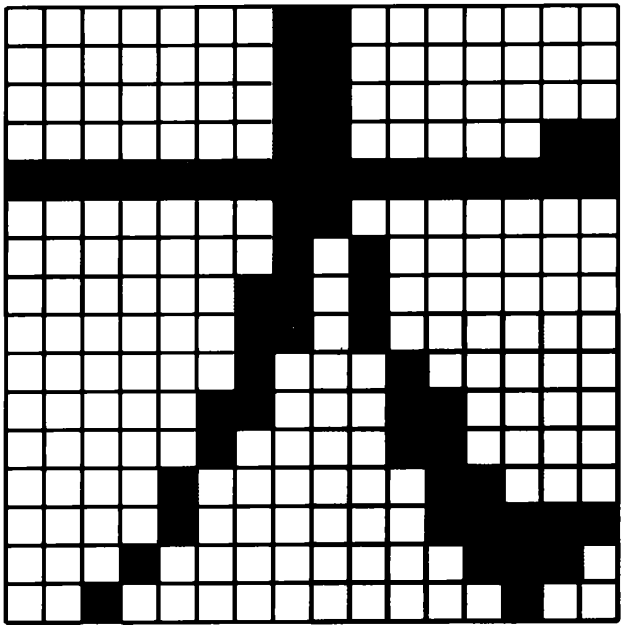
И. В. Романовский



Дискретный анализ

Четвертое издание,
исправленное и дополненное

bhv®

НЕВСКИЙ ДИАЛЕКТ

И. В. Романовский

ДИСКРЕТНЫЙ АНАЛИЗ

Издание 4-е, исправленное
и дополненное

*Допущено учебно-методическим объединением
на базе Санкт-Петербургского университета
Министерства образования Российской Федерации
в качестве учебного пособия по специальности 351500
"Математическое обеспечение и администрирование
информационных систем"*

Санкт-Петербург
2008

УДК 519.1
ББК 22.176
Р69

Романовский И. В.

Р69 Дискретный анализ: Учебное пособие для студентов, специализирующихся по прикладной математике и информатике. — 4-е изд., испр. и доп. — СПб.: Невский Диалект; БХВ-Петербург, 2008. — 336 с.: ил.

Пособие написано по материалам вводного лекционного курса, который автор читает на математико-механическом факультете Санкт-Петербургского государственного университета студентам, специализирующимся по прикладной математике и информатике. Особое внимание уделяется связям между понятиями дискретного анализа, возникающими в разных разделах математики и современной информатики.

В это издание включено много новых материалов, в связи с чем изменилась структура книги: появились новые главы и параграфы. Увеличено число упражнений. Текст дополнен алфавитным указателем и библиографическими рекомендациями.

ISBN 5-7940-0138-0 ("Невский Диалект")

Оглавление

Введение

Интерес к дискретной математике вполне понятен. Он объясняется постоянно растущей в последние десятилетия потребностью в ней в связи с развитием широких областей ее применения — в электронике, информатике, многочисленных вопросах оптимизации. Сейчас повсеместно курсы дискретной математики входят в программы обучения математиков и инженеров. За последние годы появились разнообразные учебники по этой дисциплине, по содержанию разительно не похожие друг на друга.

Предлагаемый курс, также сильно отличающийся от остальных, предназначен для *первоначального знакомства* с предметом. В нем много внимания уделяется терминологии и сопоставлению подходов к одному и тому же математическому объекту с точки зрения различных математических дисциплин, в ущерб глубине изложения. Нужно сразу же предупредить, что многие из рассматриваемых вопросов заслуживают самостоятельных курсов и это краткое изложение не в силах их заменить.

Понятия и факты, которые мы будем изучать, относятся к таким областям элементарной математики, как комбинаторика и арифметика, а также к теории вероятностей, математической логике, теории графов, теории дискретной оптимизации и даже к классическому математическому анализу, который, казалось бы, далек от дискретности. И конечно же, значительную часть курса должен составлять материал из той области математики, которую американцы называют Computer Science, а мы все чаще — информатикой.

Мы начнем с некоторых известных понятий теории множеств и математической логики, лишь незначительно расширив набор используемых понятий. Затем будут введены понятия классической комбинаторики, причем интерес к вопросам, связанным с вычислениями, будет несколько большим, чем обычно.

От комбинаторики прямой путь ведет нас к элементарной теории вероятностей, в которой подсчеты количества "благоприятных" исходов требуют обычно именно комбинаторной техники. Разумеется, нормальное университетское образование предполагает серьезный курс теории вероятностей. Такой курс сам по себе нуждается в солидной математической подготовке и поэтому читается позднее. Но некоторые из рассматриваемых нами вопросов требуют начальных

вероятностных представлений — поэтому разумно потратить немного времени на введение нужных понятий.

От вероятности нам откроется путь к некоторым вопросам теории информации и, в частности, к идеям оптимального кодирования текстов в целях их сокращения и повышения надежности передачи и к задачам оптимального поиска. Оба эти вопроса очень важны для современных информационных систем.

В связи с задачами обработки информации мы рассмотрим некоторые алгоритмы работы со строками переменной длины, а также наиболее важные способы организации информации.

Далее последуют элементы теории предикатов (отношений) и связанные с ними понятия частичного упорядочения. Будет рассмотрено использование понятия *отношения* в реляционных базах данных.

Здесь нам будет удобно ввести некоторые определения теории конечных графов, впрочем, теория графов как одна из наиболее характерных областей дискретного анализа привлечет наше внимание надолго. Одной из важнейших тем, связанных с графами, традиционно считаются экстремальные задачи на графах, в курсе будут рассмотрены наиболее важные из них. Материал, относящийся к экстремальным задачам, в этом издании увеличен.

Затем мы рассмотрим несколько типов *процессов* — специфических математических объектов, отличительной особенностью которых является возможность находиться в различных состояниях и переходить из состояния в состояние в результате внешних воздействий. Вычислительные процессы без их формального определения нам встретятся сразу же, как только мы начнем описывать алгоритмы. Но несколько специальных процессов из различных областей математики — *марковские цепи* из теории вероятностей, *конечные автоматы* и *управляемые процессы* — заслуживают отдельного внимания. Мы вкратце обсудим также некоторые использования понятия *процесса* в программировании.

В заключение будут затронуты несколько вопросов, связывающих дискретный анализ с классическим, — это вопросы асимптотического поведения комбинаторных формул и производящие функции, которые дают прекрасное средство для получения точных формул.

Я считал необходимым приводить в достаточном количестве конкретные примеры. Иногда они могут показаться слишком громоздкими, но, к сожалению, совсем простые примеры не всегда хороши психологически — неопытный читатель не видит, в чем проблема, зачем "городить огород", когда "и так все ясно".

Читателю полезно прорабатывать эти примеры — не обязательно до конца, но до полного понимания хода вычислений. Очень полезны компьютерные демонстрации. Студенты математико-механического факультета СПбГУ, слушавшие этот курс, приготовили ряд демонстрационных программ. Эти программы (их уже накопилось довольно много, мы с П. Г. Черкасовой объединили их в легко пополняемую систему) доступны в соответствии с принципом AS IS (КАК ЕСТЬ, без ответственности за ошибки) для любого желающего.

В этом издании немного увеличено число упражнений, добавлены алфавитный указатель и небольшие библиографические рекомендации.

По предыдущим изданиям я получил большое число критических замечаний и предложений, особенно много от С. С. Лаврова, И. Л. Миронова, Н. Н. Петрова, А. Л. Смирнова, С. Е. Столяра. С. В. Кузнецов существенно дополнил изложение вопросов криптографии. Я благодарен им и всем другим, кто своими замечаниями и предложениями способствовал улучшению курса. Буду благодарен и будущим критикам, появление которых предвижу, поскольку работа над текстом книги продолжается.

Август 2002 г. *И. Романовский*

E-mail: jvr@jr2793.spb.edu

Обозначения

- $\square$ — конец доказательства
- $\boxtimes$ — КОнец постановки ЗАдачи (*шеврон* восходит к фр. chevre — коза)
- $\triangleq$ — "равно по определению"
- $k : l$ — множество целых чисел от k до l
- $|A|$ — мощность (число элементов) множества A
- $\lfloor x \rfloor$ — округление числа x до ближайшего целого вниз
- $\lceil x \rceil$ — округление числа x до ближайшего целого вверх

Глава 1

Некоторые определения из теории множеств

1.1. Основные определения

В этом курсе нам предстоит рассматривать главным образом конечные множества. Поскольку теория множеств сейчас входит в школьную программу, не будем все повторять с начала, а просто вспомним некоторые классические определения и обозначения.

Принадлежность элемента i множеству A будет обозначаться так: $i \in A$. Пустое множество — $\varnothing$. Множество целых чисел от k до l будет обозначаться (это не общепринято) $k : l$. Число элементов множества A называется его *мощностью* и обозначается через $|A|$.

Говорят, что множество A есть *подмножество* множества B, и пишут $A \subset B$, если каждый элемент A принадлежит B. Множество C называется *пересечением* множеств A и B и обозначается $C = A \cap B$, если оно состоит из всех элементов, которые принадлежат обоим этим множествам. Если пересечение множеств пусто, они называются *дизъюнктными*. Множество C называется *объединением* множеств A и B и обозначается $C = A \cup B$, если оно состоит из всех элементов, которые принадлежат хотя бы одному из этих множеств *). Аналогично определяются объединение и пересечение любого (конечного) числа множеств с обозначениями типа

$$\bigcup_{i \in 1:k} A_i, \qquad \bigcap_{i \in 1:k} A_i,$$

*) В англоязычной научной литературе знаки $\cup$ и $\cap$ часто называют cup — *чашка* и cap — *шапка*.

если речь идет о множествах A_i, $i \in 1:k$. Впрочем, множество индексов не обязательно должно быть вида $1:k$, это может быть произвольное множество I, и тогда мы пишем

$$\bigcup_{i\in I} A_i, \qquad \bigcap_{i\in I} A_i.$$

Множество C называется *разностью* множеств A и B и обозначается $C = A \setminus B$, если оно состоит из всех элементов, которые принадлежат A и не принадлежат B. Аналогично определяется разность $B \setminus A$. Объединение этих двух разностей называется *симметрической* разностью A и B.

Теперь введем несколько определений, которые встречаются не так часто.

1.2. Прямое произведение

Пусть заданы два множества A и B. Множество всех возможных упорядоченных пар (i, j), где $i \in A$, $j \in B$, называется *прямым произведением* множеств A и B и обозначается через $A \times B$ (часто используется термин *декартово произведение*[*]). Мощность прямого произведения множеств равна произведению мощностей этих множеств $|A \times B| = |A| \cdot |B|$.

Аналогично определяется прямое произведение любого (конечного) числа множеств. Очень часто нам будет встречаться случай, когда произведение составляется из нескольких экземпляров одного и того же множества. В частности, этим многократно используемым сомножителем может быть множество $\mathbf{B}$ из двух элементов[**], которые традиционно отождествляются с 0 и 1, $\mathbf{B} = 0:1$. Произведение m экземпляров такого множества естественно обозначить через $\mathbf{B}^m$, это множество $\mathbf{B}^m = (0:1)^m$ состоит из 2^m элементов.

Элементами $\mathbf{B}^m$ являются наборы из m нулей и единиц, выстроенных в последовательность. Например, $(1, 1, 0, 1, 0) \in \mathbf{B}^5$. Хотелось бы различать отдельные компоненты таких наборов. Можно приписать каждому месту в последовательности его индекс. Индексация может быть различной, сразу приходят в голову записи $(1_1, 1_2, 0_3, 1_4, 0_5)$ и $(1_0, 1_1, 0_2, 1_3, 0_4)$, соответствующие нумерации этих мест от

[*] По имени Рене Декарта (Rene Descartes, 1596–1650), знаменитого французского математика и философа (ср. *декартовы координаты*). В английском языке прилагательному *декартов* соответствует Cartesian — по латинизированной форме его имени Cartesius.

[**] В некоторых областях математики множество $\mathbf{B}$ называется *двоеточием*.

единицы и от нуля. Но можно использовать и не нумерующие индексы, взять, например, множество индексов $I = \{ru, by, ua, jp, ch\}$ и его элементами помечать компоненты пятерки: $(1_{ru}, 1_{by}, 0_{jp}, 1_{ua}, 0_{ch})$. Такая индексация подсказывает нам, что элементы в записи можно переставлять: $(1_{by}, 0_{ch}, 0_{jp}, 1_{ru}, 1_{ua})$ обозначает ту же самую запись, и мы не требуем никакой упорядоченности индексов. Используя при определении декартова произведения свободную индексацию, мы будем в показателе степени ставить множество индексов $\mathbf{B}^{0:4}$, $\mathbf{B}^{1:5}$, $\mathbf{B}^{I}$.

В произведениях произвольных множеств будет всегда предполагаться свободная индексация. Немедленно воспользуемся открывающимися возможностями.

Итак, пусть задано конечное множество I, множество *индексов*, и каждому индексу $i \in I$ сопоставлено непустое множество M_i. Назовем *I-набором* совокупность элементов $\{a_i \mid i \in I, a_i \in M_i\}$, в которой каждому индексу $i \in I$ соответствует элемент из M_i. Множество $M(I)$ всех возможных I-наборов и называется прямым произведением множеств M_i.

Часто встречается операция *продолжения* I-набора $a(I)$ или множества I-наборов $A(I) \subset M(I)$ до большего набора индексов, например J. *Продолжением* набора $a(I)$ на индексы J называется множество $a(I) \times M(J \setminus I)$, т.е. совокупность всех тех J-наборов, которые совпадают с $a(I)$ на индексах из I. *Продолжение* $A(I)$ на J — это объединение продолжений всех I-наборов $a(I) \in A(I)$.

