи меньше, чем у *x*.
2. Для каждого объекта *x* объекты в правом поддереве объекта *x* имеют ключи больше, чем у *x*[3].

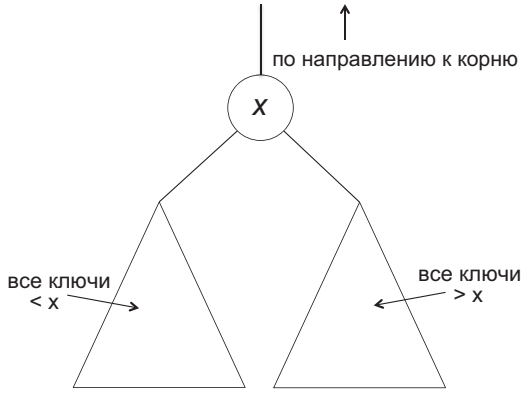
---

[1] Глава 12 охватывает хеш-таблицы, которые выполняют еще меньше работы; но то, что они делают, они делают еще лучше (за постоянное время для всех практических целей).

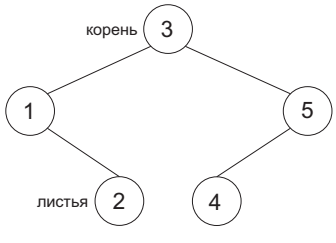
[2] Мы говорим об узлах и соответствующих объектах взаимозаменяемо.

[3] Это свойство исходит из того, что никакие два объекта не имеют одинакового ключа. Для размещения повторяющихся ключей измените значение «меньше» в первом условии на «меньше или равно».

Свойство дерева поиска накладывает требование на каждый узел дерева поиска, а не только на корень:



Например, покажем дерево поиска, содержащее объекты с ключами {1, 2, 3, 4, 5}, и таблицу, перечисляющую места назначения трех указателей в каждом узле:



а) Дерево поиска

Узел	Родитель	Левый	Правый
1	3	null	2
2	1	null	null
3	null	1	5
4	5	null	null
5	3	4	null

б) Указатели

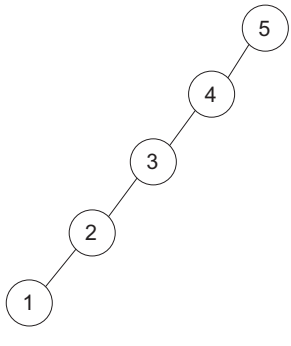
Рис. 11.1. Дерево поиска и соответствующие ему указатели на родителя и потомков

Бинарные деревья поиска и кучи различаются по нескольким направлениям. Кучи можно рассматривать как деревья, но они реализованы как массивы, без явных указателей между объектами. Дерево поиска явным образом хранит три указателя в расчете на объект и, следовательно, использует больше места (с постоянным множителем). Кучам явные указатели не нужны, потому что они всегда соответствуют полным бинарным деревьям, в то время как бинарные деревья поиска могут иметь произвольную структуру.

Деревья поиска имеют иное предназначение, чем кучи. По этой причине свойство дерева поиска несопоставимо со свойством кучи. Кучи оптимизированы для быстрых вычислений минимума, а свойство кучи — то, что ключ потомка больше только ключа родителя, — упрощает поиск объекта с минимальным ключом (это корень). Деревья поиска оптимизированы — приготовьтесь, готовы? — для поиска, и свойство дерева поиска определено соответствующим образом. Например, если вы ищете объект с ключом 23 в дереве поиска и ключом корня является 17, то вы знаете, что объект может располагаться только в правом поддереве корня, и можете исключить объекты в левом поддереве из дальнейшего рассмотрения. Это должно напомнить вам о двоичном (бинарном) поиске, как и подобает структуре данных, смысл которой заключается в симулировании динамически изменяющегося отсортированного массива.

11.3.2. Высота дерева поиска

Для заданного множества ключей существует множество разных деревьев поиска. Покажем второе дерево поиска, содержащее объекты с ключами {1, 2, 3, 4, 5}:



Оба условия в свойстве дерева поиска соблюдаются, второе — бессодержательно (так как непустых правых поддеревьев нет).

Высота дерева определяется как длина самого длинного пути от корня до листа[1]. Разные деревья поиска, содержащие идентичные множества объектов, могут иметь разную высоту, как и в первых двух примерах (где деревья име-

[1] Также именуется *глубиной* дерева.

ют высоту соответственно 2 и 4). В общем случае бинарное дерево поиска, содержащее n объектов, может иметь какую угодно высоту от

$$\underbrace{\approx \log_2 n}_{\text{идеально сбалансированное бинарное дерево (наилучший сценарий)}} \quad \text{до} \quad \underbrace{n-1}_{\text{цепочка, как показано выше (наихудший сценарий)}}$$

Остальная часть этого раздела описывает то, как реализуются все операции бинарного дерева поиска во времени, пропорциональном высоте дерева (за исключением операции Вывести в отсортированном порядке, которая выполняется за время, линейное по n). Что касается уточнений бинарных деревьев поиска, которые гарантированно имеют высоту $O(\log n)$ (см. раздел 11.4), это приводит к логарифмическому времени выполнения, отраженному в перечне показателей из табл. 11.2.

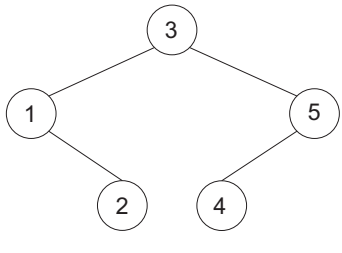
11.3.3. Реализация операции «Отыскать» со временем O (высота)

Давайте начнем с операции Отыскать:

> по ключу k вернуть указатель на объект в структуре данных с ключом k (либо сообщить, что такого объекта не существует).

Свойство дерева поиска указывает, где именно искать объект с ключом k. Если k меньше (либо, соответственно, больше) ключа корня, то такой объект должен располагаться в левом поддереве (в правом дереве соответственно) корня. Для того чтобы выполнить поиск, следуйте по прямой: начните с корня и несколько раз пройдите влево или вправо (если это необходимо), до тех пор пока вы не найдете нужный объект (успешный результат поиска) или не встретите пустой указатель (безуспешный поиск).

Например, предположим, что мы ищем объект с ключом 2 в нашем первом бинарном дереве поиска:



Поскольку ключ корня (3) — слишком большой, первый шаг проходит по указателю на левого потомка. Поскольку ключ следующего узла — слишком мал (1), второй шаг проходит по указателю на правого потомка, достигая нужного объекта. Если мы ищем объект с ключом 6, то поиск проходит по указателю на правого потомка корня (так как ключ корня слишком мал). Раз ключ следующего узла (5) также слишком мал, поиск пытается последовать по еще одному указателю на правого потомка, встречает пустой указатель и останавливается (безуспешно).

ОТЫСКАТЬ

1. Начать с корневого узла.
2. Многократно пройти по указателям на левого и правого потомка, если это необходимо (левый, если k меньше ключа текущего узла; правый, если k больше).
3. Вернуть указатель на объект с ключом k (если он найден) либо «пусто» (при достижении пустого указателя).

Время выполнения пропорционально количеству пройденных указателей, которое не превышает высоту дерева поиска (плюс 1, если считать последний пустой указатель безуспешного поиска).

11.3.4. Реализация операций «Минимум» и «Максимум» за время O(высота)

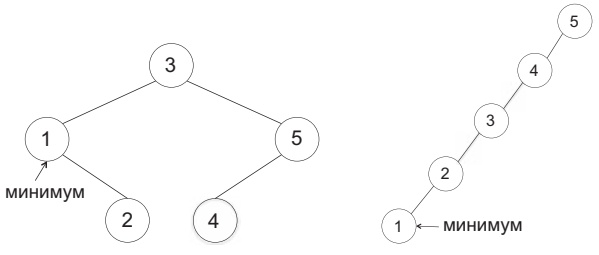
Свойство дерева поиска упрощает реализацию операций Минимум и Максимум.

> Минимум (Максимум): вернуть указатель на объект в структуре данных с наименьшим (либо, соответственно, наибольшим) ключом.

Ключи в левом поддереве корня могут быть только меньше ключа корня, и ключи в правом поддереве могут быть только больше. Если левое поддерево является пустым, то корень должен быть минимальным. В противном случае минимум левого поддерева также является минимумом всего дерева. Это на-

водит на мысль о том, чтобы следовать по указателю на левого потомка корня и повторять данный процесс.

Например, в деревьях поиска, которые мы рассматривали ранее:



многократное следование по указателям на левого потомка приводит к объекту с минимальным ключом.

МИНИМУМ (МАКСИМУМ)

1. Начать с корневого узла.
2. Пройти по указателям на левого потомка (по указателям на правого потомка) как можно дольше, до тех пор пока не встретится пустой указатель.
3. Вернуть указатель на последний посещенный объект.

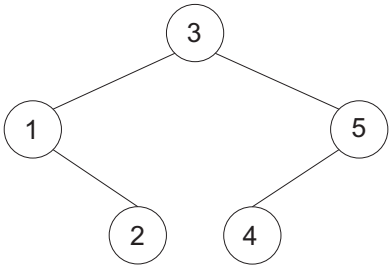
Время работы пропорционально числу пройденных указателей, которое равно O(*высота*).

11.3.5. Реализация операции «Предшественник» со временем O (высота)

Следующей идет операция Предшественник; реализация операции Преемник является похожей.

> Предшественник: по указателю на объект в структуре данных вернуть указатель на объект со следующим наименьшим значением ключа. (Если объект имеет минимальный ключ, то сообщить «пусто».)

Если учитывать объект x, то где может располагаться предшественник объекта x? Не в правом поддереве объекта x, где все ключи больше ключа объекта x (по свойству дерева поиска). Наш рабочий пример



иллюстрирует два случая. Предшественник может появиться в левом поддереве (как в случае узлов с ключами 3 и 5), либо он может быть предком дальше вверх по дереву (как в случае узлов с ключами 2 и 4).

Общий шаблон таков: если левое поддерево объекта x не пустое, то максимальным элементом этого поддерева является предшественник объекта x[1]; в противном случае предшественником объекта x является ближайший предок объекта x, который имеет меньший ключ, чем x. Равным образом, проходя по указателям на родителя по направлению вверх от x, местом назначения будет первый левый поворот[2]. Так, в приведенном выше дереве поиска трассировка указателей на родителя по направлению вверх от узла с ключом 4 сначала делает правый поворот (ведущий к узлу с большим ключом 5), а затем делает левый поворот, приходя к правильному предшественнику (3). Если x имеет пустое левое поддерево и над ним нет левых поворотов, то он является минимумом в дереве поиска и не имеет предшественника (как узел с ключом 1 в приведенном выше дереве поиска).

---

[1] Среди ключей, меньших x, те из них, что находятся в левом поддереве x, являются ближайшими к x (как вы сами можете убедиться). Среди ключей в этом поддереве максимальный находится ближе всего к x.

[2] Правые повороты могут вести только к узлам с бóльшими ключами, которые не могут быть предшественниками x. Из свойства дерева поиска также вытекает, что ни более отдаленные предки, ни не предки не могут быть предшественниками x (как вы сами можете убедиться).

ПРЕДШЕСТВЕННИК

1. Если левое поддерево x не пустое, то вернуть результат операции Максимум, примененной к этому поддереву.
2. В противном случае пройти по указателям на родителя вверх по направлению к корню. Если обход посещает последовательные узлы y и z, где узел y является правым потомком узла z, то вернуть указатель на z.
3. В противном случае сообщить «пусто».

Время работы данной операции является пропорциональным числу пройденных указателей, которое во всех случаях равняется O(*высота*).

11.3.6. Реализация операции «Вывести в отсортированном порядке» со временем O(n)

Напомним операцию Вывести в отсортированном порядке:

> вывести объекты в структуре данных по одному в порядке следования их ключей.

Ленивый способ реализации этой операции состоит в том, чтобы сначала применить операцию Минимум для вывода объекта с минимальным ключом, а затем повторно вызывать операцию Преемник для вывода остальных объектов по порядку. Оптимальным методом является использование так называемого *инфиксного обхода* дерева поиска, которое рекурсивно обрабатывает левое поддерево корня, затем корень, а потом — правое поддерево корня. Эта идея идеально сочетается со свойством дерева поиска, из которого вытекает, что операция Вывести в отсортированном порядке должна сначала выводить по порядку объекты в левом поддереве корня, затем объект в корне и затем по порядку объекты в правом поддереве корня.

ВЫВЕСТИ В ОТСОРТИРОВАННОМ ПОРЯДКЕ

1. Рекурсивно вызывать операцию Вывести в отсортированном порядке на левом поддереве корня.

2. Вывести объект в корне.
3. Рекурсивно вызывать операцию Вывести в отсортированном порядке на правом поддереве корня.

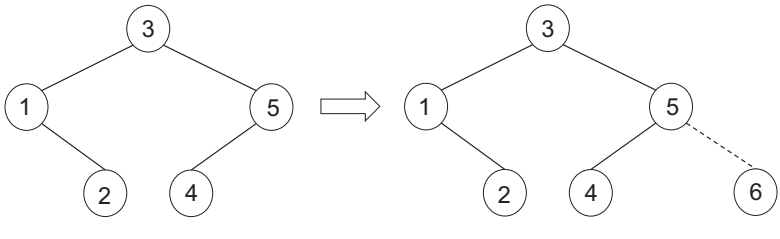
Для дерева, содержащего n объектов, данная операция выполняет n рекурсивных вызовов (по одному на каждом узле) и выполняет в каждом из них постоянный объем работы за суммарное время выполнения $O(n)$.

11.3.7. Реализация операции «Вставить» со временем O(высота)

Ни одна из операций, рассмотренных до сих пор, не изменяет данное дерево поиска, поэтому они не рискуют испортить критическое свойство дерева поиска. Следующие две операции — Вставить и Удалить — вносят изменения в дерево и должны сохранять свойство дерева поиска.

> Вставить: с учетом нового объекта x добавить x в структуру данных.

Операция вставки выезжает за счет операции Отыскать. Безуспешный поиск объекта с ключом k определяет местоположение, где такой объект должен был появиться. Это место подходит для прикрепления нового объекта с ключом k (перезапись старого пустого указателя). В нашем рабочем примере правильным местоположением нового объекта с ключом 6 является место, где завершился безуспешный поиск:



Что делать, если в дереве уже есть объект с ключом k? Если вы хотите избежать дублирования ключей, вставку можно проигнорировать. В против-

ном случае поиск проследует по левому потомку существующего объекта с ключом *k*, продвигаясь вперед до тех пор, пока не будет обнаружен пустой указатель.

ВСТАВИТЬ

1. Начать с корневого узла.
2. Многократно пройти по указателям левого и правого потомков, если это необходимо (левый, если *k* не больше ключа текущего узла, правый, если больше), до тех пор пока не встретится пустой указатель.
3. Заменить пустой указатель на новый объект. Установить указатель родителя нового узла равным его родителю и указатели на потомков, равными «пусто», то есть *null*.

Данная операция сохраняет свойство дерева поиска, поскольку она помещает новый объект туда, где он должен был находиться[1]. Ее время выполнения совпадает с временем операции Отыскать, которое равно $O(высота)$.

11.3.8. Реализация операции «Удалить» со временем O (высота)

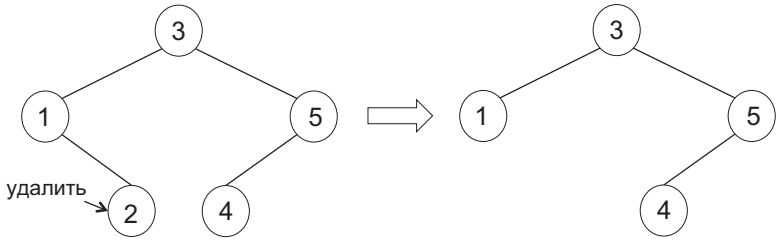
В большинстве структур данных операция Удалить является наиболее сложной. Деревья поиска не являются исключением.

> Удалить: по ключу *k* удалить объект с ключом *k* из дерева поиска, если он существует.

[1] Более формально: обозначим через *x* только что вставленный объект и рассмотрим существующий объект *y*. Если *x* не является членом поддерева с корнем в *y*, то он не может вмешиваться в свойство дерева поиска в *y*. Если он является членом поддерева с корнем в *y*, то *y* был одним из узлов, посещенных во время безуспешного поиска объекта *x*. Ключи объектов *x* и *y* явным образом сопоставлялись в этом поиске, причем *x* был помещен в левом поддереве *y* тогда и только тогда, когда его ключ не больше, чем у объекта *y*.

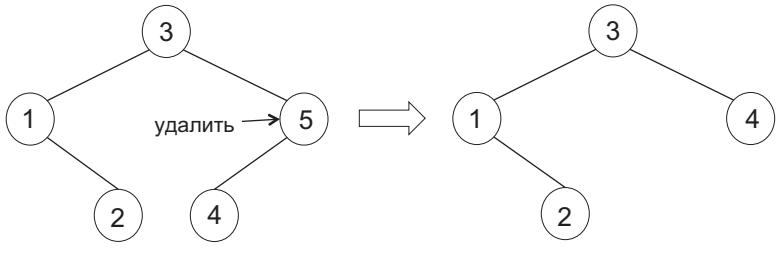
Главную трудность представляет исправление дерева после удаления узла с целью восстановления свойства дерева поиска.

Первый шаг состоит в вызове операции Отыскать, для того чтобы найти объект x с ключом k. (Если такого объекта нет, то операции Удалить не применяется.) Существует три случая, в зависимости от того, имеет ли x 0, 1 или 2 потомков. Если x является листом иерархического дерева, то его можно удалить без вреда. Например, в случае, если мы удалим узел с ключом 2 из нашего любимого дерева поиска:



Для каждого оставшегося узла y узлы в поддеревьях узла y остались, как и раньше, за исключением, возможно, удаленного узла x; свойство дерева поиска продолжает соблюдаться.