Пример. В рассмотренном выше примере продолжение I_1-набора $a(I_1) = \{0_2, 0_7, 0_9\}$ — это совокупность всех 0 : 9-наборов, у которых нули стоят на местах, соответствующих индексам 2, 7 и 9.

Множества вида $A(I) \times M(J \setminus I)$ часто называются *цилиндрическими* по естественной аналогии с цилиндрами в геометрии: здесь $A(I)$ — сечение цилиндра, а $M(J \setminus I)$ — его образующая.

Для произвольных множеств индексов I и J, где $I \subset J$, рассмотрим множества $M(I)$ и $M(J)$ и подмножество $A_J \subset M(J)$. Назовем *проекцией* множества A_J на $M(I)$ совокупность всех таких I-наборов, которые можно получить из элементов A_J выделением компонент, соответствующих I. Обозначим эту проекцию через $\mathbf{proj}_I A_J$.

Можно было бы сказать, что для получения проекции $\mathbf{proj}_I A_J$ мы берем всевозможные наборы $a_j \in A_J$ и в каждом таком наборе оставляем только компоненты с индексами из I. Все? Нет, среди получающихся наборов могут быть одинаковые, и их нужно отождествить. Оставшиеся различные наборы и образуют проекцию.

Теорема. Проекция множества $A_J \subset M(J)$ на $M(I)$ — это минимальное множество $B_I \subset M(I)$, в продолжении которого на M_J содержится все множество A_J.

Доказательство. Покажем, что если в продолжении B_I содержится все множество A_J, то $\mathbf{proj}_I A_J \subset B_I$. Пусть какая-то точка a из $\mathbf{proj}_I A_J$ не содержится в B_I. Но тогда и ее продолжение не содержится в продолжении B_I вопреки предположению. Вместе с тем любая точка из B_I, не входящая в $\mathbf{proj}_I A_J$, может быть удалена из B_I без изменения его свойств. □

Таким образом, B_I — это сечение цилиндра $B_I \times M(J \setminus I)$, содержащего все множество A_J.

1.3. Разбиения

Совокупность $A_1, A_2, \ldots, A_k$ непустых попарно дизъюнктных подмножеств множества M называется его *разбиением*, если M равно их объединению:

$$M = A_1 \cup A_2 \cup \cdots \cup A_k, \qquad A_i \cap A_j = \varnothing, \quad i \neq j.$$

Разбиение $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ множества M называется *измельчением* разбиения $\mathcal{B} = \{B_1, B_2, \ldots, B_l\}$ того же множества, если каждое из множеств A_i полностью содержится в одном из множеств B_j, т.е. *для каждого* $i \in 1:k$ *найдется единственное* $j \in 1:l$, *такое что* $A_i \subset B_j$. Будем говорить, что $\mathcal{B}$ *крупнее*, чем $\mathcal{A}$ (рис. 1.1). Для записи этого отношения между разбиениями будет использоваться знак $\sqsubset$: $\mathcal{A} \sqsubset \mathcal{B}$.

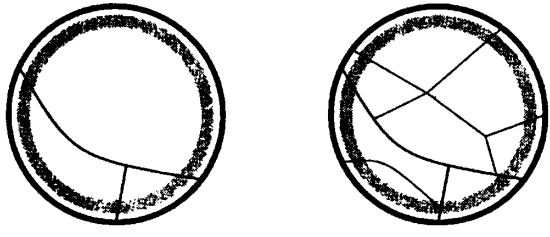


Рис. 1.1. На левом рисунке разбиение крупнее, чем на правом

Например, разбиение множества студентов факультета по курсам крупнее разбиения по учебным группам.

Разбиение часто возникает при формировании полных прообразов значений функции: если взять какое-либо отображение $f : M \to R$ и сопоставить каждому $r \in R$ множество

$$A_r = \check{f}(r) = \{i \in M \mid f(i) = r\},$$

то набор непустых множеств A_r будет обладать всеми свойствами разбиения. Обозначим это разбиение через $\mathcal{A}_f$.

Если заданы три множества M, R и S и отображения $f : M \to R$ и $g : R \to S$, то суперпозиция этих отображений, т.е. отображение $gf : M \to S$, где $(gf)(i) = g(f(i))$, задает разбиение $\mathcal{A}_{gf}$, которое крупнее, чем разбиение $\mathcal{A}_f$. (Попробуйте это доказать.)

Возьмем теперь два разбиения множества M: $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ и $\mathcal{B} = \{B_1, B_2, \ldots, B_l\}$. Разбиение $C = \{C_1, C_2, \ldots, C_m\}$ называется *произведением* разбиений $\mathcal{A}$ и $\mathcal{B}$ и обозначается $C = \mathcal{A} \bullet \mathcal{B}$, если оно является измельчением обоих этих разбиений, причем самым крупным из таких измельчений (рис. 1.2).

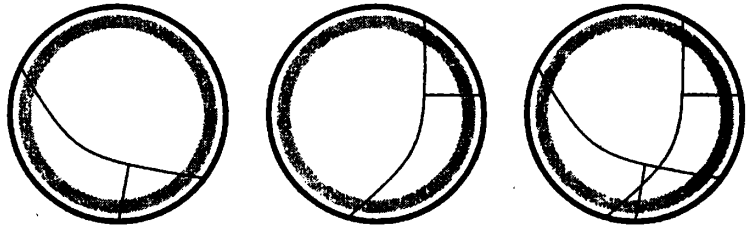


Рис. 1.2. Два разбиения и их произведение

Теорема. Произведение двух разбиений существует.

Доказательство. Взяв два произвольных разбиения $\mathcal{A}$ и $\mathcal{B}$, мы просто предъявим разбиение, которое будет их произведением. Возьмем все множества D_{ij} вида $D_{ij} = A_i \cap B_j$, $i \in 1 : k$, $j \in 1 : l$. Можно сказать, что множество индексов системы множеств D является прямым произведением множеств индексов $\mathcal{A}$ и $\mathcal{B}$. Очевидно, что подмножества D_{ij} будут дизъюнктными и их объединение равно M. Для того чтобы совокупность этих множеств образовывала разбиение, нужно только обеспечить их непустоту. Отбросив пустые множества, мы и получим разбиение C. Покажем, что это самое крупное общее измельчение $\mathcal{A}$ и $\mathcal{B}$.

Действительно, пусть наряду с C имеется еще одно такое разбиение $\mathscr{F} = \{F_1, F_2, \ldots, F_t\}$, которое мельче, чем $\mathscr{A}$ и $\mathscr{B}$. Покажем, что C крупнее, чем $\mathscr{F}$. Для любого множества F_k найдутся содержащие его множества A_{i_k} и B_{j_k}. Но, следовательно, и их (непустое!) пересечение $D_{i_k j_k}$ содержит F_k. □

Пример. Пусть $M = 0:11$. Определим разбиения

$$\mathscr{A}: A_1 = 0:3, \quad A_2 = 4:9, \quad A_3 = 10:11;$$

$$\mathscr{B}: B_1 = 0:5, \quad B_2 = 6:11.$$

Тогда

$$D_{11} = 0:3, \quad D_{21} = 4:5, \quad D_{31} = \varnothing,$$
$$D_{12} = \varnothing, \quad D_{22} = 6:9, \quad D_{32} = 10:11$$

и

$$C = \{D_{11}, D_{21}, D_{22}, D_{32}\}.$$

Рассмотрим в заключение связь разбиений и прямых произведений. Пусть заданы два множества: A с разбиением $\mathscr{A}$ и B с разбиением $\mathscr{B}$. На их произведении $C = A \times B$ легко определяется разбиение C, составленное из произведений $A_i \times B_j$, $A_i \in \mathscr{A}$, $B_j \in \mathscr{B}$, которое естественно было бы назвать *произведением* этих разбиений. Если сопоставить это с предыдущим определением, то мы увидим, что они легко согласуются.

Действительно, разбиения $\mathscr{A}$ и $\mathscr{B}$ легко превращаются в разбиения C_A и C_B множества C, если взять цилиндрические продолжения элементов этих разбиений. Разбиение C есть произведение именно этих разбиений: $C = C_A \bullet C_B$.

Упражнение 1.1. Докажите, что

$$\mathscr{A} \bullet \mathscr{B} = \mathscr{B} \bullet \mathscr{A},$$

$$\mathscr{A} \bullet (\mathscr{B} \bullet C) = (\mathscr{A} \bullet \mathscr{B}) \bullet C.$$

Упражнение 1.2. Докажите, что если $\mathscr{A}$ крупнее $\mathscr{B}$, то для любого C разбиение $\mathscr{A} \bullet C$ крупнее $\mathscr{B} \bullet C$.

Упражнение 1.3. Докажите, что если $\mathscr{A}$ и $\mathscr{B}$ крупнее C, то и $\mathscr{A} \bullet \mathscr{B}$ крупнее, чем C.

Упражнение 1.4. Докажите, что разбиение $\mathscr{A}_{gf}$ крупнее, чем $\mathscr{A}_f$.

Упражнение 1.5. Каково взаимоотношение разбиений потока студентов по году, дню недели и полной дате рождения?

1.3.1. Порядок и нумерация

Часто элементы множества можно сравнивать. Это значит, что для любых двух (несовпадающих) элементов a и b множества A можно сказать, что либо a *предшествует* b, либо b предшествует a. Такое сравнение можно рассматривать как развитие операций сравнения для чисел: мы можем сказать, что число x предшествует числу y, если $x < y$. А могли бы сказать, что x предшествует y, если $x > y$.

Мы много внимания уделим этому вопросу в дальнейшем, а здесь ограничимся одним специальным случаем, которого для начала будет достаточно и который можно рассматривать как введение в вопрос.

Скажем, что на множестве A задан *числовой порядок* ρ, если каждому элементу $a \in A$ сопоставлено некоторое число $\rho(a)$ и числа, сопоставляемые различным элементам, различны. Будем говорить, что a предшествует b в порядке ρ, а также что b следует за a в порядке ρ, и писать $a \prec_\rho b$, если $\rho(a) < \rho(b)$.

Числовые порядки ρ_1 и ρ_2 называются эквивалентными, если неравенства $\rho_1(a) < \rho_1(b)$ и $\rho_2(a) < \rho_2(b)$ равносильны, т. е. выполняются или не выполняются одновременно.

Числовой порядок называется *нумерацией*, если множество значений $\rho(a)$, $a \in A$, заполняет отрезок натурального ряда. Минимальное значение $\rho(a)$ называется началом нумерации.

Как правило, используются нумерации, начинающиеся от 0 или от 1. К сожалению, нам придется пользоваться и той и другой нумерацией *).

Лемма 1. Для каждого числового порядка можно построить эквивалентную ему нумерацию. Если два порядка эквивалентны, то соответствующие им нумерации совпадают (при одинаковом начале).

Лемма 2. Числовой порядок на множестве A порождает числовой порядок на любом его подмножестве $B \subset A$.

Эти леммы совсем очевидны, их доказательством мы заниматься не будем. Задачи, которые нам нужно решить безотлагательно, связаны с нумерацией элементов множества, разбитого на подмножества, и с нумерацией прямого произведения двух или нескольких множеств.

Рассмотрим какое-либо разбиение $\mathcal{A}$ множества A на подмножества $A_1, A_2, \ldots, A_k$. Заданные на этих подмножествах числовые порядки

*) Несколько лет назад горячо обсуждался вопрос о том, с чего начиналась нумерация лет новой эры: был ли год Рождества Христова нулевым или первым.

(возможно, порожденные числовым порядком на A) создают нумерацию на каждом из них, а затем и нумерацию на самом A, определяемую следующим образом.

Произвольно задается начало нумерации в множестве A_1. Для каждого следующего множества A_i в качестве начала нумерации выберем число, на единицу превышающее наибольшее в нумерации для A_{i-1}. Попросту говоря, перенумеруем все элементы из A_1, потом продолжим нумерацию на A_2 и т.д., вплоть до A_k. Будем говорить, что эта нумерация *порождена* разбиением $\mathcal{A}$.

Пример. Множество $A = 1:20$ разбито на подмножества $A_1 = \{2, 4, 6, 8, \ldots, 20\}$, $A_2 = \{3, 9, 15\}$, A_3 — все остальное. Каждое число имеет *локальный* номер — внутри элемента разбиения, а локальные номера переводятся в общие следующим образом:

Число	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Лок. номер	0	0	0	1	1	2	2	3	1	4	3	5	4	6	2	7	5	8	6	9
Общий номер	13	0	10	1	14	2	15	3	11	4	16	5	17	6	12	7	18	8	19	9

Такая нумерация будет неоднократно использоваться в дальнейшем. Она очень удобна и во многих вычислительных вопросах. Отметим, что при заданных локальных нумерациях в множествах A_i для создания общей нумерации достаточно определить смещения номеров d_i (так, в нашем примере $d_1 = 0$, $d_2 = 10$, $d_3 = 13$).

Элементы прямого произведения двух нумерованных множеств A и B легко перенумеровать, следуя изложенному принципу. Пусть элементы множеств A и B уже перенумерованы числами от 0 до $m-1$ и до $n-1$ соответственно. Разобьем множество $C = A \times B$ на цилиндрические множества $A \times \{b\}$, где b пробегает все множество B. В каждом из этих множеств в качестве локальных номеров возьмем номера из A. Дальше объединим эти локальные нумерации в общую в соответствии с номерами элементов b. Тогда нулевой элемент получит смещение $d_0 = 0$, первый элемент — смещение $d_1 = m$, второй — $d_2 = 2m$ и т.д. до $d_{n-1} = (n-1)m$. Таким образом, элемент $c = (a, b) \in C$ получит номер

$$\rho_C(c) = d_{\rho_B(b)} + \rho_A(a) = \rho_B(b) \cdot m + \rho_A(a).$$

Более формальное определение порядка и некоторых его обобщений еще ждет нас в дальнейшем.

Глава 2

Строки фиксированной длины

2.1. Векторы из нулей и единиц

В этом параграфе рассматривается простой, но очень важный объект — упорядоченный набор из фиксированного числа (для определенности из m) нулей и единиц, т.е. элемент множества $\mathbf{B}^m$, о котором уже говорилось раньше. Слово *упорядоченный* означает, что элементы этого набора — нумерованные, они занимают вполне определенные места в наборе, так что, например, наборы (1, 0, 1, 0, 1) и (1, 1, 1, 0, 0) различны, хотя и состоят из одинакового количества нулей и единиц.

Упорядоченный набор из чисел обычно называется *вектором*, m — *размерностью* вектора, каждый отдельный элемент набора — *компонентой* вектора.

Наборы из нулей и единиц (или просто 0–1 наборы) появляются в самых разнообразных математических моделях, и некоторые из них вы, по-видимому, уже знаете. Но часто оказывается полезно знать и совместно использовать различные варианты их трактовки, так что мы начнем с пополнения "коллекции".

(А). Начнем с того, что вектор — понятие геометрическое. В математике наравне со "школьными", знакомыми по собственному мироощущению одномерным, двумерным и трехмерным пространствами рассматриваются пространства произвольной большей размерности. Точкой в m-мерном пространстве является m-мерный вектор, каждая его компонента — эта одна из декартовых координат этой точки. Набор из нулей и единиц, рассматриваемый как точка этого пространства,

определяет *вершину куба*, построенного на ортах (единичных отрезках) координатных осей. На рис. 2.1 показаны одномерный, двумерный и трехмерный кубы, для бо́льших m постарайтесь представить себе нечто похожее.

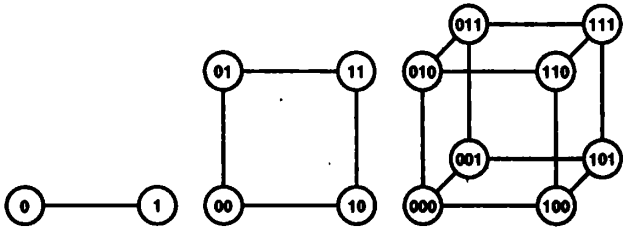


Рис. 2.1. Вершины единичного куба

(B). Используя 0–1 наборы, можно задавать *подмножества* множества $1:m$. Именно, желая задать множество $S \subset 1:m$, договоримся, что i-й элемент набора равен 1, если число i принадлежит S, и 0 — в противном случае. Определенный таким образом вектор называется *характеристическим вектором* множества S.

Представление множеств их характеристическими векторами удобно для выполнения основных операций над ними: например, характеристический вектор пересечения двух множеств имеет i-й компонентой 1, если i-е компоненты характеристических векторов обоих пересекаемых множеств равны 1.

Упражнение 2.1. Сформулируйте правила образования характеристических векторов объединения, разности и симметрической разности множеств.

(C). Очень близка к этой трактовке *логическая* интерпретация векторов из нулей и единиц. В XIX веке английский математик Дж. Буль *) предложил для математического моделирования понятий формальной логики использовать специальные переменные (называемые сейчас *логическими*, или *булевыми*), которые принимают два значения: TRUE и FALSE (ИСТИНА и ЛОЖЬ).

Над логическими величинами можно выполнять специальные *логические операции*: одноместная операция (т. е. имеющая один операнд) NOT (НЕТ) описывает *отрицание* — она вырабатывает значение TRUE

*) George Boole (1815–1864). Судя по словарям, английские лингвисты еще не успели оценить, как широко благодаря программированию распространился термин Boolean.

при значении аргумента FALSE, и наоборот. В математической логике эту операцию обозначают $\neg$.

Операция *логического умножения* AND (И), имеющая два операнда, вырабатывает значение TRUE только тогда, когда истинны оба операнда (эта операция называется также *конъюнкцией* (conjunction) и обозначается $\wedge$ или &). Можно сказать и по-другому: операция вырабатывает логическое значение "оба операнда истинны" *).

Операция *логического сложения* OR (ИЛИ) вырабатывает значение TRUE только тогда, когда истинен хотя бы один из операндов (эта операция называется *дизъюнкцией* (disjunction) и обозначается $\vee$ или |). Опять-таки можно сказать, что операция вырабатывает значение "хотя бы один из операндов истинен".

Операция EQU (EQUivalence — эквивалентность), или $\equiv$, вырабатывает значение "операнды равны"; она называется *тождественностью*. Наряду с ней встречается операция XOR (eXclusive OR — Исключающее ИЛИ), вырабатывающая ее отрицание, т.е. значение "истинен ровно один из ее операндов". Мы обозначим эту операцию символом $\not\equiv$ **).

Упражнение 2.2. Докажите, что (a XOR b) XOR b = a (это свойство операции XOR очень удобно в компьютерной графике: используя ее, можно создавать на экране временные объекты, например рамки, а затем легко восстанавливать прежнее состояние экрана).

Упражнение 2.3. Как мы знаем, точку на окружности можно задавать числом от 0 до 2π — углом доворота от начальной оси. Отрезок на окружности определяется упорядоченной парой (a, b), например $(1\frac{1}{2}\pi, \frac{1}{4}\pi)$. Докажите, что

$$x \in (a, b) \equiv ((a < b) \equiv ((a < x) \equiv (x < b))).$$

Операция IMP (IMPlication — следование), или $\supset$, вырабатывает значение "из истинности левого операнда следует истинность правого". Формально $(a \supset b) \equiv (\neg a \vee b)$. Может быть, естественнее выглядит запись $(a \supset b) \equiv \neg(a \wedge \neg b)$: единственный случай, когда из a не следует b, — это когда a истинно, а b ложно.

*) Некоторые молодые программисты боятся, иногда до старости, действий с логическими значениями и пишут, например, `if p=0 then IsZero:=True else IsZero:=False;` вместо простой записи `IsZero:=p=0;`

**) Общепринятого обозначения для XOR еще нет. Дональд Кнут [70] дал обстоятельный разбор встречающихся предложений, в котором присутствует и $\not\equiv$. Сам Кнут предпочитает обозначение $\oplus$, хотя, по-моему, оно должно употребляться наряду с другими обозначениями в кружочках. Нужно с вниманием отнестись и к обозначению $\veebar$, включенному в стандарт UNICODE.

Все эти операции легко формулируются в терминах нулей и единиц (см. таблицу) и переносятся на векторы — наборы из таких элементов. Операции над векторами выполняются *покомпонентно*, т. е. независимо над соответствующими компонентами векторов-операндов.

x	y	$x \wedge y$	$x \vee y$	$x \equiv y$	$x \not\equiv y$	$x \supset y$
0	0	0	0	1	0	1
0	1	0	1	0	1	1
1	0	0	1	0	1	0
1	1	1	1	1	0	1

Вот пример, в котором все логические операции выполняются над двумя векторами:

x	00011101010111001000111011101
y	11101010111010110101110101010
$x \wedge y$	00001000010010000000110001000
$x \vee y$	11111111111111111101111111111
$x \supset y$	11101010111010110111110101010
$x \equiv y$	00001000010010000010110001000
$x \not\equiv y$	11110111101101111101001110111

Сравнение двух последних строчек дает пример действия операции отрицания NOT, так как $(x \not\equiv y) = \neg(x \equiv y)$.

Нельзя не упомянуть здесь о логических функциях. *Логической*, или *булевой*, функцией называется функция от булевых переменных $x_1, \ldots, x_n$, принимающая булевы значения TRUE и FALSE. Любую булеву функцию можно задать таблицей, в которой каждому из 2^n наборов значений аргументов сопоставляется значение функции. Вот представление нескольких функций от трех переменных (значение 1 представляет TRUE, а значение 0 — FALSE):

x_1	0	1	0	1	0	1	0	1
x_2	0	0	1	1	0	0	1	1
x_3	0	0	0	0	1	1	1	1
$f_1 = x_1 \wedge (x_2 \not\equiv x_3)$	0	0	0	1	0	1	0	0
$f_2 = x_1 \wedge x_2$	0	0	0	1	0	0	0	1
$f_3 = (x_1 \supset x_2) \wedge (x_2 \supset x_3)$	1	0	0	0	1	0	1	1
f_4 = "ровно одно TRUE"	0	1	1	0	1	0	0	0

Это табличное задание функции легко превратить в формулу, состоящую из дизъюнкций конъюнкций (т.е. из логической суммы логических произведений), причем в каждой конъюнкции будут участвовать все n логических переменных или их отрицания. Формула получается, если каждому набору переменных, дающему функции значение TRUE, поставить в соответствие логическое произведение переменных, имеющих в наборе значение TRUE, и отрицаний переменных, имеющих значение FALSE, а затем составить логическую сумму таких наборов (предполагается, что хотя бы одному набору соответствует значение TRUE). Так, для f_1 получаем

$$f_1(x_1, x_2, x_3) = (x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3).$$

Обычно при работе с такими формулами знак логического умножения не пишут, а отрицание заменяют надчеркиванием (с таким упрощением напишем все формулы):

$$\begin{aligned}
f_1(x_1, x_2, x_3) &= x_1 x_2 \overline{x_3} \vee x_1 \overline{x_2} x_3, \\
f_2(x_1, x_2, x_3) &= x_1 x_2 x_3 \vee x_1 x_2 \overline{x_3}, \\
f_3(x_1, x_2, x_3) &= \overline{x_1 x_2 x_3} \vee \overline{x_1 x_2} x_3 \vee \overline{x_1} x_2 x_3 \vee x_1 x_2 x_3, \\
f_4(x_1, x_2, x_3) &= x_1 \overline{x_2 x_3} \vee \overline{x_1} x_2 \overline{x_3} \vee \overline{x_1 x_2} x_3.
\end{aligned}$$

Такие дизъюнкции конъюнкций именуются *дизъюнктивными нормальными формами* или сокращенно ДНФ. ДНФ, в каждом слагаемом которой участвуют все n переменных, называется *совершенной*. Эту совершенную форму часто можно сократить — построить эквивалентную и более короткую ДНФ. Возникает задача: по данной совершенной ДНФ построить эквивалентную ей ДНФ с минимальным количеством вхождений переменных. Например, для f_2 можно предложить форму

$$f_2(x_1, x_2, x_3) = x_1 x_2,$$

а для f_3 — форму

$$f_3(x_1, x_2, x_3) = \overline{x_1} x_3 \vee x_2 x_3.$$

Задачу о сокращенном (минимальном) представлении логической функции удобно формулировать в терминах множества вершин куба.

Введем в рассмотрение n-мерный единичный куб с множеством вершин $\mathbf{B}_{1:n}$. Каждой логической переменной x_i соответствует в кубе координата y_i, принимающая значения 0 или 1. Выберем какое-либо подмножество индексов $I \subset 1:n$ и зафиксируем значения координат y_i, $i \in I$, образуя I-набор. Продолжая его на $1:n$, т.е. составляя прямое произведение нашего набора с $\mathbf{B}_{1:n \setminus I}$, можно построить цилиндрическое множество, которое естественно назвать *гранью* куба. Пусть "сложность" задания грани определяется мощностью множества I. Рассмотрим непустое множество $A \subset \mathbf{B}_{1:n}$. Задача состоит в представлении A объединением граней с минимальной суммой сложностей. Мы вернемся к этой задаче позднее, на с. 269.

(D). Вектор из нулей и единиц может рассматриваться и как *двоичное представление натурального числа* (вы, конечно, знакомы с двоичной системой счисления и, наверное, уже давно ожидали этой интерпретации). Число представляется в виде суммы степеней 2, и присутствие в сумме слагаемого 2^k сопровождается появлением 1 в k-й позиции вектора. Например, число $1575 = 1024 + 512 + 32 + 4 + 2 + 1$ представляется строкой 11000100111. Обычно рассматриваются векторы с фиксированным числом компонент, например 16-разрядные. При 16 разрядах наше число запишется так*):

0000011000100111.

Но оно может быть записано и так:

1110010001100000,

если мы договоримся, что разряды числа нумеруются так, как это принято для компонент вектора, и младшие разряды числа записываются спереди. Конечно, первая запись нам привычнее по обычной арифметике, но особенности вычислительных машин иногда заставляют пересматривать наши привычки.

Двоичная запись очень удобна для компьютера, но человеку с ней разбираться трудно. Однако несколько арифметических действий в такой записи становятся более простыми и для человека. Прежде всего, это умножение и деление числа на 2. Умножение на 2 в двоичной системе счисления — это умножение на 10, т. е. приписывание нуля к записи. В компьютерах предусмотрена специальная операция, выполняющая это действие, — сдвиг набора на одну позицию влево (принято "человеческое" обозначение). Аналогично, деление на 2 соответствует сдвигу набора вправо. При таком сдвиге крайние единицы могут пропадать, за этой потерей информации нужно специально следить**).

*) Перевод из десятичной записи в двоичную вызывает у некоторых студентов трудности. Смотрите, как это разложение получается с помощью последовательного деления на 2 с остатком. Не пишите лишнего, составьте таблицу:

Делимые	41	20	10	5	2	1
Остатки	1	0	0	1	0	1

и получите $41_{10} = 101001_2$.