Когда узел x имеет одного потомка y, мы можем его отсоединить. Удаление узла x оставляет узел y без родителя, а старого родителя z узла x без одного из его потомков. Очевидным исправлением является то, чтобы дать узлу y принять предыдущую позицию узла x (в качестве потомка узла z)[1]. Например, в случае, если мы удалим узел с ключом 5 из нашего любимого дерева поиска:



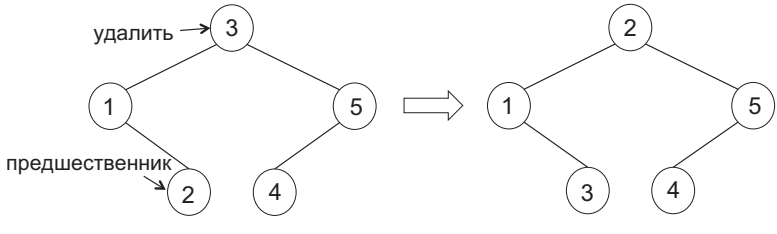
[1] Место для вашей шутки про поколение Z.

По той же логике, что и в первом случае, свойство поиска соблюдается.

Трудный случай встречается, когда x имеет *двух* потомков. Удаление узла x оставляет два узла без родителя, и неясно, куда их поместить. В нашем рабочем примере совершенно не очевидно, как исправлять дерево после удаления его корня.

Ключевая хитрость состоит в том, чтобы свести трудный случай к одному из легких. Сначала применить операцию Предшественник для вычисления предшествующего узлу x узла y[1]. Поскольку узел x имеет двух потомков, его предшественником является объект в его (непустом!) левом поддереве с максимальным ключом (см. раздел 11.3.5). Поскольку максимум вычисляется, следуя по указателям на правого потомка как можно дольше (см. раздел 11.3.4), узел y не может иметь правого потомка; он может иметь либо не иметь левого потомка.

Сумасшедшая идея: *поменять местами* узлы x и y! В нашем рабочем примере, в котором узел x выступает в качестве корневого узла:



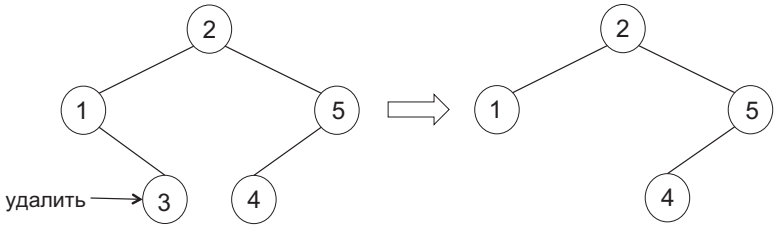
Эта сумасшедшая идея выглядит никудышной, так как мы нарушили свойство дерева поиска (где узел с ключом 3 находится в левом поддереве узла с ключом 2). Однако каждое нарушение свойства дерева поиска связано с узлом x, который мы все равно удалим[2]. Поскольку узел x теперь занимает

---

[1] При желании преемник тоже отлично работает.

[2] Для каждого узла z, отличного от узла y, единственным возможным новым узлом в поддереве узла z является узел x. Между тем узел y, как непосредственный предшественник узла x в отсортированном упорядочении всех ключей, имеет ключ больше, чем те, которые в старом левом поддереве узла x, и больше, чем те, которые в старом правом поддереве узла x. Таким образом, условие дерева поиска соблюдается для узла y в его новой позиции, кроме как по отношению к узлу x.

предыдущую позицию узла *y*, он больше не имеет правого потомка. Удаление узла *x* из его новой позиции приводит к одному из двух простых случаев: мы удаляем его, если он также не имеет левого потомка, и отсоединяем его, если он имеет левого потомка. В любом случае, когда узел *x* выходит из игры, свойство дерева поиска восстанавливается. Вернемся к нашему примеру:



УДАЛИТЬ

1. Применить операцию Отыскать, для того чтобы найти объект *x* с ключом *k*. (Если такой объект не существует, то остановиться.)
2. Если *x* не имеет потомков, удалить *x* путем установки соответствующего указателя на потомка в родителе узла *x* равным «пусто». (Если *x* был корнем, то новое дерево является пустым.)
3. Если *x* имеет одного потомка, то отсоединить *x*, заменив соответствующий указатель потомком в родителе узла *x* с потомком узла *x*, а указатель на родителя потомка узла *x* родителем узла *x*. (Если *x* был корнем, то его потомок становится новым корнем.)
4. В противном случае заменить *x* объектом в его левом поддереве, который имеет самый большой ключ, и удалить *x* из его новой позиции (где он имеет не более одного потомка).

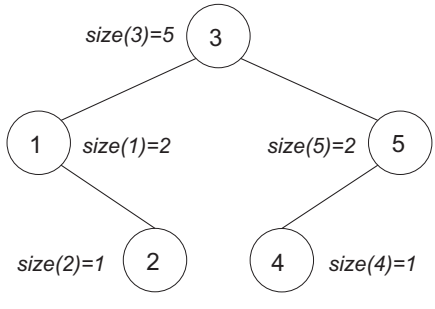
Данная операция выполняет постоянный объем работы в дополнение к одной операции Отыскать и одной операции Предшественник, поэтому она выполняется за время $O(высота)$.

11.3.9. Расширенные деревья поиска для операции «Выбрать»

Наконец, операция Выбрать:

> по числу i от 1 до числа объектов вернуть указатель на объект в структуре данных с i-м наименьшим ключом.

Для того чтобы операция Выбрать выполнялась быстро, мы *усилим* дерево поиска тем, что каждый узел будет отслеживать информацию *о структуре самого дерева*, а не только об объекте[1]. Деревья поиска могут быть усилены многими способами; здесь мы будем хранить в каждом узле x целочисленный размер $size\,(x)$, указывающий на число узлов в поддереве с корнем в узле x (включая сам узел x). В нашем рабочем примере



Мы имеем $size\,(1) = 2$, $size\,(2) = 1$, $size\,(3) = 5$, $size\,(4) = 1$ и $size\,(5) = 2$.

ТЕСТОВОЕ ЗАДАНИЕ 11.1

Предположим, что узел x в дереве поиска имеет потомков y и z. Какова связь между $size\,(x)$, $size\,(y)$ и $size\,(z)$?

а) $size\,(x) = \max\{size\,(y), size\,(z)\} + 1$.

б) $size\,(x) = size\,(y) + size\,(z)$.

[1] Эта идея также может быть использована для реализации операции Взять ранг со временем O(*высота*) (как вы сами можете убедиться).

в) $size(x) = size(y) + size(z) + 1$.

г) Общая связь отсутствует.

(Решение и пояснение см. в разделе 11.3.10.)

Чем полезна эта дополнительная информация? Представьте, что вы ищете объект с 17-м наименьшим ключом (то есть $i = 17$) в дереве поиска со 100 объектами. Начиная с корня, вы можете за постоянное время вычислить размеры его левого и правого поддеревьев. По свойству дерева поиска каждый ключ в левом поддереве меньше, чем в корне и в правом поддереве. Если популяция левого поддерева равна 25, то этими ключами являются 25 наименьших ключей в дереве, включая 17-й наименьший ключ. Если его популяция составляет только 12, то правое поддерево содержит все ключи, кроме 13 наименьших ключей, и 17-й наименьший ключ является 4-м наименьшим среди его 87 ключей. В любом случае, для того чтобы найти расположение нужного объекта, мы можем вызывать операцию Выбрать рекурсивно.

ВЫБРАТЬ

1. Начать с корня, и пусть j равно размеру его левого поддерева. (Если он не имеет указателя на левого потомка, то $j = 0$.)
2. Если $i = j + 1$, то вернуть указатель на корень.
3. Если $i < j + 1$, то рекурсивно вычислить i-й наименьший ключ в левом поддереве.
4. Если $i > j + 1$, то рекурсивно вычислить $(i - j - 1)$-й наименьший ключ в правом поддереве[1].

Поскольку каждый узел дерева поиска хранит размер своего поддерева, каждый рекурсивный вызов выполняет только постоянный объем работы. Каждый рекурсивный вызов продвигается дальше вниз по дереву, поэтому суммарный объем работы равен $O(\textit{высота})$.

[1] Структура рекурсии может напомнить вам о наших алгоритмах отбора в главе 6 *первой части*, где корневой узел играет роль опорного элемента.

За все надо платить. Мы все еще должны расплатиться за заказанную музыку. Мы добавили в дерево поиска метаданные и их эксплуатировали, и каждая модифицирующая дерево операция должна поддерживать эту информацию в актуальном состоянии, а также следить за соблюдением свойства дерева поиска. Вы должны продумать, как по-новому реализовать операции Вставить и Удалить, по-прежнему работающие с временем $O(высота)$, таким образом, чтобы все размеры поддеревьев оставались точными[1].

11.3.10. Решение тестового задания 11.1

Правильный ответ: (в). Каждый узел в поддереве с корнем в узле x является либо узлом x, либо узлом в левом поддереве узла x, либо узлом в правом поддереве узла x. Следовательно, мы имеем

$$size(x) = \underbrace{size(y)}_{\text{узлы в левом поддереве}} + \underbrace{size(z)}_{\text{узлы в правом поддереве}} + \underbrace{1}_{x}.$$

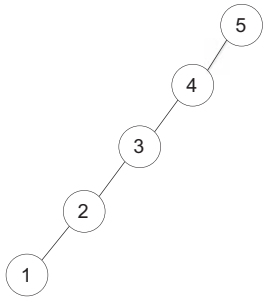
*11.4. Сбалансированные деревья поиска

11.4.1. Более напряженные усилия для улучшения баланса

Время выполнения каждой операции бинарного дерева поиска (за исключением операции Вывести в отсортированном порядке) является пропорциональным высоте дерева, которая может варьироваться от сценария в лучшем случае $\approx \log_2 n$ (для идеально сбалансированного дерева) до сценария в худшем случае $n - 1$ (для цепочки), где n — это число объектов в дереве. Плохо сбалансированные деревья поиска могут действительно возникать, например, когда объекты вставляются в отсортированном или обратно отсортированном порядке:

---

[1] Например, для операции Вставить увеличить размер поддерева для каждого узла на пути между корнем и только что вставленным объектом.



Разница между логарифмическим и линейным временем работы огромна, поэтому не грех потратить чуть больше усилий в операциях Вставить и Удалить — по-прежнему со временем $O(\textit{высота})$, но с более крупным постоянным множителем, — для того чтобы высота дерева гарантированно всегда равнялась $O(\log n)$.

Несколько разных типов сбалансированных деревьев поиска гарантируют высоту $O(\log n)$, а следовательно, достигают времени выполнения операций, заявленных в перечне показателей из табл. 11.2[1]. Дьявол кроется в мелочах реализации, и для сбалансированных деревьев поиска они могут становиться довольно хитроумными. К счастью, эти реализации легкодоступны, и маловероятно, что вам когда-нибудь понадобится кодировать свою собственную версию с нуля. Я призываю читателей, заинтересованных в том, что именно находится под капотом сбалансированного дерева поиска, обратиться к изложению данного материала в учебнике либо разведать реализации с открытым исходным кодом и визуальные демонстрации, которые находятся в свободном доступе в интернете[2]. Для того чтобы разжечь ваш интерес к дальнейшему изучению, давайте завершим данную главу одной из самых распространенных идей в сбалансированных реализациях дерева поиска.

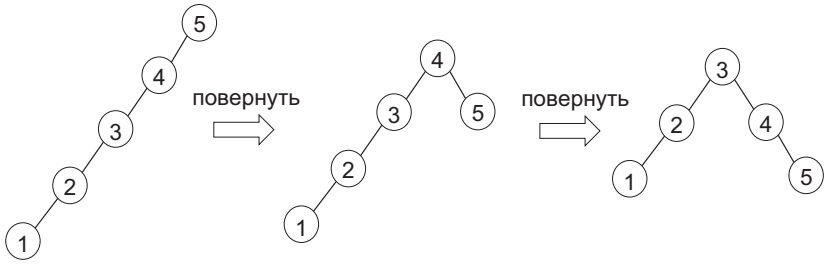
---

[1] Популярные реализации включают красно-черные деревья, 2–3-деревья, АВЛ-деревья, косые (splay) деревья и B-деревья и B+-деревья.

[2] Стандартное учебное изложение данной темы содержится в главе 13 книги *Томаса Х. Кормена, Чарльза Э. Лейзерсона, Рональда Л. Ривеста и Клиффорда Стейна «Introduction to Algorithms»* («Введение в алгоритмы»), третье издание (*MIT Press, 2009*), и разделе 3.3 книги *Роберта Седжвика и Кевина Уэйна «Algorithms»* («Алгоритмы»), четвертое издание (*Addison-Wesley, 2011*). См. также бонусные видео на www.algorithmsilluminated.org относительно основ красно-черных деревьев.

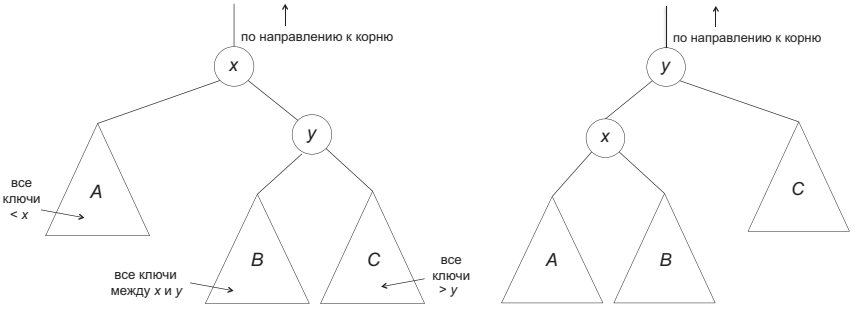
11.4.2. Повороты

Во всех наиболее распространенных реализациях сбалансированных деревьев поиска используются *повороты* — операции с постоянным временем, выполняющие небольшую локальную перебалансировку при сохранении свойства дерева поиска. Например, мы можем представить себе преобразование приведенной выше цепочки из пяти объектов в более цивилизованное дерево поиска, составив композицию из двух локальных операций перебалансировки:



Поворот берет пару «родитель — потомок» и разворачивает их связь в обратную сторону (рис. 11.2). *Правый поворот* применяется, когда потомок *y* является левым потомком своего родителя *x* (и значит, *y* имеет меньший ключ, чем *x*); после поворота узел *x* становится правым потомком узла *y*. Когда узел *y* является правым потомком узла *x*, *левый поворот* делает узел *x* левым потомком узла *y*.

Свойство дерева поиска обуславливает остальные детали. Рассмотрим левый поворот, где узел *y* является правым потомком узла *x*. Из свойства дерева поиска вытекает, что ключ узла *x* меньше, чем ключ узла *y*; что все ключи левого поддерева в узле *x* (*A* на рис. 11.2) меньше, чем ключ узла *x* (и узла *y*); что все ключи в правом поддереве узла *y* («*C*» на рис. 11.2) больше, чем ключ узла *y* (и узла *x*); и что все ключи в левом поддереве узла *y* (*B* на рис. 11.2) находятся между ключами *x* и *y*. После поворота узел *y* наследует старого родителя узла *x* и имеет узел *x* в качестве своего нового левого потомка. Существует уникальный способ собрать все части воедино, сохраняя при этом свойство дерева поиска, поэтому давайте просто будем следовать по прямой.

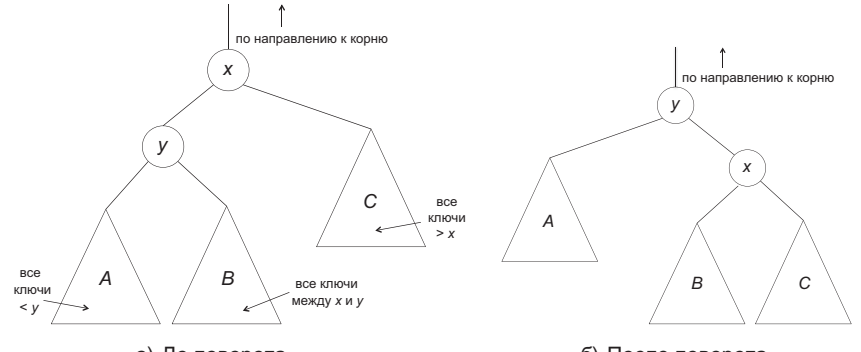


а) До поворота *б*) После поворота

Рис. 11.2. Левый поворот в действии

Существует три свободные ячейки для поддеревьев A, B и C: указатель на правого потомка узла y и оба указателя на потомков узла x. Свойство дерева поиска заставляет нас прикрепить наименьшее поддерево (A) в качестве левого потомка узла x и наибольшее поддерево (C) — в качестве правого потомка узла y. В результате остается одна ячейка для поддерева B (указателя на правого потомка узла x), и, к счастью, срабатывает свойство дерева поиска: все ключи поддерева вклиниваются между ключами x и y, и поддерево завершается с левым поддеревом узла y (где оно и должно быть) и правым поддеревом узла x (то же самое).

Правый поворот тогда является левым поворотом, только в обратную сторону (рис. 11.3).



а) До поворота *б*) После поворота

Рис. 11.3. Правый поворот в действии

Поскольку поворот просто-напросто заменит несколько указателей, он может выполняться с постоянным числом операций. По конструкции он сохраняет свойство дерева поиска.

Операции, которые модифицируют дерево поиска — Вставить и Удалить, — являются именно теми операциями, в которых должны задействоваться повороты. Без них такая операция могла бы придать дереву чуть более несбалансированную форму. Поскольку одна вставка или удаление может сеять хаос только в такой степени, вполне вероятно, что небольшое постоянное или, возможно, логарифмическое число поворотов может исправить любой вновь созданный дисбаланс. Именно это и делают приведенные выше реализации сбалансированного дерева поиска. Дополнительная работа за счет поворотов прибавляет $O(\log n)$ накладных расходов к операциям Вставить и Удалить, оставляя их совокупное время работы на уровне $O(\log n)$.

ВЫВОДЫ

- Если вашему приложению требуется поддержка полностью упорядоченного представления эволюционирующего множества объектов, то сбалансированное дерево поиска обычно является предпочтительной структурой данных.
- Сбалансированные деревья поиска поддерживают операции Отыскать, Минимум, Максимум, Предшественник, Преемник, Выбрать, Взять ранг, Вставить и Удалить со временем $O(\log n)$, где n — это число объектов.
- Бинарное дерево поиска имеет по одному узлу на объект, каждый из которых содержит указатель на родителя, указатель на левого потомка и указатель на правого потомка.
- Свойство дерева поиска констатирует, что в каждом узле x дерева ключи в левом поддереве узла x меньше, чем ключ узла x, и ключи в правом поддереве узла x больше, чем ключ узла x.
- *Высота* дерева поиска является длиной самого длинного пути от корня до листа. Бинарное дерево поиска с n объектами может иметь высоту от $\approx \log_2 n$ до $n - 1$.