**) На самом деле обычно в системе команд предусматриваются два варианта сдвига: сдвиг обычный, при котором крайние единицы пропадают, и сдвиг циклический, при котором они появляются на другом конце вектора.

Еще одно действие (кстати, тоже особо предусмотренное в современных компьютерах) — прибавление к числу единицы. Полезно уметь его делать "вручную", так как иногда приходится создавать собственную программную поддержку каких-либо действий арифметики для манипуляций с "длинными числами", размер которых превосходит стандартные возможности процессора.

Для *прибавления единицы к целому числу, представленному в двоичном виде*, нужно, просматривая это число от младших разрядов к старшим, заменять встречающиеся единичные значения разрядов нулевыми, а когда встретится нулевой разряд, заменить его единицей и прекратить процесс.

Пример. Прибавим единицу к числу 01101010111. Имеем (разряд, над которым выполняется действие, выделен)

01101010111	1 → 0 и продолжаем
01101010110	1 → 0 и продолжаем
01101010100	1 → 0 и продолжаем
01101010000	0 → 1 и заканчиваем
01101011000	Результат

Упражнение 2.4. Сформулируйте правило вычитания единицы из целого положительного числа.

Вы, наверное, уже встречали в информатике слово "бит". Оно считается сокращением слов binary digit (двоичная цифра) и обозначает (мы об этом будем говорить позднее) минимальную единицу информации — выбор между двумя возможностями, закодированными нулем и единицей *).

Когда мы рассматриваем вектор как 0–1 строку, т. е. последовательность нулей и единиц, или, можно сказать теперь, последовательность битов, то удобно разбивать ее на четверки символов, или *тетрады* (не путать с тетрадями!):

0000011000100111	В человеческой записи
1110010001100000	В машинной записи

*) Отметим, что слово bit, обозначающее малое количество, существовало в английском языке и раньше, еще в средние века. Мне кажется, что цитированное объяснение было придумано, причем известно кем — Клодом Шенноном (см. примечание на с. 86) — для готового слова.

В свою очередь, тетрада может трактоваться как цифра в системе счисления с основанием 16, т.е. в *шестнадцатеричной*, или *гексадецимальной*, системе счисления. Для представления чисел в этой системе нужно выбрать шестнадцать цифр. Общеприняты следующие обозначения: от 0 до 9 используются обычные цифры, а для остальных берутся первые шесть букв латинского алфавита: A, B, C, D, E, F, так что, например, 9 + 1 = A.

Обычная запись	Двоичная человеческая	Двоичная машинная	16-ричная
0	0000	0000	0
1	0001	1000	1
2	0010	0100	2
3	0011	1100	3
4	0100	0010	4
5	0101	1010	5
6	0110	0110	6
7	0111	1110	7
8	1000	0001	8
9	1001	1001	9
10	1010	0101	A
11	1011	1101	B
12	1100	0011	C
13	1101	1011	D
14	1110	0111	E
15	1111	1111	F

По шестнадцатеричной записи числа легко восстановить его запись в двоичной системе счисления, нужно только отчетливо понимать "правила игры" — принятую систему расположения разрядов. Так, наш пример с 1575 запишется в первом случае как `0627`, во втором случае — как `7260`. Впрочем, анализируя запись чисел в компьютере, вы, скорее всего, встретили бы запись: `27 06`.

Дело в том, что обычно память компьютера, которая состоит из битов, объединяется для удобства в более крупные единицы. Наиболее часто встречается деление памяти на восьмерки битов, такая восьмерка называется *байтом*. Байт состоит из двух четверок, полубайтов, по четыре бита в каждом, и, следовательно, представляется двумя 16-ричными цифрами. Нам удобно писать эти цифры по-человечески — старшая левее младшей. На целое число часто отводится два байта,

и старший байт числа имеет больший номер и пишется обычно дальше по-машинному *).

Более крупные единицы измерения памяти — это "машинные" тысячи, миллионы и миллиарды байтов, соответственно, килобайт (1 KB = 2^{10} = 1024 байта), мегабайт (1 MB = 2^{20} = 1 048 576 байтов), гигабайт (1 ГВ = 2^{30} = 1 073 741 824 байта) **), для обозначения триллиона байтов начал уже упоминаться и терабайт. При некоторых специальных надобностях используются промежуточные единицы (*параграфы*, *страницы*) — все такие числа обычно равны степеням 2.

Иногда сочетание арифметических и логических операций приводит к интересным вычислительным эффектам.

Пример. Рассмотрим способ поиска единиц в векторе. Каждый раз будет разыскиваться самый младший разряд с единицей. Вектор, в котором идет поиск, рассматривается одновременно и как набор битов, и как двоичное представление целого числа. Первоначально вычтем из этого числа единицу. Это действие "разменяет" младшую единицу в последовательность единиц нижестоящих разрядов. Так, взяв число a = 0001 0111 0001 1000 (в человеческой записи), после вычитания единицы получим b = 0001 0111 0001 0111. Теперь действие XOR выделит разряды, в которых эти два набора отличаются: c = 0000 0000 0000 1111, после чего достаточно логически умножить этот набор на a, чтобы получить окончательный результат.

(E). Мы перешли уже к следующей интерпретации вектора из нулей и единиц — как *состояния памяти компьютера*. Очень существенно, что в зависимости от обстоятельств содержимое памяти может трактоваться по-разному: кроме чисел это могут быть команды, тексты и многое другое. Трактовка набора битов может быть весьма сложной. Например, в операционной системе MS DOS для запоминания даты создания информационного объекта (файла) использовались два байта — 16 битов. Семь старших битов — это год (считая нулевым годом 1980-й), следующие четыре бита — месяц (с некоторым запа-

*) Термин byte появился в документах фирмы IBM в 1950-е гг., а его "восьмибитовость" была окончательно закреплена в связи с разработкой серии IBM 360.

Для обозначения различного порядка байтов внутри числа в разных компьютерах недавно стали использоваться термины big-endian — старшеконечное расположение и little-endian — младшеконечное расположение. Эти термины взяты из "Путешествий Гулливера" Джонатана Свифта и известны нам как "тупоконечники" и "остроконечники". Помните? Так назывались сторонники разных политических направлений в вопросе о том, с какого конца разбивать яйцо.

**) Числительные "приставки" *кило* и *мега* и обозначения для них используются очень широко, не только для байтов. Обычно из контекста ясно, о какой тысяче идет речь (например, "автомобиль с пробегом 93 К миль"), и делать пояснения не считают нужным.

сом), последние пять — день месяца. В частности, пара байтов 27 06 интерпретируется и как число 1575, и как дата 7 февраля 1983 г. (а почему не января?) *).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
9	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
A	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
B																
C																
D																
E	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я

Для решения в машине многочисленных задач хранения и переработки текстовой информации приходится вводить специальные способы кодирования букв и знаков. В языках, использующих латиницу, завоевал всемирное признание код ASCII — American Standard Code for Information Interchange, первоначально разработанный и стандартизованный в США. В нем фиксированы первые 128 возможностей — верхняя часть таблицы, вторая часть может варьироваться. Вот эта вторая часть и используется для кириллицы, причем, к сожалению, единой общепринятой кодировки нет. Наиболее популярны у специалистов по программированию *альтернативная кодировка*, она же кодировка cp866, которая задается приводимой здесь частично таблицей (строки соответствуют старшему полубайту, а столбцы — младшему), и кодировка cp1251, используемая в Windows. Я не вписал в таблицу две первые строки, в которых содержатся *управляющие* коды, и последнюю строку, в которой символы слишком сложные.

*) Здесь есть запас еще на 100 лет. Перед наступлением 2000 г. большие опасения вызывала "проблема Y2K" — возможность переполнения полей данных, отведеных для года. Некоторые неудобства это переполнение, действительно, вызвало.

Мы только упомянем здесь об активно внедряемом всемирном стандарте двухбайтовой кодировки Unicode *), включающем ASCII как подмножество (с кодами до 007F), и об основанных на нем кодировках.

В Уникоде каждому набору символов выделяется подмножество двухбайтовых кодов, являющееся продолжением некоторого множества префиксов. Например,

Зона	Группа	Код	Знак	Название знака
00	Латинские	0041	A	LATIN CAP. LET. A
		0061	a	LATIN SM. LET. A
037-03F	Греческие и коптские	0394	Δ	GREEK CAP. LET. DELTA
		03C4	τ	GREEK SM. LET. TAU
04	Кириллица	0411	Б	CYRILLIC CAP. LET. BE
		044B	ъ	CYRILLIC SM. LET. YERY
210-214	Буквоподобные	2103	°C	DEGREE CELSIUS
		2118	℘	SCRIPT CAP. P
22	Математические операции	2200	∀	FOR ALL
		2203	∃	THERE EXISTS
		2207	∇	NABLA
		222A	∪	UNION
		22BB	⊻	XOR

Среди промежуточных кодировок, связанных с Уникодом, должна быть упомянута восьмибитовая кодировка переменной длины UTF-8, предназначенная для экономного перехода между Уникодом и другими системами. Она обеспечивает однобайтовое представление для главной части ASCII (текст UTF-8, состоящий только из таких символов, совпадает с ASCII), двухбайтовую кодировку для еще 2K символов, трехбайтовую — для 1M символов и практически бесконечное продолжение.

Кодировка UTF-16 предназначена для дальнейшего расширения возможностей Уникода за счет четырехбайтовых кодов. У нее есть варианты UTF-16BE и UNF-16LE (что бы это значило, по вашему мнению?).

Упражнение 2.5. Попробуйте восстановить кодировку UTF-8 по приведенной выше информации о ней.

*) Этот стандарт уже поддерживают системы Internet и MS Windows, а язык программирования Java использует его как основной. Много важной и свежей информации об Уникоде можно найти в Интернете, в частности на сайте `www.unicode.org`.

Упражнение 2.6. В кодировке UTF-16 используются четырехбайтовые коды, которые состоят из "суррогатных пар", каждая по два байта. Старшая суррогатная пара является продолжением префикса 110110, а нижняя — продолжением префикса 110111. Перепишите эти определения суррогатных пар в шестнадцатеричной системе. Определите количество кодов, получающихся при использовании кодировки UTF-16.

(F). Последовательность нулей и единиц можно еще рассматривать и как *сообщение, передаваемое по каналу связи* (передаются импульсы, каждый из которых принимает одно из двух значений), и как запись результатов экспериментов, каждый из которых может закончиться успехом (1) или неудачей (0).

(G). В некоторых случаях полезно представлять себе эту последовательность как "монотонный" путь на прямоугольной решетке, в котором, например, нули соответствуют шагам направо, а единицы — шагам наверх. На рис. 2.2 представлен путь (1, 0, 0, 0, 0, 0, 1, 1, 0).

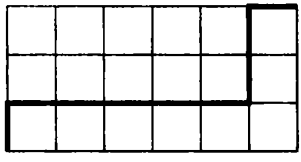


Рис. 2.2. Путь на прямоугольной решетке

(H). Еще одна очень важная трактовка набора из нулей и единиц — это кодирование геометрического изображения. Двухцветная картинка (будем по традиции говорить о черно-белой картинке — черный рисунок на белом фоне) может трактоваться как *растр* *) — совокупность отдельных точек, расставленных на прямоугольной решетке. Сопоставляя черным точкам единицы, а белым — нули, мы и закодируем картинку в виде таблицы из нулей и единиц. Затем таблицу можно "вытянуть" в одну линию, соединяя ее строки или столбцы.

Пример. Рассмотрим решетку 16 × 16 и картинку на ней, изображенные на рис. 2.3 (см. с. 28). Кодируя столбцы числами в диапазоне $0 : 2^{16} - 1$, получаем вектор, который удобнее представить в шестнадцатеричной системе:

```
0000, 0000, 1800, 2C00, 2604, 7208, 4208, 4605,
2883, 3C7C, 0C08, 0030, 0020, 0010, 0000, 0000.
```

*) Растром (от латинского rastrum — грабли) в полиграфии называют прямоугольную сетку, через которую фотографировали тоновое изображение, разбивая его на точки.

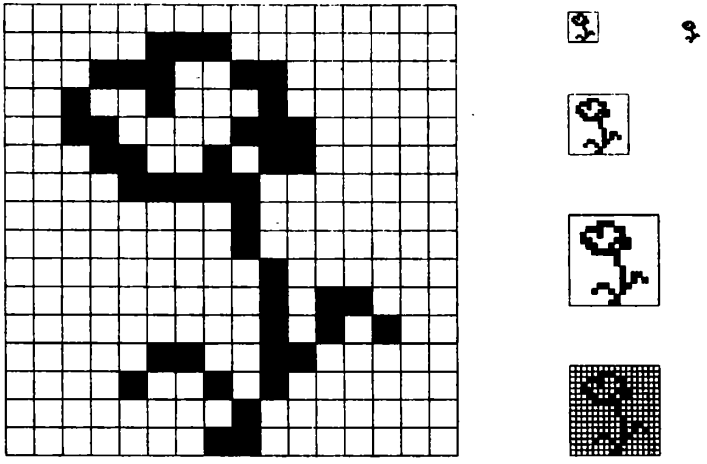


Рис. 2.3. Битовый скромный цветочек

Упражнение 2.7. Проверьте правильность кодировки и установите, какой край картинки выбран для младших разрядов. Нарисуйте сами и закодируйте какой-либо символ 8 × 8.

Упражнение 2.8. Молодая художница Маша Ч. подарила нам картинку

```
0000, 0018, 0124, 01E2, 03C4, 04B8, 0250, 0314,
027C, 5650, 6F4C, 0F92, 00A2, 0154, 01C8, 0000.
```

Что на ней изображено?

Возможность точечного рисования картинок широко используется в вычислительной технике и в обработке изображений. Например, экран дисплея рассматривается как растр, в котором (фиксируем конкретную модель) 960 строк по 1280 точек в каждой строке, т.е. в общей сложности 1 228 800 точек. Для сохранения такого набора нужно (по 8 точек в одном байте) 150 Кбайтов.

Символы, изображаемые на дисплее или на печатающем устройстве, также имеют растровое представление. При этом растр, в зависимости от конкретного устройства, трактуется как последовательность строк (дисплей и символы на нем) или как последовательность столбцов (матричный принтер).

В некоторых случаях оказывается выгодно укрупнять растр — рассматривать группу точек, например 8 × 8, как одну крупную точку и задавать ее *яркость* как долю белых точек среди всех. Принима-

ющая 65 различных значений яркость может быть задана проще, чем произвольное заполнение 64 полей (6 или 7 битов вместо 64), а для заполнения, обеспечивающего нужную яркость, выбирается один из стандартных рисунков. Такие схемы задания рисунка дают изображение, более устойчивое к искажениям (типа *муаров* или сеток).

(I). Многократное использование перечисленных элементарных конструкций, превращение их в стандартные "строительные блоки" может привести к довольно сложным структурам данных (это естественно, ведь все информационные структуры, создаваемые в компьютерах, состоят из нулей и единиц). Рассмотрим в качестве примера структуру данных в видеопамяти EGA при использовании цветового графического режима *). Растр такого старого дисплея составлял 224 000 точек. В цветовом режиме каждой из этих точек можно сопоставить один из 16 цветов, что, очевидно, требует 4 бита на каждую точку.

Эти биты в системе EGA было принято располагать не все вместе, а в отдельных областях памяти по 224 000 битов в каждой (на самом деле, конечно, по 256 килобитов = 32 килобайта). Эти области традиционно называются *цветовыми плоскостями*. Имеется возможность записи информации одновременно во все цветовые плоскости. Однако, поскольку адресуемой единицей информации в компьютере является байт, запись естественно вести сразу в восемь смежных битов.

Здесь мы можем применить наши знания о 0–1 векторах и сказать о деталях. Перед записью одного байта единой командой, специальными командами устанавливается "картинка", которая заносится в видеопамять, *маска изображения* и *способ записи* — логическая операция, выполняемая над старым состоянием байта и маской для получения нового состояния.

Более поздние системы (VGA и другие), в которых число допустимых цветов значительно больше, устроены аналогично, хотя проще — в них на задание каждого цвета отводится по 2 байта.

(J). *Отрицательные числа и числа с плавающей точкой.* Представление целых чисел любого знака требует новых решений. Из многих мыслимых вариантов устойчиво закрепились два: кодирование со смещением и дополнительный до двух код.

При *кодировании со смещением* к каждому числу r прибавляется константа, выбранная так, чтобы при любом r сумма $r + D$ была неотрицательной; получающаяся сумма и кодируется вместо r. Остается понять, какой должна быть константа, но на этот вопрос ответить несложно. Действительно, если для представления чисел выбрано k двоичных разрядов, то можно закодировать 2^k различных

*) И приводимые здесь параметры дисплея, и адаптер EGA — Enhanced Graphical Adapter — безнадежно устарели и не используются. Но этот адаптер удобнее для нашей иллюстрации, чем более простые современные.

чисел. Естественно поделить эти возможности поровну между положительными и отрицательными числами. Правда, еще есть нуль. Припишем его к положительным числам — так удобнее. Тогда константа смещения равна $D = 2^{k-1}$. Например, если $k = 5$, то $D = 2^4 = 16$ и $code(0) = 16$, $code(11) = 27$, $code(-7) = 9$, а $code(17)$ и $code(-19)$ не определены, так как эти числа выходят за границы интервала определения кода $-16 : 15$.

Дополнительный код оперирует с остатками от деления чисел на 2^k, при этом остаток r, лежащий в интервале $2^{k-1} : 2^k - 1$, считается отрицательным числом $r - 2^{k-1}$. Таким образом, для тех же примеров $code(0) = 0$, $code(11) = 11$, $code(-7) = 32 - 7 = 25$. В основной части арифметических расчетов используется дополнительный код, так как расчеты с этой записью проводятся легче и естественнее; код со смещением применяется только в специальных случаях.

Упражнение 2.9. Чем различаются правила сложения чисел, заданных в обеих кодировках, и правила их сравнения?

Упражнение 2.10. Определите правила перехода от одной кодировки к другой.

Специальный случай, когда требуется кодирование со смещением, предоставляет нам формат, применяемый при задании чисел с плавающей точкой. Этот формат, как вы, по-видимому, хорошо знаете, используется для вычислений с дробными числами и чисел, изменяющихся в больших диапазонах. В нем число x представляется в виде $x = s \cdot \alpha \cdot 2^{\beta}$, где $s = \pm 1$, α — мантисса, неотрицательное число из диапазона, разного для разных случаев, которое с некоторой условностью можно считать целым, а β — порядок, целое число. Двоичный вектор, отведенный для записи числа, разбивается на три поля: для мантиссы, для порядка и для знака числа. Знак числа имеет два значения, и ему достаточно одного разряда. Мантисса записывается в дополнительном коде, а вот для записи порядка принято использовать код со смещением (поле порядка принято называть характеристикой).

(К). *Штриховые коды.* Во многих практических информационных системах — в магазинах, в почтовой службе и в других местах, где требуется быстро и просто считывать в компьютер небольшую числовую или текстовую информацию, — используют так называемые *штриховые коды* (barcodes). Мы встречаем их на упаковках товаров, на книгах. Имеется много разновидностей этих кодов; пример одной из них, так называемого *кода 3 из 9*, представлен на рис. 2.4. Каждый символ кодируется 9 полосками, поочередно черными и белыми, в конце добавляется еще одна белая полоска. Три полоски из основных

* B A R C O D E 3 9 *

Рис. 2.4. Текст, записанный штриховым кодом

девяти имеют увеличенную ширину. Так можно закодировать $9 \cdot 8 \cdot 7/6 = 84$ разных символа — вполне достаточно для всего латинского алфавита (хотя на самом деле кодируются только прописные буквы), цифр и некоторых знаков. Сопоставляя каждой широкой полоске 1, а узкой полоске 0, получаем 10-битовые последовательности, в которых последний бит всегда равен 0, так что существенными для информации оказываются 9 битов. Посмотрим, как трактуются эти биты.

Каждой кодовой комбинации соответствует трехэлементное подмножество, которое задается вектором из трех единиц и шести нулей. Для цифр и нулей выбраны такие коды, в которых широких полосок одна белая и две черные. Удобно определять отдельно векторы черных и белых полосок. Для цифр берется вектор белых полосок (0, 1, 0, 0), а для самих цифр берутся подряд векторы $v_1 = (1, 0, 0, 0)$, $v_2 = (0, 1, 0, 0)$, ..., $v_0 = (0, 0, 1, 1)$ и т.д., кроме тех, где больше двух единиц. Пятая компонента вектора выбирается так, чтобы дополнить число единиц до 2. Таким образом, для 3 мы имеем (1, 1, 0, 0, 0), а для 7 — (0, 0, 0, 1, 1).

Буквы A–J кодируются подобно цифрам, но с вектором белых полосок (0, 0, 1, 0), буквы K–T — с вектором (0, 0, 0, 1), буквы U–Z, тире, запятая, пробел и звездочка — с вектором (1, 0, 0, 0).

Теперь можно проверить пример на рис. 2.4. Составим таблицу, в которой первый столбец заполнен символами, далее следуют 9 знаков, описывающих ширину полосок (У — узкая, Ш — широкая), затем по этим знакам сформированы коды черных и белых полосок, и наконец, эти коды переведены в условные обозначения группы и номера в группе. В качестве представителя

Знак	ЧБЧБЧБЧБЧ	Код черных	Код белых	Группа	Номер
*	УШУУШУШУУ	00110	1000	U	0
B	УУШУУШУУШ	01001	0010	A	2
A	ШУУУУШУУШ	10001	0010	A	1
R	ШУУУУУШУШ	10010	0001	K	8
C	ШУШУУШУУУ	11000	0010	A	3
O	ШУУУШУУШУ	10100	0001	K	5
D	УУУУШШУУШ	00101	0010	A	4
E	ШУУУШШУУУ	10100	0010	A	5
3	ШУШШУУУУУ	11000	0100	1	3
9	УУШШУУШУУ	01010	0100	1	9

в каждой группе выбран символ с кодом 1. Напомним, что пропуск в кодировке чисел 7 и 11, имеющих слишком много единиц, сдвинул нумерацию, и это повлияло на представление кода черных полосок у символов R и 9.

В связи с развитием так называемых multimedia — всевозможных периферийных аудио- и видеоустройств — появились многочисленные новые форматы, в частности для звуковых файлов и кинофайлов (например, формат AVI). Имеется много специальных форматов графического вывода (например, метафайлы Windows), которых просто не перечесть. Богатую информацию по этому вопросу вы можете найти в любом из многочисленных справочников по графическим форматам.

Еще об одной группе форматов нужно сказать особо — о форматах сжатия. Мы вернемся к этой теме несколько позднее.

2.2. Перебор 0–1 векторов

Сформулируем первые вопросы, которые ставятся в комбинаторике при изучении того или иного множества:

1) Сколько в этом множестве элементов?

2) Как перебрать все элементы, т. е. как организовать вычислительный процесс, на каждом шаге которого будет формироваться новый, не встречавшийся ранее, элемент этого множества?

3) Как перенумеровать элементы множества, т. е. как, зная количество элементов множества n, сформировать взаимнооднозначное соответствие между элементами множества и числами из $1 : n$ (иногда числа из промежутка $0 : (n - 1)$ брать удобнее)?

В этом параграфе мы отвечаем на все три вопроса для множества $\mathbf{B}^m$ всех наборов из m битов, когда каждый из них может быть нулем и единицей.

Ответ на первый вопрос уже известен: $|\mathbf{B}^m| = 2^m$.

Ответ на второй вопрос легко получается из взаимнооднозначного соответствия между числами из $0 : 2^m - 1$ и наборами 0–1 векторов: чтобы перебрать все наборы, нужно взять число 0 и его представление $(0, \ldots, 0)$, а дальше прибавлять к текущему значению числа по единице, имитируя прибавление на текущем наборе, до тех пор, пока не получится набор $(1, \ldots, 1)$.

Да и с третьим вопросом нет сложностей — нумерация уже готова (начинать нумерацию от 0, а не от 1 здесь действительно удобнее). Переход от набора к номеру и от номера к набору — это, соответ-

ственно, переход от двоичного представления числа к его значению и от значения к двоичному представлению.

Теперь немного усложним свою задачу. При таком просмотре 0–1 наборов очередное прибавление единицы может вызвать сильное изменение набора (например, 010111 → 011000). Иногда желательно переходить каждый раз от текущего набора к похожему на него; например, в случае 0–1 наборов можно потребовать, чтобы на каждом шаге менялось значение только одной компоненты.

Не тратя силы на возможность построения такого перебора, прямо предложим его принципиальную схему. Она основана на идее рекурсии. Чтобы перебрать все наборы длины m, зафиксируем нулевое значение у m-й компоненты и переберем все наборы длины $m-1$ для оставшихся компонент. Перебрав их, сменим значение m-й компоненты на 1 (на этом шаге, номер которого 2^{m-1}, меняется именно эта единственная компонента) и снова переберем наборы длины $m-1$, но уже в обратном порядке ("зеркально отразим" таблицу). Естественно, что ту же схему мы применим и для наборов меньшей длины.

Таблица на рис. 2.5 показывает перебор для наборов длины 4. В столбце it записан номер итерации, а в столбце k_{it} — номер обновляемой компоненты. Горизонтальные линии изображают "зеркала".

x_4	x_3	x_2	x_1	it	k_{it}
0	0	0	0	0	1
0	0	0	1	1	2
0	0	1	1	2	1
0	0	1	0	3	3
0	1	1	0	4	1
0	1	1	1	5	2
0	1	0	1	6	1
0	1	0	0	7	4
1	1	0	0	8	1
1	1	0	1	9	2
1	1	1	1	10	1
1	1	1	0	11	3
1	0	1	0	12	1
1	0	1	1	13	2
1	0	0	1	14	1
1	0	0	0	15	–

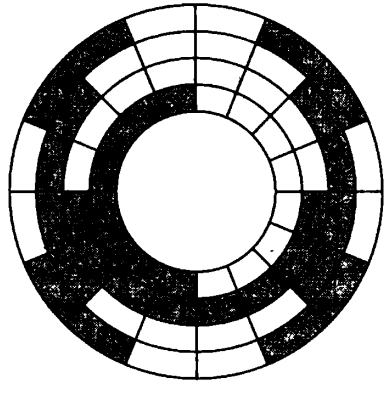


Рис. 2.5. Коды Грея и их использование для маркировки круга, разбитого на 16 частей

Алгоритм легко программируется с использованием рекуррентных процедур, однако для дальнейшего полезен другой подход. Выше отмечалось, что смена значения m-й компоненты происходит на итерации 2^{m-1}. Но это же верно для любого k: значение k-й компоненты изменяется на каждой 2^{k-1} итерации, на которой не меняется значение компоненты с бо́льшим номером. Вспомним, что именно в k-й позиции появляется единица на каждой 2^{k-1} итерации в предыдущей схеме перебора 0–1 векторов, и тоже в том и только том случае, если единица не появляется в старшей позиции.

Это соображение позволяет предложить следующий вычислительный процесс.