- ✶ В базовом бинарном дереве поиска все поддерживаемые выше операции могут быть реализованы с временем $O(\textit{высота})$. (Операции Отыскать и Взять ранг — только после усиления дерева для поддержания размеров поддеревьев на каждом узле.)
- ✶ Сбалансированные бинарные деревья поиска выполняют дополнительную работу в операциях Вставить и Удалить — по-прежнему со временем $O(\textit{высота})$, но с более крупным постоянным множителем, — для того чтобы гарантировать, что высота дерева всегда равна $O(\log n)$.

Задачи на закрепление материала

Задача 11.1. Какие из следующих утверждений являются истинными? (Отметьте все подходящие варианты.)

а) Высота бинарного дерева поиска с n узлами не может быть меньше $\Theta(\log n)$.

б) Все операции, поддерживаемые бинарным деревом поиска (кроме операции Вывести в отсортированном порядке), выполняются за время $O(\log n)$.

в) Свойство кучи является частным случаем свойства дерева поиска.

г) Сбалансированные бинарные деревья поиска всегда предпочтительнее отсортированных массивов.

Задача 11.2. Дано бинарное дерево с n вершинами (через указатель на его корень). Каждый узел дерева имеет поле с размером *size*, как в разделе 11.3.9, но эти поля еще не заполнены. Сколько времени необходимо и достаточно для вычисления правильного значения для всех полей *size*?

а) $\Theta(\textit{высота})$

б) $\Theta(n)$

в) $\Theta(n \log n)$

г) $\Theta(n^2)$

Задача по программированию

Задача 11.3. Эта задача использует задачу о поддержке медианы из раздела 10.3.3 для разведывания относительной результативности куч и деревьев поиска.

а) Реализуйте на вашем любимом языке программирования решение с использованием кучи из раздела 10.3.3 задачи о поддержке медианы.

б) Реализуйте решение данной задачи, использующее единственное дерево поиска и его операции Вставить и Удалить.

Какая реализация быстрее?

Вы можете использовать существующие реализации куч и деревьев поиска либо реализовать собственные с нуля. (Обратитесь к сайту www.algorithmsilluminated.org для получения тестовых случаев и наборов данных для задач.)

12 Хеш-таблицы и фильтры Блума

Мы завершаем вторую часть книги невероятно полезной и распространенной структурой данных под названием хеш-таблица (или хеш-таблица соответствия, hash map). Хеш-таблицы, подобно кучам и деревьям поиска, поддерживают эволюционирующее множество объектов, ассоциированных с ключами (и, возможно, большим количеством других данных). В отличие от кучи и деревьев поиска, они не поддерживают никакой информации об упорядочении вообще. Смысл существования хеш-таблицы состоит в том, чтобы облегчить сверхбыстрый поиск, который также в этом контексте называется просмотром (lookup). Хеш-таблица может сказать вам, что в ней есть, а чего нет, и может сделать это очень и очень быстро (намного быстрее, чем куча или дерево поиска). Как обычно, мы начнем с поддерживаемых операций (раздел 12.1), и только потом перейдем к применениям (раздел 12.2) и некоторым дополнительным деталям реализации (разделы 12.3 и 12.4). Разделы 12.5 и 12.6 охватывают фильтры Блума, близких родственников хеш-таблиц, которые используют меньше места за счет случайных ошибок.

12.1. Поддерживаемые операции

Смысл существования хеш-таблицы заключается в отслеживании эволюционирующего множества объектов с ключами, при этом поддерживая быстрый поиск (по ключу), благодаря чему легко проверить, что в ней есть и чего нет. Например, если ваша компания управляет сайтом электронной торговли, то вы можете использовать одну хеш-таблицу для отслеживания сотрудников (возможно, с именами в качестве ключей), другую для хранения прошлых транзакций (с идентификаторами транзакций в качестве ключей) и третью для запоминания посетителей вашего сайта (с IP-адресами в качестве ключей).

Концептуально хеш-таблицу можно рассматривать как массив. Единственно, чем хороши массивы, — это немедленным случайным к ним доступом. Нужно узнать, что находится в позиции номер 17 массива? Просто обратитесь к этой позиции напрямую, за постоянное время. Хотите изменить содержимое в позиции 23? Опять же легко, за постоянное время.

Предположим, вам нужна структура данных для запоминания телефонных номеров ваших друзей. Если повезет, все ваши друзья имеют необычно «прозаичных» родителей, которые назвали своих детей в честь положительных целых

чисел, скажем, от 1 до 10 000. В этом случае вы можете хранить телефонные номера в массиве длиной 10 000 (то есть не таком большом). Если вашего лучшего друга зовут 173, сохраните его номер телефона в позиции 173 массива. Для того чтобы забыть о своем бывшем друге 548, замените позицию 548 значением по умолчанию. Это решение на основе массива работает хорошо, даже если ваши друзья со временем меняются — пространственные требования скромны, а операции вставки, удаления и поиска выполняются за постоянное время.

Вероятно, у ваших друзей есть более интересные, но менее удобные имена, такие как Алиса, Боб, Кэрол и так далее. Можем ли мы по-прежнему использовать решение на основе массива? В принципе, вы можете поддерживать массив с записями, индексированными по каждому возможному имени, которое вы когда-либо видели (не более, скажем, 25 букв). Для того чтобы посмотреть номер телефона Алисы, то можно заглянуть в позиции «Алиса» массива (рис. 12.1).

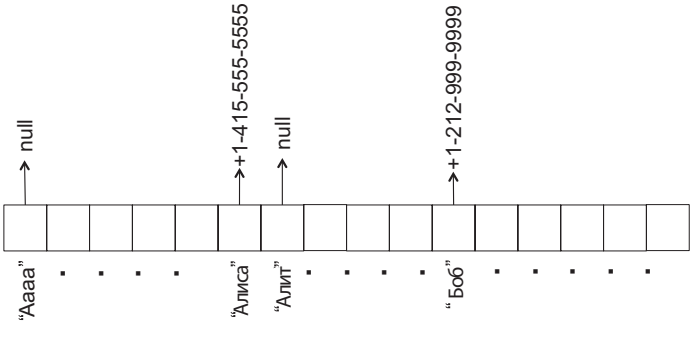


Рис. 12.1. В принципе, вы можете хранить телефонные номера ваших друзей в массиве, индексированном строковыми значениями длиной не более 25 символов

ТЕСТОВОЕ ЗАДАНИЕ 12.1

Сколько существует строковых значений длиной 25 символов? (Выберите самое сильное истинное утверждение.)

а) Больше, чем количество волос на голове.

б) Больше, чем число существующих веб-страниц.

в) Больше, чем общий объем хранилища, доступного на Земле (в битах).

г) Больше, чем число атомов во Вселенной.

(Решение и пояснение см. в разделе 12.1.1.)

Суть тестового задания 12.1 в том, что массив, необходимый для этого решения, СЛИШКОМ ВЕЛИК. Существует ли альтернативная структура данных, которая повторяет всю функциональность массива с постоянно-временными операциями вставки, удаления и поиска, а также использует пространство, пропорциональное числу хранимых объектов? Хеш-таблица является именно такой структурой данных.

ХЕШ-ТАБЛИЦЫ: ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ

Просмотреть (lookup, также именуемая Отыскать): по ключу k вернуть указатель на объект в хеш-таблице с ключом k (либо сообщить, что такого объекта не существует).

Вставить: с учетом нового объекта x добавить x в хеш-таблицу.

Удалить: по ключу k удалить объект с ключом k из хеш-таблицы, если он существует.

В хеш-таблице все эти операции обычно выполняются с постоянным временем, соответствующим наивному решению на основе массива, в рамках нескольких допущений, которые обычно соблюдаются на практике (описанных в разделе 12.3). Хеш-таблица использует пространство, линейное по числу хранимых объектов. Это радикально меньше, чем пространство, необходимое для наивного решения на основе массива, пропорциональное числу всех мыслимых объектов, которые могут когда-либо понадобиться для хранения. Соответствующий перечень показателей выглядит следующим образом.

Таблица 12.1. Хеш-таблицы: поддерживаемые операции и их типичное время выполнения. Звездочка (*) обозначает, что граница времени выполнения соблюдается тогда и только тогда, когда хеш-таблица реализована должным образом (с хорошей хеш-функцией и соответствующим размером таблицы) и данные не являются патологическими; см. раздел 12.3 по поводу подробностей

Операция	Типичное время выполнения
Просмотреть	$O(1)$*
Вставить	$O(1)$
Удалить	$O(1)$*

Подводя итог, хеш-таблицы не поддерживают много операций, но то, что они делают, им удается очень и очень хорошо. Всякий раз, когда операции просмотра составляют значительный объем работы вашей программы, у вас в мозгу должно щелкнуть: программе требуется хеш-таблица!

КОГДА ИСПОЛЬЗОВАТЬ ХЕШ-ТАБЛИЦУ

Если приложению требуется быстрый поиск с динамически изменяющимся множеством объектов, то хеш-таблица обычно является предпочтительной структурой данных.

12.1.1. Решение тестового задания 12.1

Правильный ответ: (в). Смысл этого тестового задания состоит в том, чтобы поразвлечься, думая о некоторых действительно больших числах, а не с целью определения правильного ответа как такового. Допустим, что существует 26 вариантов символов — игнорируя знаки препинания, символы верхнего и нижнего регистра и так далее. Тогда существует 26^{25} 25-буквенных строковых значений, которые имеют порядок примерно 10^{35}. (Есть также строковые значения с 24 буквами или меньше, но они незначительны по сравнению со строковыми значения длиной 25 символов.) Число волос на голове человека, как правило, составляет около 10^5. Проиндексированная Всемирная паутина имеет несколько миллиардов веб-страниц, но фактическое число веб-страниц, вероятно, составляет около одного триллиона (10^{12}). Общий объем памяти на

Земле в битах оценить трудно, но, по крайней мере в 2018 году, он наверняка не превышал йотабайт (10^{24} байт, или примерно 10^{25} бит). Между тем число атомов в известной Вселенной оценивается примерно в 10^{80}.

12.2. Применения

Просто поразительно, сколько разных приложений сводятся к повторным операциям просмотра и, следовательно, требуют наличия хеш-таблицы. Еще в 1950-х годах исследователи, создававшие первые компиляторы, нуждались в *таблице символов*, под которой подразумевалась подходящая структура данных для отслеживания имен программных переменных и функций. Хеш-таблицы были созданы именно для этого типа приложений. В качестве более современного примера представьте, что задачей сетевого маршрутизатора является блокировка пакетов данных с определенных IP-адресов, возможно, принадлежащих спамерам. Всякий раз, когда поступает новый пакет данных, маршрутизатор должен просматривать данные, проверяя, находится ли IP-адрес источника в черном списке. Если это так, то он отбрасывает пакет; в противном случае он переадресует пакет в пункт его назначения. Опять же, эти неоднократные просмотры данных находятся в ведении хеш-таблиц.

12.2.1. Применение: устранение дублирования

Устранение дублирования является каноническим применением хеш-таблиц. Предположим, вы обрабатываете огромный объем данных, поступающих потоком по одному фрагменту за раз. Допустим, что:

- Вы делаете один проход по огромному файлу, хранящемуся на диске, к примеру, со всеми транзакциями крупной розничной компании за последний год.
- Вы сканируете веб-страницы и обрабатываете миллиарды веб-страниц.
- Вы отслеживаете пакеты данных, проходящие через сетевой маршрутизатор с высокой скоростью.
- Вы наблюдаете за посетителями вашего веб-сайта.

В задаче устранения дублирования ваша ответственность заключается в том, чтобы игнорировать дубликаты и отслеживать только несовпадающие до этого ключи. Например, вас может интересовать число несовпадающих IP-адресов, которые обращались к вашему веб-сайту, в дополнение к суммарному числу посещений. Хеш-таблицы обеспечивают простое решение задачи устранения дублирования.

УСТРАНЕНИЕ ДУБЛИРОВАНИЯ С ХЕШ-ТАБЛИЦЕЙ

При поступлении нового объекта x с ключом k:

1. Применить операцию Просмотреть, для того чтобы проверить, содержит ли хеш-таблица объект с ключом k.
2. Если нет, то применить операцию Вставить, для того чтобы поместить x в хеш-таблицу.

После обработки данных хеш-таблица содержит ровно один объект в расчете на ключ, представленный в потоке данных[1].

12.2.2. Применение: задача о сумме двух чисел

Наш следующий пример более академичен, но он иллюстрирует, как повторные просмотры могут появляться в неожиданных местах. Пример касается задачи о сумме двух чисел (2-SUM)[2].

ЗАДАЧА: СУММА ДВУХ ЧИСЕЛ

Вход: несортированный массив A из n целых чисел и целевое целое число t.

[1] В большинстве реализаций хеш-таблиц можно выполнять итеративный обход хранимых объектов в некоем произвольном порядке за линейное время. Это позволяет продолжить обработку объектов после удаления дубликатов.

[2] См. описание задачи о сумме двух чисел (2-SUM) по ссылке https://coderbyte.com/algorithm/two-sum-problem. — *Примеч. пер.*

Цель: определить, существуют ли два числа x, y в A, удовлетворяющие $x + y = t$[1].

Задача о сумме двух чисел может быть решена путем поиска методом полного перебора (или «грубой силы») — методом последовательного перебора всех возможностей для x и y и проверки, работает ли какая-либо из них. Поскольку существует n вариантов для каждого из x и y, этот алгоритм является квадратично-временным ($\Theta(n^2)$).

Мы можем добиться лучшего. Первое ключевое наблюдение состоит в том, что для каждого варианта x может работать только один вариант для y (а именно, $t - x$). Так почему бы не поискать именно этот y?

СУММА ДВУХ ЧИСЕЛ (ПОПЫТКА № 1)

Вход: массив A из n целых чисел и целевое целое число t.

Выход: «да», если $A[i] + A[j] = t$ для некоторых $i, j \in \{1, 2, 3, ..., n\}$, «нет» — в противном случае.

```
for i = 1 to n do
  y := t − A[i]
  if A содержит y then     // линейный поиск
    return «да»
return «нет»
```

Это поможет? Цикл `for` имеет n итераций, а для поиска целого числа в несортированном массиве требуется линейное время, поэтому, похоже, это еще один квадратично-временной алгоритм. Но помните, сортировка является

[1] Существуют две немного разные версии этой задачи, в зависимости от того, должны ли x и y быть разными. Мы допустим $x = y$; другой случай похож (как вы сами можете убедиться).

бесплатным примитивом. Почему бы не задействовать ее, для того чтобы все операции поиска могли воспользоваться отсортированным массивом?

СУММА ДВУХ ЧИСЕЛ (РЕШЕНИЕ НА ОСНОВЕ ОТСОРТИРОВАННОГО МАССИВА)

Вход: массив A из n целых чисел и целевое целое число t.

Выход: «да», если $A[i] + A[j] = t$ для некоторых $i, j \in \{1, 2, 3, ..., n\}$, «нет» — в противном случае.

```
sort A     // используя подпрограмму сортировки
for i = 1 to n do
  y := t − A[i]
  if A содержит y then     // двоичный поиск
    return «да»
return «нет»
```

ТЕСТОВОЕ ЗАДАНИЕ 12.2

Каково время выполнения просвещенной реализации алгоритма на основе отсортированного массива для задачи о сумме двух чисел?

а) $\Theta(n)$

б) $\Theta(n \log n)$

в) $\Theta(n^{1,5})$

г) $\Theta(n^2)$

(Решение и пояснение см. в разделе 12.2.4.)

Решение задачи о сумме двух чисел на основе отсортированного массива значительно лучше по сравнению с поиском полным перебором, и оно де-

монстрирует элегантную силу алгоритмических инструментов из *первой части*. Но мы можем добиться еще большего. Окончательное понимание основывается на том, что этот алгоритм нуждался в отсортированном массиве только в той мере, в какой это необходимо для быстрого поиска. Поскольку бóльшая часть работы сводится к повторным просмотрам, у вас в мозгу должно щелкнуть: отсортированный массив является излишним, и то, к чему этот алгоритм действительно взывает, так это хеш-таблица!

СУММА ДВУХ ЧИСЕЛ (РЕШЕНИЕ НА ОСНОВЕ ХЕШ-ТАБЛИЦЫ)

Вход: массив A из n целых чисел и целевое целое число t.

Выход: «да», если $A[i] + A[j] = t$ для некоторых $i, j \in \{1, 2, 3, ..., n\}$, «нет» — в противном случае.

```
H := пустая хеш-таблица
for i = 1 to n do
    Вставить A[i] в H
for i = 1 to n do
    y := t − A[i]
    if H содержит y then    // используя операцию «Просмотреть»
        return «да»
return «нет»
```

Исходя из хорошей реализации хеш-таблицы и непатологических данных, операции Вставить и Просмотреть, как правило, выполняются с постоянным временем. В этом случае решение задачи о сумме двух чисел на основе хеш-таблиц выполняется в *линейное* время. Поскольку любой правильный алгоритм должен просматривать каждое число в A хотя бы один раз, это время выполнения является наилучшим из возможных (вплоть до постоянных множителей).

12.2.3. Применение: поиск в огромных пространствах состояний

Хеш-таблицы всецело предназначены для ускорения поиска. Одна из прикладных областей, в которой поиск встречается повсеместно, — это компьютерные игры, и, если говорить более обобщенно, задачи планирования. Подумайте, например, о шахматной программе, разведывающей последствия различных ходов. Их последовательность можно рассматривать как пути в огромном ориентированном графе, где вершины соответствуют состояниям игры (позициям всех фигур и тому, чей сейчас ход), а ребра — ходам (переходам из одного состояния в другое). Размер этого графа — астрономический (более 10^{100} вершин), поэтому нет никакой надежды записать его явно и применить любой из наших алгоритмов графового поиска из главы 8. Более сговорчивая альтернатива — выполнить алгоритм графового поиска, такой как поиск в ширину, начиная с текущего состояния, и разведать краткосрочные последствия разных ходов до достижения ограничения по времени. Для того чтобы узнать как можно больше, важно избегать разведывания вершины более одного раза, поэтому алгоритм поиска должен отслеживать уже посещенные им вершины. Как и в нашем приложении для устранения дублирования, эта задача практически создана для хеш-таблицы. Когда алгоритм поиска достигает вершины, он отыскивает ее в хеш-таблице. Если вершина уже существует, то алгоритм ее пропускает и возвращается назад; в противном случае он вставляет вершину в хеш-таблицу и продолжает ее разведывание[1, 2].

---

[1] В игровых приложениях самый популярный алгоритм графового поиска называется поиском *A** («*A* звезда»). Алгоритм поиска *A** является целеориентированным обобщением алгоритма Дейкстры (глава 9), который добавляет в дейкстрову оценку (уравнение 9.1) ребра (*v*, *w*) эвристическую оценку стоимости, необходимой для перемещения из *w* в «целевую вершину». Например, если вы вычисляете маршруты движения из заданного исходного пункта до заданного пункта назначения *t*, то эвристической оценкой может быть расстояние по прямой из *w* в *t*.

[2] Задумайтесь о современных технологиях и поразмышляйте, где еще используются хеш-таблицы. На то, чтобы у вас появилось несколько хороших догадок, не уйдет много времени!

12.2.4. Решение тестового задания 12.2

Правильный ответ: (б). Первый шаг может быть реализован за время $O(n \log n)$ с помощью алгоритма `MergeSort` (описанного в *первой части*) либо алгоритма `HeapSort` (раздел 10.3.1)[1]. Каждая из n итераций цикла может быть реализована со временем $O(\log n)$ посредством двоичного поиска. Сложив все вместе, получим конечную границу времени выполнения $O(n \log n)$.

*12.3. Реализация: высокоуровневая идея

В этом разделе рассматриваются наиболее важные высокоуровневые идеи в реализации хеш-таблиц: хеш-функции (которые отображают ключи в позиции в массиве), коллизии (разные ключи, которые отображаются в одну и ту же позицию) и наиболее распространенные стратегии разрешения коллизий. Раздел 12.4 предлагает более подробные рекомендации по реализации хеш-таблицы.

12.3.1. Два простых решения

Хеш-таблица хранит множество S ключей (и ассоциированных с ними данных), взятых из универсума U всех возможных ключей. Например, U может быть всеми 2^{32} возможными IPv4-адресами, всеми возможными строковыми значениями длиной не более 25 символов, всеми возможными состояниями шахматной доски и так далее. Множество S может быть IP-адресами, фактически посетившими веб-страницу в течение последних 24 часов, фактическими именами ваших друзей либо состояниями шахматной доски, которые ваша программа разведала за последние пять секунд. В большинстве применений хеш-таблиц размер универсума U является астрономическим, но размер подмножества S поддается управлению.

---

[1] Более быстрая реализация невозможна. По крайней мере, с помощью управляемого сравнением алгоритма сортировки (см. сноску 2 на с. 141 в главе 10).

Одним концептуально простым способом реализовать операции Просмотреть, Вставить и Удалить является отслеживание объектов в большом массиве, с одной записью в массиве для каждого возможного ключа в U. Если U является небольшим множеством наподобие всех трехсимвольных строковых значений (скажем, для того чтобы отслеживать аэропорты по их трехбуквенным кодам), то это решение на основе массива является хорошим, при этом все операции в нем выполняются за постоянное время. Во многих приложениях, в которых универсум U чрезвычайно велик, это решение будет абсурдным и невыполнимым; мы можем реалистично рассматривать только те структуры данных, которые требуют пространства, пропорционального $|S|$ (а не $|U|$).

Вторым простым решением является хранение объектов в связном списке. Хорошей новостью является то, что пространство, используемое этим решением, пропорционально $|S|$. Плохая новость заключается в том, что время выполнения операций Просмотреть и Удалить также масштабируется линейно вместе с $|S|$ — что гораздо хуже, чем постоянно-временные операции, поддерживаемые решением на основе массива. Смысл хеш-таблицы состоит в том, чтобы достичь наилучшего из обоих миров — пространства, пропорционального $|S|$, и постоянно-временных операций (табл. 12.2).

Таблица 12.2. Хеш-таблицы сочетают в себе лучшие возможности массивов и связных списков с пространством, линейным по числу хранимых объектов, и постоянно-временными операциями. Звездочка (*) указывает на то, что граница времени выполнения соблюдается тогда и только тогда, когда хеш-таблица реализована правильно и данные не являются патологическими

Структура данных	Пространство	Типичное время выполнения операции Просмотреть
Массив	$\Theta\|U\|$	$\Theta(1)$
Связный список	$\Theta\|S\|$	$\Theta\|S\|$
Хеш-таблица	$\Theta\|S\|$	$\Theta(1)$*

12.3.2. Хеш-функции

Для достижения наилучших результатов в обоих мирах хеш-таблица имитирует простое решение на основе массива, но с длиной массива n, про-

порциональной $|S|$, а не $|U|$[1]. На данный момент вы можете думать об n как примерно равном $2|S|$.

Хеш-функция выполняет трансляцию из того, что нас действительно волнует — имен наших друзей, состояний шахматной доски и так далее — в позиции в хеш-таблице. Формально хеш-функция — это отображение из множества U всех возможных ключей в множество позиций в массиве (рис. 12.2). Позиции в хеш-таблице обычно нумеруются с 0, поэтому множество позиций в массиве равно $\{0, 1, 2, ..., n - 1\}$.

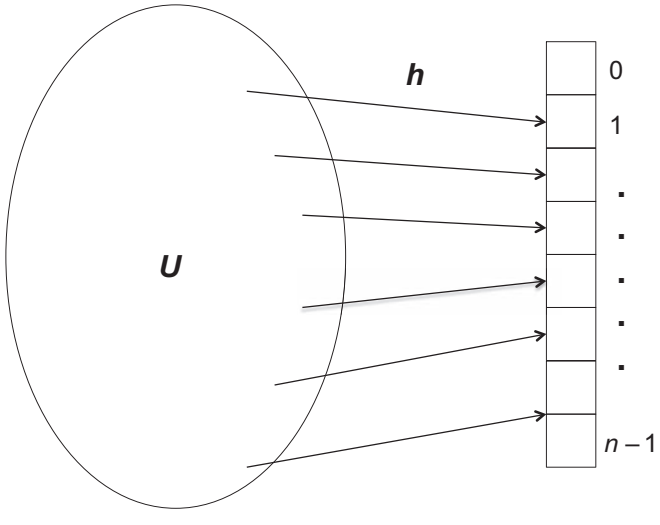


Рис. 12.2. Хеш-функция отображает каждый возможный ключ из универсума U в позицию в $\{0, 1, 2, ..., n - 1\}$. При $|U| > n$ два разных ключа будут отображаться в одинаковую позицию

ХЕШ-ФУНКЦИИ

Хеш-функция $h : U \rightarrow \{0, 1, 2, ..., n - 1\}$ присваивает каждый ключ из универсума U позиции в массиве длины n.

[1] Подождите! Разве множество S не меняется со временем? Да, это так, но совсем нетрудно периодически изменять размер массива так, чтобы его длина оставалась пропорциональной текущему размеру S; см. также раздел 12.4.2.

Хеш-функция сообщает вам, с чего начинать поиск объекта. Если вы выбираете хеш-функцию h, где h («Алиса») = 17 — в каковом случае мы говорим, что строковое значение «Алиса» хешируется в 17, — то позиция 17 массива является местом, с которого следует начинать искать номер телефона Алисы. Аналогичным образом позиция 17 является первым местом, где следует попытаться вставить номер телефона Алисы в хеш-таблицу.

12.3.3. Коллизии неизбежны

Возможно, вы заметили серьезную проблему: а что, если два разных ключа (например, «Алиса» и «Боб») хешируются в одну и ту же позицию (например, 23)? Если вы ищете номер телефона Алисы, но находите номер Боба в позиции 23 массива, как вы узнаете, имеется ли номер Алисы в хеш-таблице? Если вы пытаетесь вставить номер телефона Алисы в позицию 23, но эта позиция уже занята, куда вы ее поместите?

Когда хеш-функция h отображает два разных ключа k_1 и k_2 в одну и ту же позицию (то есть когда $h(k_1) = h(k_2)$), такая ситуация называется *коллизией*, и говорят, что ключи конфликтуют.

КОЛЛИЗИИ

Два ключа k_1 и k_2 из U конфликтуют в рамках хеш-функции h, если $h(k_1) = h(k_2)$.

Коллизии вызывают путаницу относительно того, где объект находится в хеш-таблице, и мы хотели бы, насколько это возможно, свести их к минимуму. Почему нельзя разработать сверхумную хеш-функцию без каких-либо коллизий? Потому что *коллизии неизбежны*. Причиной тому является так называемый *принцип голубей и ящиков* (или принцип Дирихле), интуитивно очевидный факт, что для каждого натурального числа n, независимо от того, как вы засовываете $n + 1$ голубей в n ячеек, всегда будет существовать ячейка по крайней мере с двумя голубями. Таким образом, всякий раз, когда число n позиций массива (ячеек) меньше размера универсума U (голубей), каждая хеш-функция (закрепление голубей за ячейками) — независимо от

того, насколько умной она является — страдает минимум от одной коллизии (см. рис. 12.2). В большинстве применений хеш-таблиц, в том числе в разделе 12.2, $|U|$ намного больше, чем n.

Коллизии еще неизбежнее, чем предполагает аргумент принципа голубей и ящиков. Причиной тому *парадокс дня рождения*, предмет следующего тестового задания.

ТЕСТОВОЕ ЗАДАНИЕ 12.3

Рассмотрим n человек со случайными днями рождения, причем каждый из 366 дней года равновероятен. (Предположим, что все n человек родились в високосный год.) Насколько большим должно быть n, прежде чем будет существовать 50 %-ный шанс, что у двух человек их день рождения будет совпадать?

а) 23

б) 57

в) 184

г) 367

(Решение и пояснение см. в разделе 12.3.7.)

Какое отношение парадокс дня рождения имеет к хешированию? Представьте себе хеш-функцию, которая независимо, равномерно и случайным образом закрепляет за каждым ключом позиции в $\{0, 1, 2, ..., n - 1\}$. Эта хеш-функция не является практически жизнеспособной (см. тестовое задание 12.5), но такие случайные функции являются золотым стандартом, с которым мы сравниваем практические хеш-функции (см. раздел 12.3.6). Из парадокса дня рождения вытекает, что даже в случае золотого стандарта мы, вероятно, начнем встречать коллизии в хеш-таблице размера n, как только будут вставлены объекты в количестве произведения малой константы на $\sqrt{n}$. Например, при n = 10 000 вставка 200 объектов может вызвать по крайней мере одну коллизию — даже если не менее 98 % позиций массива совершенно не используется!

12.3.4. Разрешение коллизий: сцепление

Когда коллизии являются неизбежным фактом жизни, хеш-таблица нуждается в каком-то методе для их решения. В этом и следующем разделах описываются два доминирующих подхода: *раздельное сцепление* (или просто *сцепление*) и *открытая адресация*. Оба подхода приводят к реализациям, в которых вставки и просмотры обычно выполняются в постоянном времени при допущении, что размер хеш-таблицы и хеш-функция выбраны соответствующим образом, а данные не являются патологическими (см. табл. 12.1).

Корзины и списки

Сцепление легко реализовать и представить в уме. Ключевая идея состоит в том, чтобы для обработки ситуаций, когда многочисленные объекты отображаются в одну и ту же позицию в массиве (рис. 12.3), по умолчанию использовать решение на основе связного списка (раздел 12.3.1). При использовании сцепления позиции массива часто называются *корзинами*, так как каждая из них может содержать несколько объектов. Тогда операции Просмотреть, Вставить и Удалить сводятся к одному оцениванию хеш-функции (с целью определить правильную корзину) и соответствующей операции связного списка.

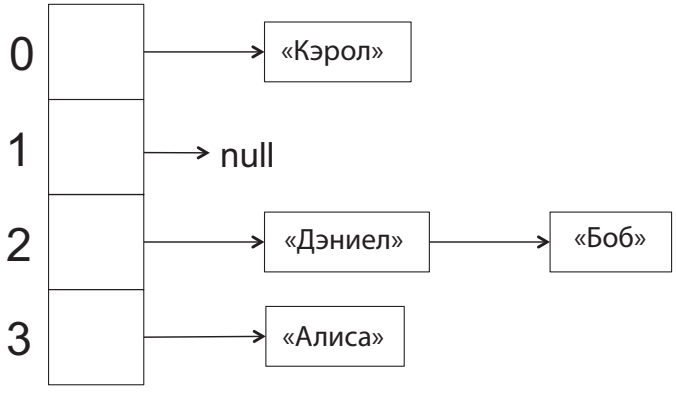


Рис. 12.3. Хеш-таблица с коллизиями, решаемая с помощью сцепления, с четырьмя корзинами и четырьмя объектами. Строковые значения «Боб» и «Дэниел» конфликтуют в третьей корзине (корзина 2). Показаны только ключи, а не ассоциированные с ними данные (например, номера телефонов)

СЦЕПЛЕНИЕ

1. Хранить связный список в каждой корзине хеш-таблицы.
2. Для того чтобы Просмотреть/Вставить/Удалить объект с ключом k, выполнить операции Просмотреть/Вставить/Удалить на связном списке в корзине $A[h(k)]$, где h обозначает хеш-функцию и A — массив хеш-таблицы.

Результативность сцепления

При условии, что h может быть оценена за постоянное время, операция Вставить также занимает постоянное время — новый объект может быть вставлен непосредственно в начало списка. Операции Просмотреть и Удалить должны выполнить поиск в списке, хранящемся в $A[h(k)]$, что занимает время, пропорциональное длине списка. Для достижения постоянно-временного поиска в хеш-таблице со сцеплением списки корзин должны оставаться короткими, в идеале — с длиной не более малой константы.

Длины (и время просмотров) списков уменьшаются, если хеш-таблица становится сильно заселенной. Например, если в хеш-таблице с длиной массива n хранятся $100n$ объектов, то типичная корзина имеет для просеивания 100 объектов. Время поиска также может ухудшиться из-за неудачно подобранной хеш-функции, которая приводит к большому числу коллизий. Например, в предельном случае, когда все объекты конфликтуют и оказываются в одной корзине, поиск может занять время, линейное по размеру совокупности данных. Раздел 12.4 подробно описывает, как управлять размером хеш-таблицы и выбирать соответствующую хеш-функцию для достижения границ времени выполнения, указанных в табл. 12.1.

12.3.5. Разрешение коллизий: открытая адресация

Вторым популярным методом разрешения коллизий является открытая адресация. Открытую адресацию намного проще реализовать и представить

в уме, когда хеш-таблица должна поддерживать только операции Вставить и Просмотреть (но не Удалить); мы сосредоточимся на этом случае[1].

В случае открытой адресации каждая позиция в массиве хранит не список, а 0 или 1 объект. (Для того чтобы это имело смысл, размер $|S|$ совокупности данных не может превышать размер n хеш-таблицы.) Коллизии создают прямое затруднение для операции Вставить: куда поместить объект с ключом k, если другой объект уже хранится в позиции $A[h(k)]$?

Зондажные последовательности

Идея состоит в том, чтобы связать каждый ключ k с *зондажной последовательностью* (probe sequence) позиций, а не только с одной позицией. Первое число последовательности указывает на позицию, которую нужно рассмотреть первой; второе — на следующую позицию, которую нужно рассмотреть, когда первая уже занята; и так далее. Объект хранится в первой незанятой позиции зондажной последовательности его ключа (рис. 12.4).

ОТКРЫТАЯ АДРЕСАЦИЯ

1. Вставить: с учетом объекта с ключом k пролистать зондажную последовательность, связанную с k, сохраняя объект в первой пустой найденной позиции.
2. Просмотреть: с учетом ключа k перелистывать зондажную последовательность, связанную с k, до тех пор пока не будет встречен нужный объект (в этом случае вернуть его) либо встречена пустая позиция (в этом случае сообщить «пусто», или «*null*»)[2].

[1] В целом ряде применений хеш-таблиц операция Удалить не требуется, включая три приложения в разделе 12.2.

[2] Если вы столкнулись с пустой позицией i, то вы можете быть уверены, что никакой объект с ключом k не находится в хеш-таблице. Такой объект хранился бы либо в позиции i, либо в более ранней позиции в зондажной последовательности ключа k.

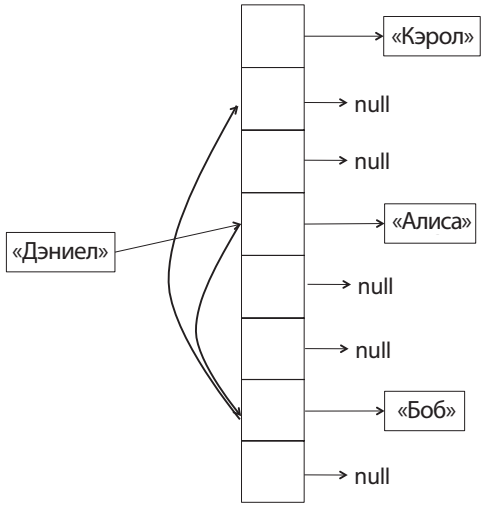


Рис. 12.4. Вставка в хеш-таблицу с коллизиями, решаемая с помощью открытой адресации. Первый элемент зондажной последовательности для ключа «Дэниел» конфликтует с ключом «Алиса», а второй — с ключом «Боб», но третий элемент является незанятой позицией

Линейное зондирование

Для определения зондажной последовательности существует несколько способов использования одной или нескольких хеш-функций. Самый простой представлен *линейным зондированием*. Этот метод использует одну хеш-функцию h и определяет последовательность зондирования для ключа k как $h(k)$, затем как $h(k) + 1$, затем как $h(k) + 2$ и так далее (циклически переходя в начало по достижении последней позиции). То есть данная хеш-функция указывает на стартовую позицию для вставки или просмотра, и операция сканирует вправо до тех пор, пока не найдет нужный объект или пустую позицию.