Создать два набора x и y, каждый из m битов. Первоначально $x = y = (0, \ldots, 0)$.

На каждой итерации прибавить 1 к числу, для которого x является двоичным разложением, и фиксировать позицию k, где нуль сменяется единицей.

Изменить k-ю компоненту в наборе y: $y_k := 1 - y_k$.

Выдать набор y как результат итерации.

Здесь поучительно использование набора x, единственное назначение которого — дать простой способ получения позиции k. Такое искусственное расширение состояния вычислительного процесса, упрощающее вычисления, с нашей точки зрения, одно из самых главных умений, которыми должен владеть хороший конструктор алгоритмов.

Описанная здесь последовательность 0–1 векторов называется *кодом Грея*, так как Ф. Грей *) в 1953 г. запатентовал в США ее использование для кодирования двоичных чисел (в таблице — это значение it, соответствующее коду). Запатентованная Греем идея применялась к определению угла поворота вращающегося круга по цветам специальных дорожек (см. рис. 2.5) на уровне контрольного радиуса. Выбор цвета может быть неоднозначным на границе двух цветов, но в случае, если на каждом радиусе меняется цвет не больше чем у одной полоски, эта неоднозначность не страшна.

Упражнение 2.11. Для кода Грея важно уметь находить число по набору цветов дорожек. Постройте алгоритм, переводящий $y[1 : m]$ в соответствующее ему значение it.

*) Frank Gray. В очень интересной статье Хита [66] указывается, что задолго до Грея этот код использовал французский инженер Эмиль Бодо (Emile Baudot, 1845–1903), изобретатель печатающего телеграфа.

Упражнение 2.12. Кроме кода Грея еще используется и "цепной код" (chain code), в котором 0–1 векторы длины n перебираются в таком порядке, что все n последовательностей изменений i-го бита получаются друг из друга циклическим сдвигом на один шаг, как показано на рис. 2.6.

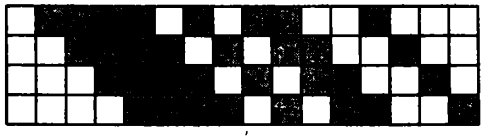


Рис. 2.6. Цепной код для $n = 4$. Код задается одной из горизонтальных строк, остальные получаются из нее циклическими сдвигами

Докажите существование цепного кода для произвольного n и разработайте алгоритм его построения.

2.3. Перебор элементов прямого произведения множеств

Ответ на поставленные в предыдущем параграфе три вопроса для другого встречавшегося нам множества — прямого произведения $M(1:k) = M_1 \times M_2 \times \cdots \times M_k$ — получается на основе тех же принципов. Мы уже говорили, что число элементов в этом множестве равно

$$|M_1 \times M_2 \times \cdots \times M_k| = \prod_{i \in 1:k} m_i,$$

где $m_i = |M_i|$ — число элементов множества M_i. Будем считать, что каждое M_i состоит из целых чисел от 0 до $m_i - 1$ (можно просто перенумеровать элементы и отождествить каждый элемент с его номером). Тогда каждый элемент $M(1:k)$ — это последовательность неотрицательных чисел $r_1, \ldots, r_k$, причем $r_i < m_i$.

Можно ввести специальную *систему счисления с переменным основанием*, т.е. с основанием, которое для разных разрядов может быть различным. Такая система используется при измерении времени: часов в сутках — 24, минут в часе — 60, секунд в минуте — тоже 60, или даже подробнее: 24 часа, 6 десятков минут в часе, 10 минут в десятке и так же с секундами. 24 часа тоже можно разбить на две "цифры": два *времени суток* и по 12 часов в каждой из этих половин.

Еще в школе нас учили и как прибавлять одну секунду с учетом переполнения секундных, минутных и часовых разрядов, и как переводить часы-минуты-секунды в секунды и обратно. Так что мы сразу скажем, что набор (18, 25, 13), представляющий 13 часов 25 минут 18 секунд, соответствует

$$((13 \times 60) + 25) \times 60 + 18 = 48318$$

секундам (т.е. это набор номер 48318, считая от 0), а набор номер 11637 легко вычисляется по номеру: 11637 с = 193 мин 57 с = 3 ч 13 мин 57 с, значит, речь идет о наборе (57, 13, 3).

Таких систем счисления с переменным основанием много среди старинных мер, например меры длины: 1 ярд = 3 фута, 1 фут = 12 дюймов, 1 дюйм = 12 линий, 1 линия = 6 пунктов*). Подсчитайте сами, сколько пунктов, например, в 7 футах 7 дюймах 3 линиях.

Тем, кто связан с программированием, прямые произведения множеств могут встретиться в некоторых способах представления данных. Среди них один из самых распространенных — это массивы, прямоугольные таблицы (размерности один, два, три и иногда даже больше). Таблица размерности 2 нам встречалась в предыдущем параграфе как растр экрана.

Общая формула перехода от элемента $(r_1, \ldots, r_k)$ к номеру этого элемента, очевидно, такова:

$$\mathrm{num}(r_1, \ldots, r_k) = \sum_{i=1}^{k} r_i \times \left(\prod_{j=1}^{i-1} m_j \right).$$

Обратите внимание на то, что num — *линейная функция* от чисел r_i (хочется назвать их индексами — помня об использовании этой формулы для вычисления адреса элемента многомерного массива).

В ближайшем же параграфе нам понадобится случай, когда каждое из множеств M_i представляет собой множество чисел от 0 до $i - 1$. Обозначим произведение k таких множеств через T_k. Множество T_k содержит $\prod_{i \in 1:k} i = k!$ элементов**).

*) Эти единицы измерения устойчиво держатся в некоторых областях техники. Традиционно в дюймах выражается линейка размеров компьютерных устройств. В пунктах измеряется размер типографского шрифта, например шрифт 10 пунктов. Это число задает высоту не буквы, а литеры, бруска металла, на котором литера изображается в традиционной, идущей от Гутенберга, системе набора.

**) Читается: k-факториал, от слова factor (множитель). Иногда [3] система счисления с таким выбором переменного базиса называется факториальной.

Упражнение 2.13. Вспомним стихи К. Чуковского:

Шел Кондрат в Ленинград,
А навстречу — 12 ребят.
У каждого по 3 лукошка,
В каждом лукошке — кошка,
У каждой кошки — 12 котят,
У каждого котенка в зубах по 4 мышонка.

Перенумеруйте мышат, начиная от 1. Укажите, где находится мышонок номер 191.

2.4. Перестановки

2.4.1. Определение и перебор перестановок

В этом и следующем параграфах речь идет о наборах или расположениях различных элементов множества. Не умаляя общности, можно говорить о наборе различных чисел, так как они могут быть номерами элементов любого множества.

Перестановкой *) из k элементов называется упорядоченный набор из k различных чисел из диапазона $1:k$. (Иными словами, это набор, получающийся из набора $(1, \ldots, k)$ расстановкой элементов по имеющимся k местам.) Обозначим множество всех перестановок из k элементов через $\mathbf{P}_k$ и попробуем ответить на те же три вопроса, что и раньше, применительно к этому множеству.

Хотя было бы легко доказать непосредственно, что $|\mathbf{P}_k| = k!$, удобнее ответить сразу на все три вопроса, построив взаимнооднозначное соответствие между $\mathbf{P}_k$ и $\mathbf{T}_k$. Действительно, пусть в нашем распоряжении будет отображение $f : \mathbf{P}_k \to \mathbf{T}_k$ и обратное отображение $f^{-1} : \mathbf{T}_k \to \mathbf{P}_k$. Уже из существования таких отображений следует, что число элементов в этих множествах одинаково. Далее, перебор элементов $t_1, t_2, \ldots$ порождает последовательность $f^{-1}(t_1), f^{-1}(t_2), \ldots$ Наконец, если потребуется по элементу $p_k \in \mathbf{P}_k$ найти его номер, то это будет номер элемента $f(p_k) \in \mathbf{T}_k$.

Построим взаимнооднозначное соответствие между $\mathbf{P}_k$ и $\mathbf{T}_k$. Возьмем перестановку $(r_1, \ldots, r_k)$ и сопоставим ей элемент $(t_1, \ldots, t_k)$ следующим образом: для любого $i \in 1:k$ найдем число значений, меньших r_i среди $r_{i+1}, \ldots, r_k$, — это число мы и примем в качестве t_i.

В соответствии с таким определением чисел t_i, в множестве $\mathbf{T}_k$ будет естественно сделать значения m_i не возрастающими, как было написано раньше, а убывающими до 1.

*) Permutation.

Таким образом, например, для перестановки $(4, 8, 1, 5, 7, 2, 3, 6)$ мы получим

i	1	2	3	4	5	6	7	8
r_i	4	8	1	5	7	2	3	6
t_i	3	6	0	2	3	0	0	0
m_i	8	7	6	5	4	3	2	1

По записи $(t_1, \ldots, t_k)$ легко восстановить исходную перестановку (построить элемент обратного отображения). Для этого, меняя i от 1 до k, нужно просто помнить множество значений S_i, которые могут быть в перестановке на i-м месте. Так, $S_1 = 1 : 8$ и $t_1 = 3$ означает, что $r_1 = 4$, поскольку именно ему предшествуют в этом множестве три элемента. Получаем $S_2 = 1 : 3 \cup 5 : 8$ и $t_2 = 6$, что дает $r_2 = 8$ и т. д.

Если использовать это отображение при переборе, то перестановки будут перебираться в *лексикографическом порядке*. Это значит, что перестановка $(r_1, \ldots, r_k)$ предшествует перестановке $(R_1, \ldots, R_k)$ в том и только том случае, если начала этих перестановок совпадают до какого-то индекса d, а дальше $r_d < R_d$. Мы будем использовать лексикографическое сравнение и в дальнейшем, а сейчас отметим такой факт.

Лемма. Какова бы ни была позиционная система, для того чтобы число A было меньше числа B, необходимо и достаточно, чтобы запись A в этой системе лексикографически предшествовала записи B.

Доказательство оставляется читателю.

Очевидно, что если факториальная запись $(t_1, \ldots, t_k)$ лексикографически предшествует записи $(T_1, \ldots, T_k)$, то этот порядок верен и для соответствующих перестановок. А из этого факта уже прямо следует наше утверждение о лексикографическом порядке перебираемых перестановок.

Функция готова, настала очередь перебора. Просматривая все элементы T_k в "естественном" порядке, по каждому t_k строим p_k. Проследите, как в порядке возрастания номеров перебираются элементы P_4.

Из того, что перестановки перебираются в лексикографическом порядке, можно вывести простое правило получения следующей перестановки из данной. Действительно, лексикографический порядок предполагает, что любое начало (префикс) перестановки нужно по возможности дольше сохранять неизменным. Имея фиксированный префикс длины s в перестановке длины k, мы должны перебрать все $(k - s)!$

num	t	p	num	t	p	num	t	p	num	t	p
0	0000	1234	6	1000	2134	12	2000	3124	18	3000	4123
1	0010	1243	7	1010	2143	13	2010	3142	19	3010	4132
2	0100	1324	8	1100	2314	14	2100	3214	20	3100	4213
3	0110	1342	9	1110	2341	15	2110	3241	21	3110	4231
4	0200	1423	10	1200	2413	16	2200	3412	22	3200	4312
5	0210	1432	11	1210	2431	17	2210	3421	23	3210	4321

возможных окончаний (суффиксов) и только после этого менять что-то в префиксе. Но это же рассуждение относится и к префиксам длины $s-1$, и, конечно, любой меньшей длины. Получается очень простое правило:

1. В заданной перестановке $(r_1, \ldots, r_k)$ найти ее наибольший суффикс $(r_t, \ldots, r_k)$, в котором элементы расположены по убыванию $r_t > \cdots > r_k$ (максимальность суффикса означает, что $r_{t-1} < r_t$).
2. Выбрать в $(r_t, \ldots, r_k)$ элемент, следующий по величине после r_{t-1}, и поставить его в перестановку на место $t-1$. Оставшиеся элементы, включая r_{t-1}, расположить за ним в порядке возрастания.

Для примера рассмотрим несколько шагов перебора перестановок из 9 элементов. "Убывающий конец" каждой перестановки выделен жирным шрифтом.

3 4 2 1 **7 8 9 5 6**
3 4 2 1 **7 8 9 6 5**
3 4 2 1 **7 9 5 6 8**
3 4 2 1 **7 9 5 8 6**
3 4 2 1 **7 9 6 5 8**
3 4 2 1 **7 9 6 8 5**
3 4 2 1 **7 9 8 5 6**
3 4 2 1 **7 9 8 6 5**
3 4 2 1 **8 5 6 7 9**

Упражнение 2.14. Напишите перестановку из 12 элементов, следующую после перестановки (11, 3, 5, 9, 1, 4, 7, 6, 8, 12, 10, 2).

Упражнение 2.15. Не выполняя перебора, запишите перестановку из 8 элементов, стоящую при переборе перестановок в лексикографическом порядке через 219 шагов после перестановки (3, 5, 1, 4, 7, 6, 2, 8).

Из примеров видно, что от итерации к итерации очередная перестановка изменяется сильно. Хотелось бы предложить процесс перебора, использующий небольшие изменения текущей перестановки, например одну *транспозицию* (перемену мест) двух соседних элементов.

Опять-таки, рекурсивные соображения показывают, что это сделать очень просто. При фиксированной перестановке из $k-1$ элемента можно перебрать все k вариантов добавления к этой перестановке элемента k, и этот перебор можно осуществить, передвигая элемент k каждый раз на соседнее место. Например,

$$3652147 \rightarrow 3652174 \rightarrow 3652714 \rightarrow 3657214$$

и т. д. На фоне такого перебора позиций k-го элемента должны проводиться переборы перестановок меньшего порядка, к которым применяется тот же принцип, т. е., например, в нашем случае после получения набора 7365214 требуется сдвинуть влево или вправо элемент 6. Так влево или вправо? Я не знаю. Нужен механизм для подсказки на каждом шаге, какой элемент двигать и в какую сторону.