Двойное хеширование

Более сложный метод представлен *двойным хешированием*, использующим две хеш-функции[1]. Первая сообщает вам первую позицию зондажной после-

[1] Существует несколько «дешевых и сердитых» способов определить две хеш-функции из одной хеш-функции h. Например, если ключами являются неотрицательные целые числа, представленные в двоичной системе, то определить h_1 и h_2

довательности и вторая указывает на смещение для последующих позиций. Например, если $h_1(k) = 17$ и $h_2(k) = 23$, то первым местом, где следует искать объект с ключом k, является позиция 17; в случае неуспеха — позиция 40; в случае неуспеха — позиция 63; если это невозможно — позиция 86; и так далее. Для другого ключа k' зондажная последовательность может выглядеть совершенно иначе. Например, если $h_1(k') = 42$ и $h_2(k') = 27$, то зондажная последовательность будет 42, после которой идет 69, после которой идет 96, после которой идет 123 и так далее.

Результативность открытой адресации

При использовании сцепления время выполнения операции Просмотреть определяется длиной списков корзин; при использовании открытой адресации оно определяется типичным числом зондирований, необходимых для отыскания пустой ячейки либо искомого объекта. Понять результативность хеш-таблицы с открытой адресацией сложнее, чем со сцеплением, но интуитивно должно быть ясно, что результативность страдает, по мере того как хеш-таблица становится все более полной — если очень мало пустых ячеек, то зондажной последовательности обычно требуется продолжительное время на то, чтобы отыскать ячейки, — либо когда неудачно подобранная хеш-функция приводит к большому числу коллизий (см. также тестовое задание 12.4). С приемлемым размером хеш-таблицы и хеш-функцией открытая адресация достигает границ времени выполнения, указанных в табл. 12.1 для операций Вставить и Просмотреть; дополнительные сведения см. в разделе 12.4.

12.3.6. Что делает хеш-функцию хорошей?

Независимо от того, какую стратегию разрешения коллизий мы используем, результативность хеш-таблиц снижается вместе с числом конфликтов. Как выбрать хеш-функцию так, чтобы не было слишком много коллизий?

из h, прикрепляя новую цифру (0 либо 1) в конец заданного ключа k: $h_1(k) = h(2k)$ и $h_2(k) = h(2k + 1)$.

Плохие хеш-функции

Существует несметное число различных способов определения хеш-функции, и ее выбор имеет значение. Например, что происходит с результативностью хеш-таблицы при самом тупейшем выборе хеш-функции из возможных?

ТЕСТОВОЕ ЗАДАНИЕ 12.4

Рассмотрим хеш-таблицу длиной $n \geq 1$, и пусть h равно хеш-функции, где $h(k) = 0$, для каждого ключа $k \in U$. Предположим, что набор данных S вставлен в хеш-таблицу, где $|S| \leq n$. Каково типичное время выполнения последующих операций Просмотреть?

а) $\Theta(1)$ со сцеплением, $\Theta(1)$ с открытой адресацией.

б) $\Theta(1)$ со сцеплением, $\Theta(|S|)$ с открытой адресацией.

в) $\Theta(|S|)$ со сцеплением, $\Theta(1)$ с открытой адресацией.

г) $\Theta(|S|)$ со сцеплением, $\Theta(|S|)$ с открытой адресацией.

(Решение и пояснение см. в разделе 12.3.7.)

Патологические совокупности данных и «ахиллесова пята» хеш-функции

Никто из нас никогда не будет реализовывать дурацкую хеш-функцию из тестового задания 12.4. Вместо этого мы будем упорно работать для того, чтобы спроектировать умную хеш-функцию, которая гарантированно приводит к малому числу коллизий, а еще лучше отыскать подобную функцию в такой книге, как эта. К сожалению, я не могу показать вам подобную функцию. Чем я оправдаюсь? *Каждая хеш-функция, какой бы толковой она ни была, имеет свою «ахиллесову пяту»* в виде огромной совокупности данных, в случае которой все объекты конфликтуют, и результативность хеш-таблицы ухудшается, как в тестовом задании 12.4.

ПАТОЛОГИЧЕСКИЕ СОВОКУПНОСТИ ДАННЫХ

Для каждой хеш-функции h: $U \to \{0, 1, 2, ..., n-1\}$ существует множество S ключей размера $|U|/n$, таких что $h(k_1) = h(k_2)$ для каждого $k_1, k_2 \in S$[1].

Возможно, это покажется невероятным, но данное утверждение является всего лишь обобщением нашего аргумента на основе принципа голубей и ящиков из раздела 12.3.3. Зададим произвольно умную хеш-функцию h. Если h идеально разделяет ключи в U между n позициями, то каждая позиция будет иметь ровно $|U|/n$ назначенных ей ключей; в противном случае одной и той же позиции будут назначены еще больше, чем $|U|/n$ ключей. (Например, если $|U| = 200$ и $n = 25$, то h должна назначить не менее восьми разных ключей одной и той же позиции.) В любом случае, существует позиция $i \in \{0, 1, 2, ..., n-1\}$, которой h назначает не менее $|U|/n$ несовпадающих ключей. Если все ключи в совокупности данных S оказываются назначенными этой позиции i, то все объекты в совокупности данных конфликтуют.

Приведенный выше набор данных S является патологическим в том, что был сконструирован с единственной целью — нарушить выбранную хеш-функцию. Почему нас должен заботить такой искусственный набор данных? Основная причина заключается в том, что он объясняет звездочки в наших границах времени выполнения для хеш-табличных операций в табл. 12.1 и 12.2. В отличие от большинства алгоритмов и структур данных, которые мы встречали до сих пор, нет никакой надежды на гарантию времени выполнения, которая соблюдается абсолютно без каких-либо допущений о входных данных. Лучшее, на что мы можем надеяться, — это гарантия, которая применима ко всем непатологическим совокупностям данных, то есть наборам данных, определенным независимо от выбранной хеш-функции[2].

---

[1] В большинстве применений хеш-таблиц $|U|$ намного больше, чем n, и в этом случае набор данных размера $|U|/n$ является огромным!

[2] Можно также рассмотреть рандомизированные решения в духе алгоритма `QuickSort` в главе 5 *первой части*. Этот подход, называемый универсальным хешированием, гарантирует, что для каждой совокупности данных случайный выбор хеш-функции из малого класса функций обычно приводит всего к нескольким коллизиям. Дополнительные сведения и примеры см. в бонусных видео на www.algorithmsilluminated.org.

Хорошей новостью является то, что в случае с хорошо продуманной хеш-функцией обычно нет необходимости беспокоиться о патологических наборах данных на практике. Однако приложения, связанные с системами безопасности, представляют собой важное исключение из этого правила[1].

Случайные хеш-функции

Патологические наборы данных показывают, что ни одна хеш-функция не гарантирует наличие малого числа коллизий для каждого набора данных. Лучшее, на что мы можем надеяться, — это хеш-функция, которая имеет несколько коллизий для всех «непатологических» наборов данных[2].

Экстремальным подходом к декорреляции выбора хеш-функции и наборов данных является выбор случайной функции, то есть функции h, где для каждого ключа $k \in U$ значение $h(k)$ выбирается независимо и равномерно случайным образом из позиций массива $\{0, 1, 2, ..., n-1\}$. Функция h выбирается раз и навсегда при первоначальном создании хеш-таблицы. Интуитивно мы ожидаем, что такая случайная функция, как правило, распределяет объекты наборов данных S примерно равномерно по n позициям, при условии, что S определяется независимо от h. До тех пор пока n примерно равно $|S|$, она приводит к управляемому числу коллизий.

ТЕСТОВОЕ ЗАДАНИЕ 12.5

Почему нецелесообразно использовать совершенно случайный выбор хеш-функции? (Выберите все подходящие варианты.)

а) На самом деле это практично.

б) Выбор не является детерминированным.

---

[1] Интересное тематическое исследование описано в статье *Скотта А. Кросби и Дэна С. Уоллаха «Denial of Service via Algorithmic Complexity Attacks»* («Отказ в обслуживании посредством атак алгоритмической сложности») (*Proceedings of the 12th USENIX Security Symposium, 2003*). Кросби и Уоллах показали, как поставить на колени систему сетевого вторжения на основе хеш-таблиц с помощью умной конструкции патологической совокупности данных.

[2] Тупая хеш-функция в тестовом задании 12.4 приводит к ужасной результативности для каждой совокупности данных, патологической или иной.

в) Ее хранение займет слишком много места.

г) Ее оценивание займет слишком много времени.

(Решение и пояснение см. в разделе 12.3.7.)

Хорошие хеш-функции

Хорошая хеш-функция — это функция, которая пользуется преимуществами случайной функции, не страдая ни от одного из ее недостатков.

ЖЕЛАЕМАЯ ХЕШ-ФУНКЦИЯ

1. Дешевая при ее оценивании, в идеале за время $O(1)$.
2. Простая при ее сохранении, в идеале с пространством $O(1)$.
3. Имитирует случайную функцию, распределяя непатологические наборы данных приблизительно равномерно по позициям хеш-таблицы.

Как выглядит хорошая хеш-функция?

Хотя подробное описание современных хеш-функций выходит за рамки этой книги, возможно, вы изголодались в отношении чего-то более конкретного, чем описанные выше пожелания.

Например, рассмотрим ключи, которые являются целыми числами между 0 и некоторым крупным числом M[1]. Естественной первой попыткой создать хеш-функцию будет получение значения ключа по модулю числа n корзин:

$$h(k) = k \bmod n,$$

где $k \bmod n$ является результатом многократного вычитания n из k, до тех пор пока итогом не будет целое число между 0 и $n - 1$.

[1] Для того чтобы применить эту идею к нечисловым данным, таким как строковые значения, необходимо сначала преобразовать данные в целые числа. Например, в Java метод `hashCode` реализует такое преобразование.

Хорошей новостью является то, что эта функция дешева при ее оценивании и не требует хранения (кроме запоминания n)[1]. Плохая новость заключается в том, что многие реальные множества ключей неравномерно распределены в своих наименее значимых битах. Например, если $n = 1000$, и все ключи имеют одинаковые последние три цифры (основание 10) — возможно, оклады в компании, все из которых кратны 1000, или цены автомобилей, все из которых оканчиваются на «999», — тогда все ключи хешируются в одну и ту же позицию. Использование только самых значимых битов может вызвать аналогичные проблемы — подумайте, например, о международных и региональных кодах телефонных номеров.

Следующая идея состоит в том, чтобы зашифровать ключ перед применением операции взятия остатка после деления:

$$h(k) = (ak + b) \bmod n,$$

где a и b — это целые числа в $\{1, 2, ..., n - 1\}$. Эта функция опять же дешева при ее вычислении и проста при ее хранении (просто запомнить a, b и n). Для правильно выбранных a, b и n эта функция, вероятно, достаточно пригодна для использования в прототипе из серии «дешево и сердито». Однако же в случае критически важного исходного кода нередко необходимо использовать более сложные хеш-функции, которые обсуждаются далее в разделе 12.4.3.

В заключение упомянем две наиболее важные вещи, которые нужно знать о проектировании хеш-функции:

КЛЮЧЕВЫЕ МОМЕНТЫ

1. Эксперты изобрели хеш-функции, которые дешевы при их оценивании и просты в их хранении и которые ведут себя как случайные функции для всех практических целей.
2. Спроектировать такую хеш-функцию чрезвычайно трудно; вы должны оставить эту задачу экспертам, если это, конечно, возможно.

[1] Есть гораздо более быстрые способы вычислить $k \bmod n$, чем повторное вычитание!

12.3.7. Решения тестовых заданий 12.3–12.5

Решение тестового задания 12.3

Правильный ответ: (а). Хотите верьте, хотите нет, но вам потребуется всего 23 человека в комнате, прежде чем вероятность присутствия в ней двух человек с одинаковым днем рождения будет равна их отсутствию[1]. Вы можете сделать (или посмотреть) соответствующий расчет вероятности или убедиться в этом с помощью некоторых простых симуляций.

При наличии 367 людей будет существовать 100 %-ный шанс, что два человека имеют одинаковый день рождения (по принципу голубей и ящиков). Но с 57 людьми вероятность составляет уже примерно 99 %. А со 184? 99,99...% с большим числом девяток в периоде.

Большинство людей находят ответ противоречивым, вот почему этот пример называется парадоксом «дня рождения»[2]. В более общем случае, на планете с k днями в каждом году вероятность повторяющихся дней рождения достигает 50 % с $\Theta(\sqrt{k})$ людьми[3].

Решение тестового задания 12.4

Правильный ответ: (г). Если конфликты решаются с помощью сцепления, то хеш-функция h хеширует каждый объект в одинаковую корзину: корзину 0. Хеш-таблица превращается в простое решение на основе связных списков с временем $\Theta(|S|)$, необходимым для операции Просмотреть.

В случае открытой адресации будем считать, что хеш-таблица использует линейное зондирование. (Ситуация будет одинаковой для более сложных стра-

---

[1] Хороший трюк для незанудных вечеринок, скажем, с 35 участниками или около того.

[2] Название «парадокс» не совсем неправильное; здесь нет никакого логического противоречия, просто еще одна иллюстрация того, как мозг большинства людей не предназначен иметь хорошее интуитивное понимание вероятности.

[3] Причина состоит в том, что n человек обеспечивают не только n возможностей для повторяющихся дней рождения, но и $\binom{n}{2} \approx n^2/2$ разных возможностей (по одной для каждой пары людей). Два человека имеют один и тот же день рождения с вероятностью $1/k$, и вы ожидаемо начнете встречать коллизии, как только число возможностей коллизий будет примерно равно k (при $n = \Theta(\sqrt{k})$).

тегий, таких как двойное хеширование.) Счастливый первый объект $|S|$ будет закреплен за позицией 0 массива, следующий объект — за позицией 1 и так далее. Операция Просмотреть эволюционирует в линейный поиск по первым $|S|$ позициям неотсортированного массива, который требует времени $\Theta(|S|)$.

Решение тестового задания 12.5

Правильные ответы: (в), (г). Случайное отображение (функция) из U в $\{0, 1, 2, ..., n-1\}$ практически представляет собой просмотровую таблицу длины $|U|$ с $\log_2 n$ битами на элемент таблицы. При большом универсуме (как в большинстве применений) о записи такой функции либо ее оценивании не может быть и речи.

Мы могли бы попытаться определить хеш-функцию ситуативно по мере необходимости ее знать, закрепляя случайное значение за $h(k)$ при первом появлении ключа k. Но тогда оценивание $h(k)$ требует сначала проверить, был ли ключ уже определен. Это сводится к просмотру k в хеш-таблице, что как раз и является задачей, которую мы должны решить!

*12.4. Дополнительные детали реализации

Этот раздел предназначен для читателей, которые хотят реализовать свою собственную хеш-таблицу с нуля. В проектировании хеш-таблиц нет чудодейственного средства, поэтому я могу предложить только высокоуровневое руководство. Наиболее важными уроками являются: (i) управление загрузкой хеш-таблицы; (ii) использование хорошо протестированной современной хеш-функции; (iii) тестирование нескольких конкурирующих реализаций с целью определения лучшей из них для вашего конкретного приложения.

12.4.1. Загрузка против результативности

Результативность хеш-таблицы снижается по мере увеличения ее популяции: в случае сцепления списки корзин становятся длиннее; в случае открытой адресации становится сложнее найти пустую ячейку.

Загрузка хеш-таблицы

Мы измеряем популяцию хеш-таблицы посредством ее *загрузки*:

$$\text{загрузка хеш-таблицы} = \frac{\text{число хранимых объектов}}{\text{длина } n \text{ массива}}. \qquad (12.1)$$

Например, в хеш-таблице со сцеплением загрузка представляет собой среднюю популяцию в одной из корзин таблицы.

ТЕСТОВОЕ ЗАДАНИЕ 12.6

Какая хеш-табличная стратегия возможна для загрузок больше 1?

а) И сцепление, и открытая адресация.

б) Ни сцепление, ни открытая адресация.

в) Только сцепление.

г) Только открытая адресация.

(Решение и пояснение см. в разделе 12.4.5.)

Идеализированная результативность со сцеплением

В хеш-таблице со сцеплением время выполнения операции Просмотреть или Удалить масштабируется вместе с длиной списков корзин. В сценарии для лучшего случая хеш-функция распределяет объекты по корзинам равномерно. С загрузкой α этот идеализированный сценарий приводит к $\lceil \alpha \rceil$ объектам в расчете на корзину[1]. Тогда операции Просмотреть или Удалить занимают только $O(\lceil \alpha \rceil)$ времени, как и постоянно-временные операции, при условии что $\alpha = O(1)$[2]. Поскольку хорошие хеш-функции распределяют большинство

[1] Форма записи $\lceil x \rceil$ обозначает функцию «ceiling», которая округляет свой аргумент вверх до ближайшего целого числа.

[2] Мы потрудились написать $O(\lceil \alpha \rceil)$ вместо $O(\alpha)$ только для обработки случая, когда α находится близко к 0. Время выполнения каждой операции всегда равно $\Omega(1)$, независимо от того, насколько мала α — если нет ничего другого, существует одно оценивание хеш-функции, которое должно учитываться. В качестве альтернативы вместо $O(\lceil \alpha \rceil)$ мы могли бы написать $O(1 + \alpha)$.

наборов данных по корзинам примерно равномерно, эта результативность для лучшего случая приближенно подкрепляется практическими реализациями хеш-таблиц на основе сцепления (с хорошей хеш-функцией и непатологическими данными)[1].

Идеализированная результативность с открытой адресацией

В хеш-таблице с открытой адресацией время выполнения операции Просмотреть или Вставить масштабируется вместе с числом зондирований, необходимых для отыскания пустой ячейки (либо искомого объекта). Когда загрузка хеш-таблицы равна α, часть α ее ячеек заполнена, а оставшаяся часть $1 - \alpha$ является пустой. В сценарии для лучшего случая каждое зондирование не взаимодействует с содержимым хеш-таблицы и имеет $(1 - \alpha)$-й шанс найти пустую ячейку. В этом идеализированном сценарии ожидаемое число требуемых зондирований равно $1/(1 - \alpha)$[2]. Если α отделено от 1 — к примеру, 70 процентами, — идеализированное время выполнения всех операций составит $O(1)$. Эта оптимальная результативность приближенно подкрепляется практическими хеш-таблицам, реализованными с двойным хешированием или

---

[1] Приведем математическую аргументацию для читателей, которые помнят основные положения из теории вероятностей. Хорошая хеш-функция имитирует случайную функцию, поэтому пойдем далее и предположим, что хеш-функция h независимо закрепляет каждый ключ за одной из n корзин равномерно случайным образом. (См. раздел 12.6.1 с дальнейшим обсуждением этого эвристического допущения.) Предположим, что ключи всех объектов не совпадают и что ключ k отображается в позицию i через h. В рамках нашего допущения для каждого второго ключа k', представленного в хеш-таблице, вероятность того, что h также отобразит k' в позицию i, равна $1/n$. В общей сложности свыше $|S|$ ключей в совокупности данных S, то есть ожидаемое число ключей, которые делят между собой корзину k, равно $|S|/n$, то есть числу, которое также называется загрузкой α. (Технически это следует из линейности математического ожидания и «схемы декомпозиции», описанной в разделе 5.5 *первой части*.) Поэтому ожидаемое время выполнения операции Просмотреть объект с ключом k равно $O(\lceil \alpha \rceil)$.

[2] Это похоже на эксперимент с подбрасыванием монеты: если монета имеет вероятность p приземлиться «орлом», то каково среднее число подбрасываний, необходимых для того, чтобы увидеть ваших первых «орлов»? (Для нас $p = 1 - \alpha$.) Как описано в разделе 6.2 *первой части* — либо отыщите в «Википедии» «геометрическую случайную величину» — ответ равен $1/p$.

другими изощренными зондажными последовательностями. В случае линейного зондирования объекты имеют тенденцию собираться в расположенные подряд ячейки, что приводит к замедлению времени работы: грубо говоря, $1/(1-\alpha)^2$, даже в идеализированном случае[1]. Это по-прежнему составляет время $O(1)$ при условии, что α значительно меньше 100 %.

Таблица 12.3. Идеализированная результативность хеш-таблицы как функция от ее загрузки α и ее стратегии разрешения коллизий[2]

Стратегия разрешения коллизий	**Идеализированное время выполнения операции** Просмотреть
Сцепление	$O(\lceil \alpha \rceil)$
Двойное хеширование	$O(1/1-\alpha)$
Линейное зондирование	$O(1/(1-\alpha)^2)$

12.4.2. Управление загрузкой вашей хеш-таблицы

Вставки и удаления изменяют числитель в (12.1), и для того чтобы идти в ногу, реализация хеш-таблицы должна обновлять знаменатель. Хорошим эмпирическим правилом является периодическое изменение размера хеш-табличного массива с целью, чтобы загрузка таблицы оставалась ниже 70 % (или, возможно, даже меньше, в зависимости от применения и стратегии разрешения коллизий). Тогда, при наличии правильно выбранной хеш-функции и непатологических данных, все наиболее распространенные стратегии разрешения коллизий, как правило, приводят к постоянно-временным хеш-табличным операциям.

---

[1] Этот весьма неочевидный результат был впервые получен Дональдом Э. Кнутом, родоначальником анализа алгоритмов. Это произвело на него большое впечатление: «Я впервые сформулировал следующий вывод в 1962 году. С тех пор анализ алгоритмов стал одной из главных тем в моей жизни» (Дональд Э. Кнут. «Искусство программирования», т. 3 (2-е изд.), Аддисон-Уэсли, 1998, с. 536).

[2] Дополнительные сведения о том, как результативность разных стратегий разрешения коллизий зависит от загрузки хеш-таблицы, см. в бонусных видеороликах на веб-сайте www.algorithmsilluminated.org.

Самый простой способ реализовать изменение размера массива — отслеживать загрузку таблицы и при достижении ею 70 % удваивать число *n* корзин. Все объекты затем повторно хешируются в новую, более крупную хеш-таблицу (которая теперь имеет загрузку 35 %). При необходимости, если последовательность удалений значительно понижает загрузку, то массив может быть соответственно уменьшен в размере с целью сэкономить пространство (все остальные объекты повторно хешируются в меньшую таблицу). Такие изменения размера могут занимать много времени, но в большинстве приложений они происходят нечасто.

12.4.3. Выбор своей хеш-функции

Проектирование хороших хеш-функций представляет собой сложное и темное искусство. Очень легко предложить внешне разумно выглядящие хеш-функции, которые в итоге оказываются с небольшими изъянами, что приводит к низкой результативности хеш-таблицы. По этой причине я не советую создавать свои собственные хеш-функции с нуля. К счастью, ряд умных программистов разработали широкий спектр хорошо протестированных и общедоступных хеш-функций, которые вы можете использовать в своей работе.

К какой хеш-функции прибегнуть? Задайте этот вопрос десяти программистам, и вы получите минимум одиннадцать разных ответов. Поскольку разные хеш-функции оказываются в более благоприятном положении на разных распределениях данных, следует выполнять сопоставление результативности нескольких современных хеш-функций в своем конкретном приложении и среде выполнения. На момент написания этой книги (в 2018 году) хеш-функции, которые являются хорошей отправной точкой для дальнейшего изучения, включают `FarmHash`, `MurmurHash3`, `SpookyHash` и `MD5`. Все они являются *некриптографическими* хеш-функциями и не предназначены для защиты от враждебных атак, подобных атакам Кросби и Уоллаха (см. сноску 1 на с. 218)[1]. *Криптографические* хеш-функции сложнее и медленнее оцениваются, чем их некриптографические аналоги,

[1] Хеш-функция `MD5` изначально была разработана как криптографическая хеш-функция, но она больше не считается безопасной.

но зато они защищают от таких атак[1]. Хорошей отправной точкой является хеш-функция `SHA-1` и ее более современные родственники, такие как `SHA-256`.

12.4.4. Выбор стратегии разрешения коллизий

Что лучше использовать для разрешения коллизий: сцепление или открытую адресацию? Что лучше использовать в случае открытой адресации: линейное зондирование, двойное хеширование или что-то еще? Как обычно, когда я представляю вам несколько решений задачи, ответом будет «смотря по обстоятельствам». Например, сцепление занимает больше места, чем открытая адресация (для хранения указателей в связных списках), поэтому последняя может быть предпочтительнее, когда пространство является первоочередной проблемой. Реализация удалений выполняется сложнее с открытой адресацией, чем со сцеплением, поэтому второе может быть предпочтительнее в приложениях с большим числом удалений.

Сравнивать линейное зондирование с более сложными реализациями на основе открытой адресации, такими как двойное хеширование, также непросто. Линейное зондирование приводит к большим скоплениям расположенных подряд объектов в хеш-таблице и, следовательно, к большему числу зондирований, чем в более изощренных подходах; однако эта стоимость может быть компенсирована его дружественным взаимодействием с иерархией памяти в среде выполнения. Как и при выборе хеш-функции, в случае критически важного исходного кода ничто не может заменить программирование нескольких конкурирующих реализаций и наблюдение за тем, какое из них лучше всего подходит для вашего приложения.

---

[1] Все хеш-функции, даже криптографические, имеют патологические совокупности данных (раздел 12.3.6). Криптографические хеш-функции обладают особым свойством, которое состоит в том, что реконструировать патологический набор данных вычислительно невыполнимо, в том же смысле, что факторизация больших целых чисел и взлом криптосистемы с публичным ключом RSA вычислительно невыполнимы.

12.4.5. Решение тестового задания 12.6

Правильный ответ: (в). Поскольку хеш-таблицы с открытой адресацией хранят не более одного объекта в расчете на позицию в массиве, у них никогда не может быть загрузки больше 1. Как только загрузка стала равной 1, вставить больше объектов уже не получится.

В хеш-таблицу со сцеплением может быть вставлено произвольное число объектов, хотя ее результативность снижается по мере вставки большего числа объектов. Например, если загрузка равна 100, то средняя длина списка корзины тоже равна 100.

12.5. Фильтры Блума: основы

Фильтры Блума являются близкими родственниками хеш-таблиц[1]. Они очень компактны, но взамен периодически делают ошибки. В данном разделе дается описание того, чем хороши фильтры Блума и как они реализуются, в то время как раздел 12.6 излагает кривую компромисса между объемом используемого фильтром пространства и его частотой ошибок.

12.5.1. Поддерживаемые операции

Причина существования фильтров Блума по существу такая же, как и у хеш-таблицы: сверхбыстрые операции вставки и просмотра, благодаря которым вы можете быстро вспомнить, что вы видели, а что нет. Почему нас должна беспокоить другая структура данных с тем же множеством операций? Потому что фильтры Блума предпочтительнее хеш-таблиц в приложениях, в которых пространство ценится на вес золота, и случайная ошибка не является помехой для сделки.

---

[1] Названы в честь их изобретателя, см. работу *«Space/Time Trade-offs in Hash Coding with Allowable Errors»* («Компромиссы пространства/времени в хеш-кодировании с допустимыми ошибками») *Бертона Х. Блума* (*Communications of the ACM, 1970*).

Как и хеш-таблицы с открытой адресацией, фильтры Блума гораздо проще реализовать и представить в уме, когда они поддерживают только операции Вставить и Просмотреть (и без операции Удалить). Мы сосредоточимся на этом случае.

ФИЛЬТРЫ БЛУМА: ПОДДЕРЖИВАЕМЫЕ ОПЕРАЦИИ

Просмотреть: по ключу *k* вернуть «да», если *k* был ранее вставлен в фильтр Блума, и «нет» — в противном случае.

Вставить: добавить новый ключ *k* в фильтр Блума.

Фильтры Блума являются очень пространственно-эффективными; в типичном случае они могут потребовать всего 8 бит на вставку. Это довольно невероятно, так как 8 бит совершенно недостаточно, для того чтобы запомнить даже 32-битный ключ или указатель на объект! По этой причине операция Просмотреть в фильтре Блума возвращает только ответ «да» / «нет», тогда как в хеш-таблице данная операция возвращает указатель на искомый объект (если он найден). Именно поэтому операция Вставить теперь принимает только ключ, а не (указатель на) объект.

В отличие от всех других изученных нами структур данных фильтры Блума могут ошибаться. Существует два вида ошибок: ложные отрицания, когда операция Просмотреть возвращает «ложь», даже если запрашиваемый ключ был уже вставлен ранее, и ложные утверждения (или срабатывания), когда операция Просмотреть возвращает «истину», хотя запрашиваемый ключ еще не был вставлен в прошлом. В разделе 12.5.3 мы увидим, что базовые фильтры Блума никогда не страдают от ложных отрицаний, но они могут иметь «фантомные элементы» в виде ложных утверждений. В разделе 12.6 показано, что частотой ложных утверждений можно управлять, соответствующим образом настраивая использование пространства. Типичная реализация фильтра Блума может иметь частоту ошибок около 1 % либо 0,1 %.

Времена выполнения операций Вставить и Просмотреть являются такими же быстрыми, как и в хеш-таблице. И даже лучше, эти операции *гарантированно* выполняются в постоянном времени, независимо от реализации фильтра Блу-

ма и набора данных[1]. Однако реализация и набор данных влияют на частоту ошибок фильтра.

Подведем итоги преимуществ и недостатков фильтров Блума над хеш-таблицами:

ФИЛЬТР БЛУМА ПРОТИВ ХЕШ-ТАБЛИЦЫ

1. **Плюсы**: более пространственно-эффективный.
2. **Плюсы**: гарантированно постоянно-временные операции для каждой совокупности данных.
3. **Минусы**: не может хранить указатели на объекты.
4. **Минусы**: более сложные удаления в сравнении с хеш-таблицей со сцеплением.
5. **Минусы**: ненулевая вероятность ложного утверждения.

Перечень показателей для базовых фильтров Блума выглядит следующим образом.

Таблица 12.4. Базовые фильтры Блума: поддерживаемые операции и время их выполнения. Знак кинжала (†) указывает на то, что операция Просмотреть имеет управляемую, но ненулевую вероятность ложных утверждений

Операция	Время выполнения
Просмотреть	$O(1)^{\dagger}$
Вставить	$O(1)$

Фильтры Блума следует использовать вместо хеш-таблиц в приложениях, в которых имеют значение их преимущества, а их недостатки не являются помехой для сделки.

[1] При условии, что оценивания хеш-функции занимают постоянное время и что число бит является постоянным в расчете на вставленный ключ.

КОГДА ИСПОЛЬЗОВАТЬ ФИЛЬТР БЛУМА

Если приложению требуется быстрый поиск с динамически эволюционирующим множеством объектов, пространство ценится на вес золота и приемлемо малое число ложных утверждений, то фильтр Блума обычно является предпочтительной структурой данных.

12.5.2. Применения

Далее идут три применения с повторными просмотрами, где экономия пространства может быть важной, а ложные утверждения не являются помехой для сделки.

Спеллчекеры. Еще в 1970-х годах фильтры Блума использовались для реализации средств проверки орфографии — спеллчекеров. На этапе предобработки каждое слово в словаре вставляется в фильтр Блума. Орфографическая проверка документа сводится к одной операции Просмотреть на слово в документе, помечая любые слова, для которых данная операция возвращает «нет».

В этом приложении ложное утверждение соответствует недопустимому слову, которое спеллчекер непреднамеренно принимает. Такие ошибки не делали спеллчекеры идеальными. Однако в начале 1970-х годов пространство ценилось на вес золота, поэтому в то время использование фильтров Блума было беспроигрышной стратегией.

Запрещенные пароли. Старое приложение, которое остается актуальным до сего дня, отслеживает запрещенные пароли — пароли, которые слишком распространены либо их слишком легко угадать. Изначально все запрещенные пароли вставляются в фильтр Блума; дополнительные запрещенные пароли могут быть вставлены позже, по мере необходимости. Когда пользователь пытается установить либо сбросить свой пароль, то система ищет предложенный пароль в фильтре Блума. Если поиск возвращает «да», то пользователю предлагается повторить попытку с другим паролем. Здесь ложное утверждение транслируется в прочный пароль, который система отклоняет.

При условии, что частота ошибок не слишком велика, скажем не более 1 % либо 0,1 %, это не имеет большого значения. Время от времени некоторым пользователям потребуется одна дополнительная попытка найти пароль, приемлемый для системы.

Интернет-маршрутизаторы. Ряд сногсшибательных сегодняшних применений фильтров Блума имеют место в глуби интернета, где пакеты данных проходят через маршрутизаторы с потоковой скоростью. Существует много причин, почему маршрутизатор мог бы захотеть быстро вспомнить то, что он видел в прошлом. Например, маршрутизатор может захотеть найти истоковый IP-адрес пакета в списке заблокированных IP-адресов, отследить содержимое кэша, для того чтобы избежать паразитных просмотров кэша, или вести статистику, помогающую в идентификации сетевой атаки «отказ в обслуживании». Скорость поступления пакетов требует сверхбыстрых просмотров, а ограниченная память маршрутизатора делает пространство на вес золота. Эти применения находятся в прямом ведении фильтра Блума.

12.5.3. Реализация

Заглянув внутрь фильтра Блума, можно увидеть элегантную реализацию. Структура данных поддерживает n-битовую строку или, равным образом, массив A длины n, в котором каждый элемент равен 0 или 1. (Все элементы инициализируются нулем.) Данная структура также использует m хеш-функций $h_1, h_2, ..., h_m$, при этом каждая отображает универсум U всех возможных ключей в множество $\{0, 1, 2, ..., n - 1\}$ позиций в массиве. Параметр m пропорционален числу бит, используемых фильтром Блума для вставки, и, как правило, является малой константой (например, 5)[1].

[1] В разделах 12.3.6 и 12.4.3 содержатся указания по выбору одной хеш-функции. Сноска на с. 214 описывает дешевый и сердитый способ получения двух хеш-функций из одной; та же идея может быть использована для получения m хеш-функций из одной. Альтернативный подход, обусловленный двойным хешированием, заключается в использовании двух хеш-функций h и h' для определения $h_1, h_2, ..., h_m$ по формуле $h_i(k) = (h(k) + (i - 1) \times h'(k)) \bmod n$.

Всякий раз, когда ключ вставляется в фильтр Блума, каждая из m хеш-функций ставит флаг, устанавливая соответствующий бит массива A равным 1.

ФИЛЬТР БЛУМА: ВСТАВИТЬ (ПО КЛЮЧУ k)

for $i = 1$ to m **do**

$A[h_i(k)] := 1$

Например, если $m = 3$ и $h_1(k) = 23$, $h_2(k) = 17$ и $h_3(k) = 5$, вставка k приводит к тому, что 5-й, 17-й и 23-й биты массива устанавливаются равными 1 (рис. 12.5).

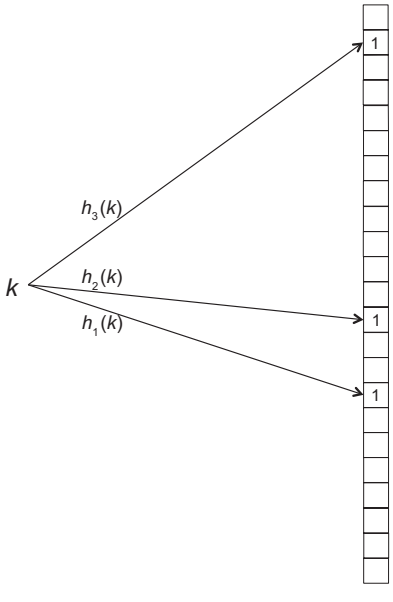


Рис. 12.5. Вставка нового ключа k в фильтр Блума устанавливает биты в позиции $h_1(k)$, ..., $h_m(k)$ равными 1

В операции Просмотреть фильтр Блума отыскивает отпечаток, который мог бы остаться при вставке k.

ФИЛЬТР БЛУМА: ПРОСМОТРЕТЬ (ПО КЛЮЧУ *k*)

for $i = 1$ to m **do**

  if $A[h_i(k)] = 0$ **then**

    return «нет»

return «да»

Теперь мы можем увидеть, почему фильтры Блума не могут страдать от ложных отрицаний. Когда ключ k вставляется, соответствующие m бит устанавливаются равными 1. За время жизни фильтра Блума биты могут менять свое значение с 0 на 1, но не наоборот. Тем самым эти m бит остаются 1 навсегда. Каждая последующая операция Просмотреть k гарантированно возвращает правильный ответ «да».