Что же должен делать этот механизм? Каждые $k-1$ итераций он должен давать команду на сдвиг k-го элемента, а затем менять направление движения его на противоположное и давать команду на сдвиг элемента с меньшим номером; для этих выделенных итераций нужно делать то же самое: на $k-2$ из них двигать $(k-1)$-й элемент, а на $(k-1)$-й итерации сменить ему направление движения, и т. д. Датчиком ритмов может, очевидно, служить система перечисления элементов T_k, если в ней принять t_k в качестве младшего разряда, а затем взять t_{k-1} и т. д.

Перебор перестановок

Состояние вычислительного процесса. Кроме рабочей перестановки r и ее номера в факториальной системе t (младший разряд — последний) потребуется иметь массив d, задающий текущие направления движения всех элементов. Удобно еще иметь массив, сопоставляющий каждому элементу i то место p_i, на котором i стоит в перестановке r.

Начальное состояние.

$$r = (1, 2, \ldots, k), \qquad p = (1, 2, \ldots, k),$$
$$t = (0, 0, \ldots, 0), \qquad d = (-1, -1, \ldots, -1).$$

Стандартный шаг. Увеличить вектор t на 1. При этом несколько младших разрядов получат нулевые значения, а в одном из разрядов, j-м, значение увеличится на 1 (при $j = 1$ процесс заканчивается).

Сменить направление движения всех элементов младше j-го, т. е. положить $d_i := -d_i$ для $i > j$. Поменять местами j-й элемент и соседний с ним (если $d_j = -1$ — левый, иначе — правый). Как это сделать?

Элемент j стоит на месте $s = p_j$. Это значит, что $r_s = j$. Соседнее место — это $s' = p_j + d_j$. На нем стоит какой-то элемент $j' = r_{s'}$. Поменять местами в перестановке элементы j и j' означает поменять местами содержимое p_j и $p_{j'}$, а также r_s и $r_{s'}$.

Разберитесь теперь в работе этого механизма на примере, приведенном в таблице.

i	t	d	p	r	j	Комментарий
1	0000	− − −−	1234	1234	–	Здесь j не определено
2	0001	− − −−	1243	1243	4	Начинается движение эл. 4
3	0002	− − −−	1342	1423	4	
4	0003	− − −−	2341	4123	4	
5	0010	− − −+	2431	4132	3	Шаг эл. 3, у 4 — смена напр.
6	0011	− − −+	1432	1432	4	
7	0012	− − −+	1423	1342	4	
8	0013	− − −+	1324	1324	4	
9	0020	− − −−	2314	3124	3	Второй шаг эл. 3
10	0021	− − −−	2413	3142	4	
11	0022	− − −−	3412	3412	4	
12	0023	− − −−	3421	4312	4	
13	0100	− − ++	4321	4321	2	Шаг эл. 2. Смена напр. у 3 и 4
14	0101	− − ++	4312	3421	4	
15	0102	− − ++	4213	3241	4	
16	0103	− − ++	3214	3214	4	
17	0110	− − +−	3124	2314	3	
18	0111	− − +−	4123	2341	4	
19	0112	− − +−	4132	2431	4	
20	0113	− − +−	4231	4231	4	
21	0120	− − ++	3241	4213	3	
22	0121	− − ++	3124	2413	4	
23	0122	− − ++	2143	2143	4	
24	0123	− − ++	2134	2134	4	
25	1000	− + −−	—	—	1	Остановка процесса

Упражнение 2.16. Как обойтись в этом алгоритме без массива t?

Многие методы перебора перестановок и других комбинаторных объектов рассмотрены в книге В. Липского [33].

2.4.2. Экстремальные задачи, связанные с перестановками

Перестановка — объект, очень часто встречающийся в самых разных областях математики. Вам предстоит иметь дело с перестановками в алгебре в связи с теорией определителей и теорией групп. Здесь же мы рассмотрим некоторые неалгебраические использования понятия перестановки. Это прежде всего *экстремальные задачи*, в которых на множестве перестановок данного размера определена некоторая функция, называемая *целевой функцией*, и требуется найти перестановку, на которой эта функция принимает экстремальное значение (максимальное или минимальное, в зависимости от смысла задачи). Первая из задач несколько раз встретится нам в дальнейшем.

Задача о минимуме скалярного произведения. Пусть заданы m чисел $x_1, x_2, \ldots, x_m$ и еще m чисел $y_1, y_2, \ldots, y_m$. Составим пары (x, y), включив каждое x_i и каждое y_j ровно в одну пару. Затем перемножим числа каждой пары и сложим получившиеся произведения. Требуется найти такое разбиение чисел на пары, при котором значение получившейся суммы будет наименьшим. ◻

Теорема. Наименьшее значение суммы попарных произведений достигается при сопоставлении возрастающей последовательности x_i убывающей последовательности y_i.

Доказательство. Мы просто покажем, что если найдутся две пары чисел (x_i, y_i) и (x_j, y_j), такие что $x_i < x_j$ и $y_i < y_j$, то значение суммы попарных произведений можно уменьшить, заменив эти две пары парами (x_i, y_j) и (x_j, y_i). Действительно, так как

$$(x_j - x_i)(y_j - y_i) > 0,$$

то, раскрывая скобки, получим после элементарных преобразований

$$x_i \cdot y_i + x_j \cdot y_j > x_i \cdot y_j + x_j \cdot y_i.$$

Поскольку число возможных расположений равно $m!$, т.е. конечное число, то начиная с любого расположения за конечное число шагов мы закончим процесс улучшений на расположении, которое дальше улучшить невозможно. На нем и достигается минимум. □

Задача о максимальной возрастающей подпоследовательности. Допустим, что задана перестановка $R = (r_1, \ldots, r_n)$. Найти такую после-

довательность $i_1, \ldots, i_k$, для которой

$$1 \le i_1 < i_2 < \cdots < i_k \le n,$$

$$r_{i_1} < r_{i_2} < \cdots < r_{i_k}$$

и длина k этой последовательности максимальна. ⊠

Мы сначала опишем алгоритм для нахождения искомой последовательности, а затем докажем, что длина получающейся последовательности максимальна. Будет строиться разбиение множества $1 : n$ на подмножества $I_1, \ldots, I_s$, обладающие свойством "убывания". Подмножество I_t обладает этим свойством, если для всех пар $i, j \in I_t$, $i < j$, выполняется неравенство $r_i > r_j$. Иными словами, последовательность $(r_1, \ldots, r_n)$ разбивается на s убывающих подпоследовательностей.

По этому разбиению будет построена возрастающая последовательность длины s, которая будет содержать по одному элементу. Покажем, что она будет искомой, т.е. что s будет максимально возможным значением k.

Действительно, длина возрастающей подпоследовательности k не может быть больше s: больше одного элемента из любого I_t эта подпоследовательность содержать не может. Разбиение перестановки на убывающие подпоследовательности начнем с простого примера, в котором в первой строчке записана сама перестановка, а ниже — получающиеся подпоследовательности:

```
19  7 18 20 11  3  6  4  8  2 14 10 17  5 16  1 15 13  9 12
19  7           3           2                 1
      18    11       6  4
         20                  8                 5
                                  14 10                 9
                                        17    16    15 13    12
```

Мы стремимся "вести себя экономно" — записываем каждое очередное число в ту из последовательностей, в которую его можно записать, не нарушая порядка и выбирая последовательность с наименьшим последним числом.

Если порядок нарушится в любой из имеющихся подпоследовательностей, мы открываем новую. Так, число 19 открывает новую последовательность, 7 приписывается к 19, 18 и 20 открывают новые, 11 можно приписать к 18 или 20, но к 18 экономнее, и т.д. Получилось пять подпоследовательностей.

Возьмем теперь любое число в последней строчке, например последнее. Почему 12 записано в последнюю строчку? Потому что в предыдущей строчке уже было записано меньшее число, а именно 9. Оно было записано в четвертую строчку из-за значения 4 во второй строчке. А значению 4 предшествует 3 в первой строчке. Вот мы и построили возрастающую последовательность: 3, 4, 5, 9, 12.

Так будет и в общем случае: у каждого числа в каждой строчке, кроме первой, имеется предшественник, "не пустивший его" в предыдущую строчку. Предшественник меньше и стоит в перестановке раньше. Проход от последней строчки по цепочке предшественников и дает нам искомую возрастающую последовательность. По сделанному замечанию ее длина максимальна.

Упражнение 2.17. (Из школьных олимпиад по математике.) Докажите, что в любой перестановке из 2003 элементов найдется монотонная подпоследовательность длины 45.

Упражнение 2.18. Как нужно модифицировать наш алгоритм, если мы ищем максимальную возрастающую подпоследовательность не в перестановке, а в произвольной подпоследовательности длины n?

Третья экстремальная задача внешне очень проста. Она имеет некоторое отношение к вычислительным задачам молекулярной биологии.

Задача о минимальном числе инвертирований. Назовем *инвертированием* в перестановке $R = (r_1, \ldots, r_n)$ операцию, которая заменяет "сплошной кусок" перестановки $(r_k, \ldots, r_l)$ $(1 \le k < l \le n)$ теми же элементами, записанными в обратном порядке. Например, инвертированием выполняется преобразование

$$(3, 6, 8, 1, 2, 4, 5, 7) \to (3, 4, 2, 1, 8, 6, 5, 7).$$

Требуется выбрать последовательность инвертирований, переводящую данную перестановку в тождественную $(1, 2, \ldots, n)$ за минимальное число шагов. ☒

Например, перестановку (3, 6, 8, 1, 2, 4, 5, 7) можно перевести в тождественную за пять инвертирований следующим образом:

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

Мы добавили в этой таблице один лишний столбец слева и два справа. Зачем добавили — сейчас будет объяснено.

Алгоритм, достаточно быстро находящий в этой задаче точный минимум, не известен (подчеркнем *достаточно быстро*, так как полный перебор всех возможностей в принципе осуществим, но нас не устраивает). Задача интересна тем, что в ней удается найти оценку снизу для минимального числа инвертирований $d(R)$ и построить алгоритм, по которому число инвертирований не превосходит $4d(R)$. В этом мы сейчас и убедимся.

Добавим к перестановке R элемент 0 слева и элемент $n+1$ справа. Эти элементы переставляться не будут, они полезны как "надежные границы" нашей перестановки. Обозначим через $s(R)$ количество точек разрыва — пар соседних элементов, отличающихся больше чем на 1. В последнем столбце таблицы и выписаны значения $s(R)$. Все остальное записывается в следующих легко доказываемых утверждениях.

Лемма 1. Никакое инвертирование не может уменьшить $s(R)$ более чем на 2, и следовательно, $d(R) \geq s(R)/2$.

Назовем *убывающей полосой* набор элементов между двумя последовательными точками разрыва или точкой разрыва и граничным элементом, если полоса состоит из одного элемента или в ней элементы расположены по убыванию, и *возрастающей* — в противном случае.

Лемма 2. Если в перестановке есть убывающая полоса, то найдется инвертирование, уменьшающее $s(R)$.

Доказательство. Возьмем убывающую полосу с наименьшим правым концом. Пусть это значение r_i, стоящее на месте i. Значение $r_i - 1$ входить в убывающую полосу, очевидно, не может. В какой позиции j оно расположено, слева от i или справа? Если слева и $j < i$, то инвертируем участок $(j+1, i)$. При этом переход $(j, j+1)$ перестанет быть точкой разрыва (а почему он был точкой разрыва — ответьте сами), а переход $(i, i+1)$ не станет новой точкой разрыва — он заведомо был точкой разрыва, но может "заклеиться" (почему?). Если $j > i$, то инвертируем участок $(i+1, j)$. При этом переход $(i, i+1)$ перестанет быть точкой разрыва, а переход $(j, j+1)$ не станет новой точкой разрыва, — все, как в предыдущем случае (почему?). Отсюда следует наше утверждение. □

Лемма 3. Если в перестановке нет убывающих полос и она не тождественная, то для нее найдется инвертирование, не увеличивающее $s(R)$ и порождающее убывающую полосу.

Доказательство. Докажем, что в перестановке есть возрастающая полоса с точками разрыва по краям. Если самая левая полоса имеет разрыв в начале или самая правая имеет разрыв в конце, достаточно взять любую из них. Если у этих полос разрыва на соответствующей границе нет, то ввиду нетождественности перестановки найдется полоса между ними с разрывами на обоих концах.

Выбрав возрастающую полосу с двумя разрывами, инвертируем ее. Полоса станет убывающей, а точек разрыва не прибавится. □

Теперь ясно, как строить последовательность: пока есть убывающие полосы, выбирать инвертирование так, как это рекомендуется в лемме 2, а когда убывающих полос нет, то так, как рекомендуется в лемме 3.

Теорема. Число шагов в описанном алгоритме не превосходит $2s(R)$ и, следовательно, $4d(R)$.

Этот алгоритм был предложен в работах Дж. Д. Кесесиоглу и Д. Санкова (см. [69], а также [13], где об этой задаче и родственных ей сказано больше).

Четвертая задача будет более прикладной, она называется *задачей о порядке запуска деталей*, так как выбираемая перестановка задает расписание обработки деталей. Точная постановка задачи такова.