Мы также можем увидеть, как возникают ложные утверждения. Предположим, что $m = 3$ и четыре ключа k_1, k_2, k_3, k_4 имеют следующие хеш-значения:

Ключ	**Значение** h_1	**Значение** h_2	**Значение** h_3
k_1	23	17	5
k_2	5	48	12
k_3	37	8	17
k_4	32	23	2

Предположим, что мы вставляем k_1, k_2, k_3 и k_4 в фильтр Блума (рис. 12.6). Эти три вставки в общей сложности приводят к тому, что девять бит устанавливаются равными 1, включая три бита в отпечатке ключа k_1 (5, 17 и 23). На этом этапе фильтр Блума больше не может различать, был или нет вставлен ключ k_1. Даже если k_1 не был вставлен в фильтр, поиск вернет «да», что является ложным утверждением.

Интуитивно можно предположить, что с увеличением размера n фильтра Блума число наложений между отпечатками разных ключей должно уменьшаться, что, в свою очередь, приводит к меньшему числу ложных утверждений. Но

первоочередная цель фильтра Блума — экономить пространство. Существует ли золотая середина, где и n, и частота ложных утверждений одновременно малы? Ответ не очевиден и требует некоторого математического анализа, предпринятого в следующем далее разделе[1].

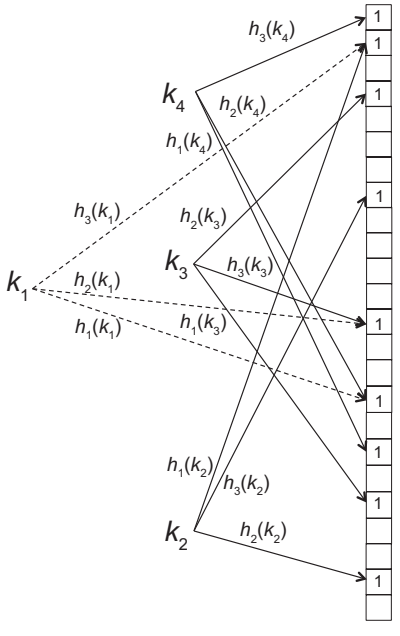


Рис. 12.6. Ложные утверждения: фильтр Блума может содержать отпечаток ключа k_1, даже если k_1 никогда не вставлялся

*12.6. Фильтр Блума: эвристический анализ

Цель этого раздела — понять количественный компромисс между потреблением пространства и частотой ложных утверждений фильтра Блума. Иными

[1] Внимание, спойлер! Ответ — да. Например, использование 8 бит в расчете на ключ обычно приводит к ложному утверждению с вероятностью примерно 2 % (при условии хорошо продуманных хеш-функций и непатологической совокупности данных).

словами, насколько быстро уменьшается частота ложных утверждений фильтра Блума как функция от длины его массива?

Если фильтр Блума использует битовый массив длиной n и хранит множество S ключей (отпечатки этого множества), то хранилище в битах в расчете на ключ составляет

$$b = \frac{n}{|S|}.$$

Нас интересует случай, когда b меньше, чем число бит, необходимых для явного хранения ключа или указателя на объект (который, как правило, ≥ 32). Например, b может быть равно 8 или 16.

12.6.1. Эвристические допущения

Связь между хранилищем ключей b и частотой ложных утверждений не так легко угадать, и ее выработка требует некоторых вероятностных расчетов. Для того чтобы их понять, вам нужно вспомнить из теории вероятностей только вот это:

> Вероятность возникновения двух независимых событий равна произведению их вероятностей.

Например, вероятность того, что два независимых броска правильного шестигранного кубика дадут «4», за которым последует нечетное число, равна $1/6 \times 3/6 = 1/12$[1].

Для упрощения вычисления мы сделаем два неоправданных допущения — те же самые, которые мы использовали при прохождении эвристического анализа результативности хеш-таблиц (раздел 12.4.1).

---

[1] Дополнительную общую информацию о теории вероятностей см. в приложении Б *первой части* либо в вики-справочнике по дискретной вероятности (https://en.wikibooks.org/wiki/High_School_Mathematics_Extensions/Discrete_Probability).

НЕОБОСНОВАННЫЕ ДОПУЩЕНИЯ

1. Для каждого ключа $k \in U$ в совокупности данных и хеш-функции h_i фильтра Блума $h_i(k)$ является равномерно распределенным, причем каждая из n позиций в массиве является равновероятной.
2. Все $h_i(k)$, охватывающие все ключи $k \in U$ и хеш-функции h_1, h_2, ..., h_m, являются независимыми случайными величинами.

Первое допущение говорит, что для каждого ключа k, каждой хеш-функции h_i и каждой позиции $q \in \{0, 1, 2, \ldots, n - 1\}$ в массиве вероятность того, что $h_i(k) = q$ равна в точности $1/n$. Из второго допущения вытекает, что вероятность того, что $h_i(k_1) = q$, а также $h_j(k_2) = r$, равна произведению индивидуальных вероятностей, также рассчитываемая как $1/n^2$.

Оба допущения были бы правомерны, если бы мы случайно выбирали каждую из хеш-функций фильтра Блума независимо от множества всех возможных хеш-функций, как в разделе 12.3.6. Полностью случайные хеш-функции нереализуемы (напомним тестовое задание 12.5), поэтому на практике используется фиксированная, «случайно-подобная» функция. Это означает, что в действительности *наши эвристические допущения являются ложными*. С фиксированными хеш-функциями каждое значение $h_i(k)$ является полностью детерминированным, без какой-либо случайности. Именно поэтому мы называем данный анализ эвристическим.

ОБ ЭВРИСТИЧЕСКОМ АНАЛИЗЕ

Какая польза от математического анализа, основанного на ложных предпосылках? В идеале вывод данного анализа остается валидным в практических ситуациях, даже если эвристические допущения не удовлетворены. Для фильтров Блума надежда заключается в том, что при условии, что данные не являются патологическими и используются добротные «случайно-подобные» хеш-функции, частота ложных утверждений ведет себя *так, как если бы* хеш-функции были полностью случайными.

Вы всегда должны с подозрением относиться к эвристическому анализу и обязательно проверять его выводы на конкретной реализации. К счастью, эмпирические исследования показывают, что частота ложных утверждений фильтров Блума на практике сопоставима с предсказанием нашего эвристического анализа.

12.6.2. Доля установленных бит (равных 1)

Начнем с предварительного расчета.

ТЕСТОВОЕ ЗАДАНИЕ 12.7

Предположим, что набор данных S вставляется в фильтр Блума, который использует m хеш-функций и битовый массив длины n. В рамках наших эвристических допущений какова вероятность того, что первый бит массива равен 1?

а) $(1/n)^{|S|}$

б) $(1 - 1/n)^{|S|}$

в) $(1 - 1/n)^{m|S|}$

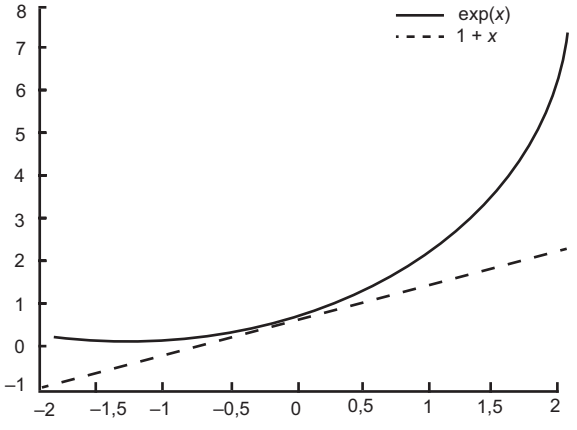
г) $1 - (1 - 1/n)^{m|S|}$

(Решение и пояснение см. в разделе 12.6.5.)

В первом бите фильтра Блума нет ничего особенного. По симметрии ответом на тестовое задание 12.7 также является вероятность того, что 7-й, или 23-й, или 42-й бит установлен равным 1.

12.6.3. Вероятность ложного утверждения

Решение тестового задания 12.7 является запутанным. Для того чтобы его привести в порядок, мы можем использовать тот факт, что e^x является хорошей аппроксимацией $1 + x$ при x близком к 0, где $e \approx 2{,}18...$ является основанием натурального логарифма. Этот факт очевиден из графика двух функций:



Для нас релевантным значением x является $x = -1/n$, которое является близким к 0 (игнорируя неинтересный случай крошечного n). Таким образом, мы можем использовать количество

$1 - (e^{-1/n})^{m|S|}$ в качестве замены для $1 - (1 - 1/n)^{m|S|}$.

Мы можем дальше упростить левую сторону до

$$1 - e^{-m|S|/n} = \underbrace{1 - e^{-m/b}}_{\text{оценка вероятности, что данный бит равен 1}},$$

где $b = n/|S|$ обозначает число бит в расчете на вставку.

Все это замечательно, но что насчет частоты ложных утверждений? Ложное утверждение случается для ключа k не в S, когда все m бит $h_1(k)$, ..., $h_m(k)$ в его отпечатке установлены равными 1 ключами в S[1]. Поскольку вероятность того, что данный бит равен 1, приближенно равна $1 - e^{-m/b}$, вероятность того, что все m этих бит установлены равными 1, приближенно равна

$$\underbrace{(1 - e^{-m/b})^m}_{\text{оценка частоты ложного утверждения}}. \qquad (12.2)$$

[1] В целях упрощения мы исходим из того, что каждая из m хеш-функций хеширует k в другую позицию (как это обычно бывает).

Мы можем проверить работоспособность этой оценки, разведав значения b. По мере того как фильтр Блума вырастает до сколь угодно больших значений (с $b \to \infty$) и становится все более пустым, данная оценка (12.2) стремится к 0, как мы и надеемся (потому что e^{-x} стремится к 1 по мере того, как x стремится к 0). И наоборот, при очень малом значении b оценка вероятности ложного утверждения является большой (к примеру, ≈ 63,2 % при $b = m = 1$)[1].

12.6.4. Кульминационный момент

Мы можем использовать нашу точную оценку (формула 12.2) частоты ложных утверждений, для того чтобы понять компромисс между пространством и точностью. В дополнение к пространству в расчете на ключ b оценка в (12.2) зависит от m, числа хеш-функций, которые используются фильтром Блума. Значение m находится под полным контролем проектировщика фильтра Блума, так почему бы не настроить его так, чтобы оно минимизировало оценочную частоту ошибок? То есть, оставляя b фиксированным, мы можем подобрать m для минимизации (12.2). С помощью дифференциального исчисления можно подобрать наилучшее m, приравняв производную (12.2) по m нулю и найдя решение для m. Вы можете выполнить эти вычисления в уединении домашней обстановки с тем результатом, что $(\ln 2) \times b \approx 0{,}693 \times b$ является оптимальным вариантом для m. Это не целое число, поэтому округлите его вверх или вниз, для того чтобы получить идеальное число хеш-функций. Например, при $b = 8$ число хеш-функций m должно быть либо 5, либо 6.

Теперь мы можем конкретизировать оценку ошибки в (12.2) с участием оптимального числа $m = (\ln 2) \times b$ и получить оценку

$$(1 - e^{-\ln 2})^{(\ln 2) \times b} = (1/2)^{(\ln 2) \times b}.$$

[1] В дополнение к нашим двум эвристическим допущениям этот анализ обманывал дважды. Во-первых, $e^{-1/n}$ не совсем равно $1 - 1/n$, но оно близко. Во-вторых, даже с нашими эвристическими допущениями значения двух разных бит фильтра Блума не являются независимыми — знание того, что один бит равен 1, делает чуть более вероятным то, что другой бит равен 0, — но они близки. Оба обмана являются близкими аппроксимациями реальности (с учетом эвристических допущений), а то, что они приводят к правильному заключению, можно перепроверить как математически, так и эмпирически.

Это именно то, чего мы хотели с самого начала. У нас есть формула, которая выдает ожидаемую частоту ложных утверждений как функцию объема пространства, которое мы готовы использовать[1]. Данная формула уменьшается *экспоненциально* вместе с пространством b в расчете на ключ, и именно поэтому существует золотая середина, где и размер фильтра Блума, и его частота ложных утверждений малы одновременно. Например, при хранении только 8 бит на ключ ($b = 8$) эта оценка составляет чуть более 2 %. Что, если взять $b = 16$ (см. задачу 12.3)?

12.6.5. Решение тестового задания 12.7

Правильный ответ: (г). Мы можем визуализировать вставку ключей из S в фильтр Блума как бросание дротиков в игре дартс в мишень с n секторами, причем каждый дротик попадает в каждый сектор равновероятно. Поскольку фильтр Блума использует m хеш-функций, каждая вставка соответствует метанию m дротиков, в общей сложности $m|S|$ дротиков в целом. Дротик, попавший в i-й сектор, соответствует установке i-го бита фильтра Блума равным 1.

По первому эвристическому допущению для каждых $k \in S$ и $i \in \{1, 2, ..., m\}$ вероятность того, что дротик попадет в первый сектор (то есть что $h_i(k) = 0$), равна $1/n$. Следовательно, дротик *промахивается* по первому сектору (то есть $h_i(k)$ *не* равно 0) с оставшейся вероятностью $1 - 1/n$. По второму эвристическому допущению разные дротики являются независимыми. Следовательно, вероятность того, что *каждый* дротик промахнется по первому сектору — то есть $h_i(k) \neq 0$ для *каждых* $k \in S$ и $i \in \{1, 2, ..., m\}$ — равна $(1 - 1/n)^{m|S|}$. С оставшейся $1 - (1 - 1/n)^{m|S|}$ вероятностью, что некий дротик попадает в первый сектор (то есть первый бит фильтра Блума установлен равным 1).

---

[1] Равным образом, если у вас есть целевая частота e ложных утверждений, то вы должны взять пространство в расчете на ключ равным не менее $b \approx 1{,}44 \log_2 1/e$. Как и ожидалось, чем меньше целевая частота e ошибок, тем больше потребности в пространстве.

ВЫВОДЫ

- Если вашему приложению требуются быстрые просмотры на эволюционирующем множестве объектов, то хеш-таблица обычно является предпочтительной структурой данных.
- Хеш-таблицы поддерживают операции Вставить и Просмотреть, и в некоторых случаях операцию Удалить. С хорошо реализованной хеш-таблицей и непатологическими данными все операции обычно выполняются за время $O(1)$.
- Хеш-таблица использует хеш-функцию для трансляции ключей объектов в позиции в массиве.
- Два ключа k_1, k_2 конфликтуют в рамках хеш-функции h, если $h(k_1) = h(k_2)$. Коллизии неизбежны, и хеш-таблице необходим метод их разрешения, такой как сцепление или открытая адресация.
- Хорошая хеш-функция дешева при ее оценивании, проста при хранении и имитирует случайную функцию, распределяя непатологические совокупности данных примерно поровну по позициям хеш-табличного массива.
- Эксперты опубликовали целый ряд хороших хеш-функций, которые вы можете использовать в своей работе.
- Размер хеш-таблицы должен периодически изменяться, с тем чтобы ее загрузка оставалась малой (например, менее 70 %).
- В случае критически важного исходного кода ничто не может заменить апробирование нескольких конкурирующих реализаций хеш-таблиц.
- Фильтры Блума также поддерживают операции Вставить и Просмотреть в постоянном времени, и они предпочтительнее хеш-таблиц в приложениях, в которых пространство ценится на вес золота, а периодическое ложное утверждение не является помехой для сделки.

Задачи на закрепление материала

Задача 12.1. Что из перечисленного ниже не является свойством, которое вы ожидаете от хорошо спроектированной хеш-функции?

а) Хеш-функция должна распределять каждый набор данных примерно равномерно по всему диапазону.

б) Хеш-функция должна быть проста при ее вычислении (за постоянное время или близко к нему).

в) Хеш-функция должна быть легкой в хранении (в постоянном пространстве или близко с ним).

г) Хеш-функция должна распределять большинство совокупностей данных примерно равномерно по всему диапазону.

Задача 12.2. Хорошая хеш-функция имитирует золотой стандарт случайной функции для всех практических целей, поэтому интересно исследовать коллизии с помощью случайной функции. Если местоположения двух разных ключей $k_1, k_2 \in U$ выбираются независимо и равномерно случайным образом в n позициях в массиве (причем все возможности являются равновероятными), то какова вероятность того, что k_1 и k_2 будут конфликтовать?

а) 0

б) $1/n$

в) $2/n(n-1)$

г) $1/n^2$

Задача 12.3. Мы интерпретировали наш эвристический анализ фильтров Блума в разделе 12.6, ограничив его на случае с 8 битами пространства в расчете на ключ, вставляемый в фильтр. Предположим, что мы готовы использовать вдвое больше места (16 бит на вставку). Что, согласно нашему эвристическому анализу, вы можете сказать о соответствующей частоте ложных утверждений, исходя из допущения, что число m хеш-таблиц задано оптимально? (Выберите самое сильное истинное утверждение.)

а) Частота ложных утверждений будет меньше 1 %.

б) Частота ложных утверждений будет меньше 0,1 %.

в) Частота ложных утверждений будет меньше 0,01 %.

г) Частота ложных утверждений будет меньше 0,001 %.

Задача по программированию

Задача 12.4. Реализуйте на предпочитаемом вами языке программирования хеш-табличное решение задачи о сумме двух чисел (2-SUM) раздела 12.2.2. Например, вы можете сгенерировать список S из миллиона случайных целых чисел между -10^{11} и 10^{11} и подсчитать число целей t между −10 000 и 10 000, для которых существуют несовпадающие $x, y \in S$, где $x + y = t$.

Вы можете применить существующие реализации хеш-таблиц либо реализовать свои собственные с нуля. В последнем случае сравните вашу результативность в рамках разных стратегий разрешения коллизий, таких как сцепление и линейное зондирование. (Обратитесь к сайту www.algorithmsilluminated.org для получения тестовых случаев и совокупностей данных для задач.)

Приложение B. Краткий обзор асимптотической формы записи

В этом приложении рассматривается асимптотическая форма записи, в особенности обозначение *O*-большое. Если вы встречаете такой материал впервые, то, возможно, захотите дополнить это приложение более подробным описанием, например, главой 2 *первой части* либо соответствующими видеороликами с веб-сайта www.algorithmsilluminated.org[1]. Если вы встречали его раньше, то не обязательно читать данное приложение от корки до корки — освежите в памяти те моменты, которые того требуют.

B.1. Суть

Асимптотическая форма записи определяет золотую середину подробности рассуждений об алгоритмах и структурах данных. Она достаточно груба, чтобы подавлять все детали, которые вы хотите проигнорировать, — детали, которые зависят от выбора аппаратной архитектуры, выбора языка программирования, выбора компилятора и так далее. С другой стороны, она достаточно точна, чтобы выполнять полезные сравнения между разными алгоритмическими высокоуровневыми подходами к решению задачи, в особенности на больших входных данных (требующих алгоритмической изобретательности).

[1] См. также https://ru.wikipedia.org/wiki/Вычислительная_сложность. — *Примеч. пер.*

Хорошее семисловное резюме асимптотической формы записи выглядит так:

СИСТЕМА АСИМПТОТИЧЕСКИХ ОБОЗНАЧЕНИЙ В СЕМИ СЛОВАХ

устранить $\underbrace{\text{постоянные множители}}_{\text{слишком системно-зависимы}}$ и $\underbrace{\text{члены низших порядков}}_{\text{нерелевантны для больших входных данных}}$.

Важнейшим понятием асимптотического обозначения является обозначение *O*-большое. Интуитивно можно предположить, что нечто равное $O(f(n))$ для функции $f(n)$ означает, что $f(n)$ — это то, с чем вы остаетесь после устранения постоянных множителей и членов низших порядков. Например, если $g(n) = 6n \log_2 n + 6n$, то $g(n) = O(n \log n)$[1]. Обозначение *O*-большое распределяет алгоритмы по группам согласно их асимптотическим временам работы в худшем случае, таким как линейно-временные ($O(n)$) или логарифмически-временные $O(n \log n)$ алгоритмы и операции.

B.2. Обозначение *O*-большое

Обозначение *O*-большое относится к функциям $T(n)$, определенным на положительных целых числах $n = 1, 2, \ldots$ Для нас $T(n)$ почти всегда будет обозначать границу времени работы в худшем случае алгоритма или операции на структуре данных как функции от размера n входных данных.

ОБОЗНАЧЕНИЕ *O*-БОЛЬШОЕ (РУССКАЯ ВЕРСИЯ)

$T(n) = O(f(n))$ тогда и только тогда, когда $T(n)$ в конечном итоге ограничена сверху постоянным кратным функции $f(n)$.

[1] При игнорировании постоянных множителей нам не нужно указывать основание логарифма. (Разные логарифмические функции отличаются только на постоянный множитель.)

Вот соответствующее математическое определение обозначения *O*-большое, определение, которое вы должны использовать в формальных доказательствах.

ОБОЗНАЧЕНИЕ *O*-БОЛЬШОЕ (МАТЕМАТИЧЕСКАЯ ВЕРСИЯ)

$T(n) = O(f(n))$ тогда и только тогда, когда существуют положительные константы c и n_0, такие что

$$T(n) \leq c \times f(n) \qquad \text{(B.1)}$$

для всех $n \geq n_0$.

Константа c квантифицирует термин «постоянное кратное», константа n_0 квантифицирует словосочетание «в конечном счете». Например, на рис. B.1 константа c соответствует 3, а n_0 соответствует точке пересечения функций $T(n)$ и $c \times f(n)$.

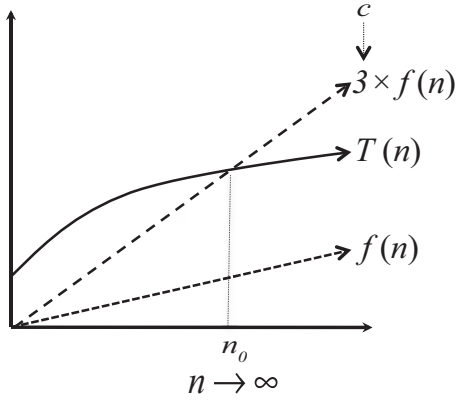


Рис. B.1. Рисунок, иллюстрирующий, когда $T(n) = O(f(n))$. Константа квантифицирует термин «постоянное кратное» функции $f(n)$, а константа n_0 квантифицирует словосочетание «в конечном итоге»

ПРЕДОСТЕРЕЖЕНИЕ

Когда мы говорим, что c и n_0 являются константами, мы имеем в виду, что *они не могут зависеть от n*. Например, на рис. B.1 c и n_0 были фиксированными числами (например, 3 или 1000), а затем мы рассмотрели

неравенство (B.1), по мере того как n растет сколь угодно большим (на графике в направлении вправо до бесконечности). Если вы когда-нибудь обнаружите, что в предполагаемом доказательстве *O*-большое говорите «взять $n_0 = n$» или «взять $c = \log_2 n$», то вам нужно начать заново с выбора c и n_0, которые являются независимыми от n.

B.3. Примеры

Мы утверждаем, что если $T(n)$ является многочленом с некоторой степенью k, то $T(n) = O(n^k)$. Следовательно, обозначение *O*-большое действительно устраняет постоянные множители и члены низших порядков.

Утверждение B.1. *Предположим, что*

$$T(n) = a_k n_k + ... + a_1 n + a_0,$$

где $k \geq 0$ — это неотрицательное целое число, a_i — вещественные числа (положительные или отрицательные). Тогда $T(n) = O(n^k)$.

Доказательство: доказательство формального утверждения *O*-большое сводится к реконструированию соответствующих значений для констант c и n_0 — здесь, чтобы было легко проследить, мы вытащим значения для этих констант из шляпы: $n_0 = 1$ и c, равные сумме абсолютных значений коэффициентов[1]:

$$c = |a_k| + ... + |a_1| + |a_0|.$$

Оба эти числа не зависят от n. Теперь нам нужно показать, что эти варианты констант удовлетворяют определению, имея в виду, что $T(n) \leq c \times n^k$ для всех $n \geq n_0 = 1$.

Для того чтобы перепроверить это неравенство, зададим положительное целое число $n \geq n_0 = 1$. Нам нужна последовательность верхних границ на $T(n)$, кульминацией которых станет верхняя граница $c \cdot n^k$. Сначала применим определение $T(n)$:

$$T(n) = a_k n^k + ... + a_1 n + a_0.$$

[1] Напомним, что *абсолютное значение* $|x|$ вещественного числа x равняется x, когда $x \geq 0$, и $-x$, когда $x \leq 0$. В частности, $|x|$ всегда неотрицательно.

Если взять абсолютное значение каждого коэффициента a_i в правой части, то выражение становится только больше. ($|a_i|$ может быть только больше, чем a_i, и поскольку n^i является положительным, $|a_i|n^i$ может быть только больше, чем $a_i n^i$.) Это означает, что

$$T(n) \leq |a_k|n^k + ... + |a_1|n + |a_0|.$$

Теперь, когда коэффициенты неотрицательны, мы можем использовать аналогичный метод, чтобы превратить разные степени n в общую степень n. Поскольку $n \geq 1$, n^k больше только n^i для каждого $i \in \{0, 1, 2, ..., k\}$. Поскольку $|a_i|$ является неотрицательной, $|a_i|n^k$ больше только $|a_i|n^i$. Это означает, что

$$T(n) \leq |a_k|n^k + \ldots + |a_1|n^k + |a_0|n^k = \underbrace{(|a_k| + \ldots + |a_1| + |a_0|)}_{=c} n^k.$$

Это неравенство соблюдается для каждого $n \geq n_0 = 1$, то есть именно то, что мы хотели доказать. *Ч. т. д.*

Мы также можем использовать определение обозначения *O*-большое, для того чтобы доказать, что одна функция не является *O*-большим другой функции.

Утверждение В.2. *Если* $T(n) = 2^{10n}$, *то* $T(n)$ *не является* $O(2^n)$.

Доказательство: чтобы доказать, что одна функция не является *O*-большое другой, как правило, пользуются доказательством от обратного. Итак, допустим противоположную формулировку в утверждении, что $T(n)$ на самом деле равняется $O(2^n)$. По определению обозначения *O*-большое существуют положительные константы c и n_0, так что

$$2^{10n} \leq c \times 2^n$$

для всех $n \geq n_0$. Так как 2^n является положительным числом, мы можем вычеркнуть его с обеих сторон этого неравенства, получив

$$2^{9n} \leq c$$

для всех $n \geq n_0$. Но это неравенство явно ложное: правая сторона является фиксированной константой (независимой от n), в то время как левая сторона уходит в бесконечность, по мере того как n становится большим. Это показы-

вает, что наше допущение о том, что $T(n) = O(2^n)$ не может быть правильным, и мы можем заключить, что 2^{10n} не является $O(2^n)$. *Ч. т. д.*

В.4. Обозначения Омега-большое и Тета-большое

Обозначение *O*-большое на сегодняшний день является наиболее важной и повсеместной концепцией для обсуждения асимптотического времени выполнения алгоритмов и операций на структурах данных. С парой его близких родственников, обозначениями Омега-большое и Тета-большое, также стоит познакомиться. Если *O*-большое является аналогом выражения «меньше или равно (≤)», то Омега-большое и Тета-большое соответственно аналогичны выражениям «больше или равно (≥)» и «равно (=)».

Формальное определение обозначения Омега-большое совпадает с обозначением *O*-большое. На русском языке мы говорим, что одна функция $T(n)$ является Омега-большое другой функции $f(n)$ тогда и только тогда, когда $T(n)$ в конечном итоге ограничивается снизу постоянным кратным $f(n)$. В этом случае мы пишем $T(n) = \Omega(f(n))$. Как и раньше, мы используем две константы c и n_0 для квантификации термина «постоянное кратное» и выражения «в конечном итоге».

ОБОЗНАЧЕНИЕ ОМЕГА-БОЛЬШОЕ

$T(n) = \Omega(f(n))$ тогда и только тогда, когда существуют положительные константы c и n_0, так что

$$T(n) \geq c \times f(n)$$

для всех $n \geq n_0$.

Обозначение Тета-большое, или просто Тета, аналогично выражению «равняется». Говоря, что $T(n) = \Theta(f(n))$, имеется в виду, что $T(n) = \Omega(f(n))$ и $T(n) = O(f(n))$. Равным образом, $T(n)$ в конечном итоге зажата между двумя разными постоянными кратными $f(n)$.

ОБОЗНАЧЕНИЕ ТЕТА-БОЛЬШОЕ

$T(n) = \Theta(f(n))$ тогда и только тогда, когда существуют положительные константы c_1, c_2 и n_0,так что

$$c_1 \times f(n) \leq T(n) \leq c_2 \times f(n)$$

для всех $n \geq n_0$.

ПРЕДОСТЕРЕЖЕНИЕ

Поскольку проектировщики алгоритмов особо сосредоточены на гарантиях времени выполнения (то есть верхних границах), они, как правило, используют обозначение *O*-большое, даже когда обозначение Тета-большое было бы более точным; например, заявляя о времени выполнения алгоритма как $O(n)$, даже если понятно, что оно равно $\Theta(n)$.

Следующее далее тестовое задание проверяет ваше понимание обозначений *O*-большое, Омега-большое и Тета-большое.

ТЕСТОВОЕ ЗАДАНИЕ В.1

Пусть $T(n) = 1/2n^2 + 3n$. Какие из следующих утверждений являются истинными? (Выберите все подходящие варианты.)

а) $T(n) = O(n)$

б) $T(n) = \Omega(n)$

в) $T(n) = \Theta(n^2)$

г) $T(n) = O(n^3)$

(Решение и пояснение см. ниже.)

Правильные ответы: (б), (в), (г). Все три последних ответа являются правильными, и, надо надеяться, интуитивно понятно, почему. $T(n)$ является

квадратичной функцией. Линейный член $3n$ не имеет значения для больших n, поэтому следует ожидать, что $T(n) = \Theta(n^2)$ (ответ (в)). Из этого автоматически вытекает, что $T(n) = \Omega(n^2)$ и, следовательно, также $T(n) = \Omega(n)$ (ответ (б)). Схожим образом, из $T(n) = \Theta(n^2)$ вытекает, что $T(n) = \Theta(n^2)$ и, следовательно, также $T(n) = \Theta(n^3)$ (ответ (г)). Доказательство этих утверждений формально сводится к демонстрации соответствующих констант, удовлетворяющих определению. Например, взяв $n_0 = 1$ и $c = 1/2$, мы доказываем (б). Взяв $n_0 = 1$ и $c = 4$, мы доказываем (г). Объединив эти константы ($n_0 = 1$, $c_1 = 1/2$, $c_2 = 4$), мы доказываем (в). Доказательство от противного в духе утверждения В.2 показывает, что (а) не является правильным ответом.

Решения отдельных задач

Задача 7.1. Условия (а) и (в) удовлетворяются некоторыми разреженными графами (такими как звездный граф) и некоторыми плотными графами (такими как полный граф с одним наклеенным дополнительным ребром). Условие (б) выполняется только разреженными графами, а условие (г) только плотными графами.

Задача 7.2: (в). Следует просмотреть строку, соответствующую v в матрице смежности.

Задача 8.1. Все четыре утверждения соблюдаются: (а) алгоритмом `UCC` из раздела 8.3; (б) алгоритмом `Augmented-BFS` из раздела 8.2; (в) алгоритмом `Kosaraju` из раздела 8.6; (г) по алгоритму `TopoSort` из раздела 8.5.

Задача 8.2: (в). Требуется время $\Omega(n^2)$, потому что в худшем случае правильный алгоритм должен заглянуть в каждый из n^2 элементов матрицы смежности минимум один раз. Время $O(n^2)$ достижимо, например, путем конструирования представления входного графа в виде списков смежности с одним проходом по матрице смежности (за время $O(n^2)$), а затем выполнения алгоритма `DFS` с новым представлением за время $O(m + n) = O(n^2)$.

Задача 8.7: (в). Вычисление «волшебного упорядочивания» при первом проходе алгоритма `Kosaraju` требует поиска в глубину. (См. доказательство теоремы 8.10.) На втором проходе, с учетом волшебного упорядочивания вершин, любой экземпляр алгоритма `GenericSearch` (включая `BFS`) успешно обнаружит сильно связную компоненту в обратном топологическом порядке.

Задача 8.8: (а), (б). Модификация в (а) не изменяет порядок, в котором алгоритм рассматривает вершины во втором проходе, и поэтому он остается правильным. Модификация в (б) соответствует работе алгоритма `Kosaraju` на развороте входного графа в обратную сторону. Поскольку граф и его разворот в обратную сторону имеют точно такие же сильно связные компоненты (тестовое задание 8.6), алгоритм остается правильным. Модификации в (в) и (г) эквивалентны, как и в приведенном выше аргументе (а), и не приводят к правильному алгоритму. С целью рассмотрения контрпримера вернитесь к нашему примеру (и в особенности к обсуждению на с. 91).

Задача 9.2: (б). Две суммы несовпадающих степеней двойки не могут быть одинаковыми. (Представьте, что числа записаны в двоичном формате.) Для (а) и (в) существуют контрпримеры с тремя вершинами и тремя ребрами.

Задача 9.3: (в), (б). Утверждение (г) соблюдается, потому что когда P имеет только одно ребро, каждый путь увеличивается в длину по крайней мере на столько же, сколько P. Это также показывает, что (б) является ложным. Пример, подобный примеру в разделе 9.3.1, показывает, что (а) является ложным, и из этого следует, что (в) является истинным.

Задача 9.7. В строках 4 и 6 алгоритма `Dijkstra` (с. 115) следует заменить $\text{len}(v) + l_{vw}$ на $\max\{\text{len}(v), l_{vw}\}$ и $\text{len}(v^*) + l_{v^*w^*}$ на $\max\{\text{len}(v^*), l_{v^*w^*}\}$ соответственно.

Задача 10.1: (б), (в). Смысл существования кучи заключается в поддержке быстрых вычислений минимума, при этом алгоритм `HeapSort` (раздел 10.3.1) является каноническим приложением. Взятие ключа каждого объекта с противоположным знаком превращает кучу в структуру данных, которая поддерживает быстрые вычисления максимумов. Кучи обычно не поддерживают быстрые операции просмотра, если только вы не ищете объект с минимальным ключом.

Задача 10.4: (а). С помощью одной кучевой операции можно извлечь только объект с наименьшим ключом. Вызов операции Извлечь минимум пять раз подряд возвращает объект в куче с пятым наименьшим ключом. Для извлечения объекта с медианным или максимальным ключом потребуется линейное число кучевых операций.

Задача 10.5. В строке 14 реализации алгоритма `Dijkstra` на основе кучи (с. 150) следует заменить $\text{len}(w^*) + l_{w^*y}$ на $\max\{\text{len}(w^*), l_{w^*y}\}$.

Задача 11.1: (а). Утверждение (а) соблюдается, потому что на i-м уровне бинарного дерева существует не более i узлов и, следовательно, не более $1 + 2 + 4 + \cdot + 2^i \leq 2^{i+1}$ узлов на уровнях от 0 до i вместе взятых. Для размещения n узлов требуется $2^{h+1} \geq n$, где h — это высота дерева, поэтому $h = \Omega(\log n)$. Утверждение (б) соблюдается для сбалансированных бинарных деревьев поиска, но обычно является ложным для несбалансированных бинарных деревьев поиска (см. сноску 2 на с. 172 в главе 11).

Утверждение (в) является ложным, поскольку свойства кучи и дерева поиска несопоставимы (см. с. 1174). Утверждение (г) является ложным, так как отсортированный массив предпочтительнее сбалансированного бинарного дерева поиска, когда множество хранимых объектов статично, без вставок и удалений (см. с. 171).

Задача 12.1: (а). Патологические совокупности данных показывают, что свойство (а) невозможно и поэтому не может быть ожидаемым (см. раздел 12.3.6). Остальные три свойства удовлетворяются современными хеш-функциями.

Задача 12.2: (б). Существует n возможностей для местоположений k_1 и n возможностей для местоположений k_2, в общей сложности n^2 исходов. Из них k_1 и k_2 конфликтуют ровно в n из них — исход, в котором обоим назначается первая позиция, исход, в котором обоим назначается вторая позиция, и так далее. Поскольку каждый исход равновероятен (с вероятностью $1/n^2$ каждый), вероятность коллизии равна $n \times 1/n^2 = 1/n$.

Тим Рафгарден

Совершенный алгоритм. Графовые алгоритмы и структуры данных

Перевел с английского *А. Логунов*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*А. Логунов*
Литературный редактор	*А. Тазеева*
Художественный редактор	*В. Мостипан*
Корректоры	*М. Молчанова, Г. Шкатова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214,
тел./факс: 208 80 01.

Подписано в печать 08.05.19. Формат 70x100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1200. Заказ 0000.