Задача о порядке запуска деталей. Имеется n деталей, которые должны пройти обработку на m станках, причем на всех и в одном и том же порядке — от первого до последнего. Детали на каждом станке обрабатываются в одном и том же порядке. Обработка детали на станке не может быть прервана или совмещена с обработкой этой детали на другом станке или другой детали на этом же станке. Задано время обработки t_{ij} каждой детали j на каждом станке i. Требуется выбрать такую последовательность обработки деталей, при которой время завершения обработки последней детали на последнем станке будет минимальным. ◻

Заранее скажем, что решение этой задачи известно только для случая двух станков. Но нам сейчас интересно не только само решение задачи. Очевидно, что множество допустимых решений — это множество перестановок из n элементов, каждая перестановка как раз и задает порядок обработки деталей. Возьмем какую-нибудь перестановку π, которую мы будем сейчас рассматривать как функцию, сопоставляющую каждой детали j ее номер в обработке $\pi(j)$. Введем набор чисел τ_{ij}, где $\tau_{i\pi(j)} = t_{ij}$, это будет набор времен, расположенных в порядке обработки деталей.

Введем теперь числа

$$\begin{aligned} s_{11} &= \tau_{11}, \\ s_{1j} &= s_{1,j-1} + \tau_{1j}, && j > 1, \\ s_{i1} &= s_{i-1,1} + \tau_{i1}, && i > 1, \\ s_{ij} &= \max\{s_{i-1,j}, s_{i,j-1}\} + \tau_{ij}, && i, j > 1. \end{aligned}$$

Легко видеть, что s_{ij} — это момент завершения обработки j-й по порядку детали на i-м станке, если первая деталь начала обрабатываться на первом станке в момент 0.

Например, пусть $m = 3$, $n = 7$ и времена обработки заданы таблицей:

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

При последовательности обработки деталей (2, 4, 7, 1, 3, 5, 6) мы получаем для чисел s таблицу:

	1	2	3	4	5	6	7
1	3	9	17	22	24	25	29
2	4	12	19	29	35	44	48
3	8	14	26	34	41	50	53

А при последовательности (2, 5, 7, 1, 3, 6, 4) —

	1	2	3	4	5	6	7
1	3	4	12	17	19	23	29
2	4	13	15	24	30	34	37
3	8	19	26	31	37	40	42

Видно, как сильно зависит значение целевой функции от порядка обработки деталей.

Информацию из этих таблиц можно очень наглядно выразить так называемыми *ленточными диаграммами*, в которых каждому станку соответствует своя "календарная полоска", а на ней показаны временные интервалы, когда станок занят обработкой той или иной детали (рис. 2.7).

Как уже говорилось, для случая двух станков удается найти простое решение этой задачи. Возьмем какое-либо расположение (перестановку) деталей

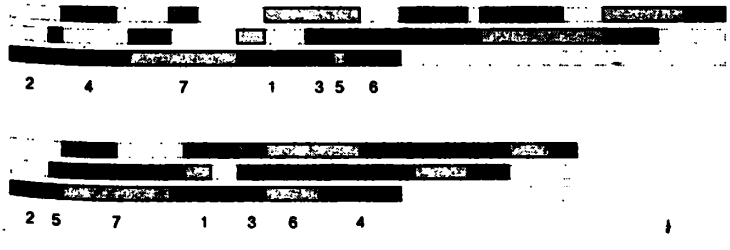


Рис. 2.7. Ленточные диаграммы для обоих расписаний

и рассмотрим две детали j и k, идущие в нем подряд. Если деталь j обрабатывается первой, то на первом станке она займет время t_{1j}, затем первый станок будет время t_{1k} обрабатывать вторую деталь, а второй станок — время t_{2j} обрабатывать первую деталь, и только после максимального из этих времен второй станок за время t_{2k} обработает вторую деталь. Итого общее время

$$T(j,k) = t_{1j} + \max\{t_{1k}, t_{2j}\} + t_{2k}.$$

Правда, эта формула не будет верна, если второй станок был занят предыдущими заказами, но в этом случае "вклад" наших двух деталей в общее время равен $t_{2j} + t_{2k}$ и не зависит от порядка обработки. Равным образом порядок несуществен, если важно только время занятости первого станка. Поэтому нам важно расположить детали так, чтобы минимизировать общее время в нашей формуле. Очевидно,

$$T(j,k) = \sum_{i \in 1:2} (t_{ij} + t_{ik}) - \min\{t_{1k}, t_{2j}\},$$

и мы получаем следующее правило расположения: перенумеровать детали так, чтобы при $j < k$ выполнялось неравенство $\min\{t_{1k}, t_{2j}\} \geq \min\{t_{1j}, t_{2k}\}$.

Упражнение 2.19. Покажите, что такая нумерация деталей всегда возможна и задаваемый ею порядок запуска минимизирует время обработки.

Мы еще раз обратимся к экстремальным задачам, связанным с перестановками, существенно позднее.

2.5. Размещения и сочетания

Размещением[*)] из n элементов по m называется (упорядоченный) набор из m различных чисел, принадлежащих множеству $1:n$. Обозначим множество всех таких размещений через $\mathcal{A}(n,m)$, а количество элементов этого множества через A_n^m.

Естественно задать те же основные вопросы теперь уже относительно множества $\mathcal{A}(n,m)$: чему равно A_n^m, как перебрать элементы этого множества, как их перенумеровать?

При определении количества элементов можно действовать так же просто, как при определении числа перестановок. Имеется n способов для выбора элемента, который ставится на первое место, $n-1$ способов выбора второго элемента и т.д. Это "и т.д." кончается $n-m+1$ способом выбора последнего, m-го, элемента. Как и в случае перестановок, для определения полного числа перестановок нужно перемно-

[*)] Allocation.

жить все эти числа, и мы получаем:

$$A_n^m = n \times (n-1) \times \cdots \times (n-m+1) = \frac{n!}{(n-m)!}.$$

Перебор размещений и их нумерация очень похожи на соответствующие действия с перестановками. Рассмотрение этих вопросов оставляется читателю в качестве полезного упражнения.

Больше даже и сказать-то про них нечего. По-моему, за всю мою научную жизнь я ни разу нигде не использовал размещений. Кроме одного их применения, которое и оправдывает их существование, — для определения сочетаний.

Сочетанием[*)] из n элементов по m называется множество (или, другими словами, неупорядоченный набор) из m различных чисел, принадлежащих множеству $1:n$. Обозначим совокупность таких подмножеств через $C(n, m)$, а количество ее элементов через C_n^m[**)].

Каждому такому подмножеству соответствует $m!$ размещений, которые получаются при задании того или иного порядка элементов, входящих в подмножество. Таким образом, $A_n^m = C_n^m \times m!$. Подставляя $A_n^m = \frac{n!}{(n-m)!}$, получаем окончательно

$$C_n^m = \frac{n!}{m!(n-m)!}.$$

Полезно осознать, почему при делении на $m!$ мы не получаем здесь дробного значения. Ответ прост: взятые подряд m сомножителей числителя содержат все простые сомножители знаменателя в той же или даже в большей степени, чем $m!$. Докажите это сами. Не забудьте, что нужно учесть кратности сомножителей.

Наше определение для числа сочетаний осмысленно при $m \in 0:n$. Но достаточно естественно доопределить его и считать равным нулю для всех остальных случаев. Будем считать, что доопределение сделано. Скоро оно нам понадобится.

А пока рассмотрим некоторые простые свойства сочетаний.

1) $C_n^m = C_n^{n-m}$.

Это следует прямо из формулы — она симметрична относительно m и $n-m$.

2) $C_n^m + C_n^{m+1} = C_{n+1}^{m+1}$.

*) Combination.

**) В англоязычной литературе предпочитают обозначение $\binom{n}{m}$. Обратите внимание: n наверху.

Действительно, непосредственно из формулы получаем

$$C_n^m + C_n^{m+1} = \frac{n!}{m!(n-m)!} + \frac{n!}{(m+1)!(n-m-1)!} =$$
$$= \frac{n!}{(m+1)!(n-m)!}((m+1)+(n-m)) = C_{n+1}^{m+1}.$$

3) $C_n^m \cdot C_m^k = C_n^k \cdot C_{n-k}^{m-k}$.

Эта формула также проверяется непосредственно.

С каждым сочетанием из n по m можно связать вектор из n нулей и единиц, в котором единиц ровно m, — числа, входящие в данное сочетание, просто задают номера этих единиц. В свою очередь, такому вектору, как уже говорилось, можно сопоставить траекторию, в которой, например, единицам соответствуют горизонтальные шаги единичной длины, а нулям — вертикальные шаги. Получается траектория, в которой m горизонтальных и $n-m$ вертикальных шагов; например, эта траектория ведет из точки $(0, 0)$ в точку $(m, n-m)$. Теперь доказанное нами свойство 2) можно истолковать следующим образом.

Число траекторий, ведущих в точку $(m+1, n-m)$, *равно сумме числа траекторий, ведущих в* $(m, n-m)$, *и числа траекторий, ведущих в* $(m+1, n-m-1)$. (Речь идет о разбиении множества на два подмножества.)

Для перебора сочетаний выберем удобное стандартное представление сочетания, например в виде монотонно возрастающего набора из m чисел, лежащих в диапазоне $1:n$, и будем перебирать только такие стандартные записи сочетаний.

Перебор сочетаний

Состояние вычислительного процесса. Массив $(x_1, \ldots, x_m)$ номеров, включенных в сочетание.

Начальное состояние. Принять $x_i = i$ для всех $i \in 1:m$.

Стандартный шаг. Просматривать компоненты вектора x начиная с x_m и искать первую компоненту, которую можно увеличить (нельзя увеличить $x_m = n$, $x_{m-1} = n-1$ и т.д.). Если такой компоненты не найдется, закончить процесс. В противном случае пусть k — наибольшее число, для которого $x_k < n-m+k$. Увеличить x_k на единицу, а для всех следующих за k-й компонентой продолжить натуральный ряд от нового значения x_k, т.е. положить $x_i = x_k + (i-k)$ для $i > k$.

Покажем порядок действий на сочетаниях из 7 элементов по 5. Количество таких сочетаний равно, очевидно, $(7 \cdot 6)/(1 \cdot 2) = 21$.

num	Сочетание					k
1	1	2	3	4	5	5
2	1	2	3	4	6	5
3	1	2	3	4	7	5
4	1	2	3	5	6	4
5	1	2	3	5	7	5
6	1	2	3	6	7	4
7	1	2	4	5	6	3
8	1	2	4	5	7	5
9	1	2	4	6	7	4
10	1	2	5	6	7	3

num	Сочетание					k
11	1	3	4	5	6	2
12	1	3	4	5	7	5
13	1	3	4	6	7	4
14	1	3	5	6	7	3
15	1	4	5	6	7	2
16	2	3	4	5	6	1
17	2	3	4	5	7	5
18	2	3	4	6	7	4
19	2	3	5	6	7	3
20	2	4	5	6	7	2
21	3	4	5	6	7	1

Упражнение 2.20. Сформулируйте предложенный способ перебора сочетаний в терминах наборов из n нулей и единиц с m единицами.

Организовать хорошую нумерацию для сочетаний существенно сложнее, чем для перестановок и размещений. В качестве одного из возможных подходов к такой нумерации может быть использование свойства 2), которое задает удобное разбиение множества сочетаний с данными параметрами на два подмножества, мощности которых известны. Идея в том, чтобы нумеровать для каждого такого сочетания сначала элементы одного подмножества, а потом второго (как это предлагалось на с. 14), в нумерации которых поступать точно так же, и формировать номер рекурсивно.

Удобно пользоваться представлением сочетаний как векторов из нулей и единиц. Выведем формулу для номера:

$$\mathrm{num}(b[1:n],\ m) = \begin{cases} \mathrm{num}(b[1:n-1],\ m), & b[n] = 0, \\ C_{n-1}^{m} + \mathrm{num}(b[1:n-1],\ m-1), & b[n] = 1. \end{cases}$$

Верхняя строчка определяет номера сочетаний из первого подмножества (такие же, как в этом подмножестве), нижняя — номера сочетаний из второго подмножества со сдвигом на число элементов первого подмножества.

Например, вектор $(0,1,0,1,0,0,1)$, представляющий одно из сочетаний из 7 элементов по 3 (их всего 35), получает номер

$$\begin{aligned}\mathsf{num}\big((0,1,0,1,0,0,1),3\big) &= C_6^3 + \mathsf{num}\big((0,1,0,1,0,0),2\big) = \\ &= C_6^3 + \mathsf{num}\big((0,1,0,1,0),2\big) = C_6^3 + \mathsf{num}\big((0,1,0,1),2\big) = \\ &= C_6^3 + C_3^2 + \mathsf{num}\big((0,1,0),1\big) = C_6^3 + C_3^2 + \mathsf{num}\big((0,1),1\big) = \\ &= C_6^3 + C_3^2 + C_1^1 + \mathsf{num}\big((0),0\big) = 20 + 3 + 1 + 0 = 24.\end{aligned}$$

Упражнение 2.21. Пусть единицы в 0–1 векторе $b[1:n]$ перенумерованы подряд, начиная с 1, и p_i — номер позиции, которую занимает i-я единица. Считая, что $C_s^t = 0$ при $s < t$, докажите, что $\mathsf{num}(b[1:n], m) = \sum_{i \in 1:m} C_{p_i-1}^i$.

Ту же нумерацию можно было бы ввести, исходя и из обычного представления сочетаний. Обратимся к последней таблице. В ней первые 15 номеров получают сочетания, в которые входит элемент 1 (число таких сочетаний равно C_6^4, так как мы израсходовали один элемент в n и один в m). Дальше 6 номеров получают сочетания, в которые элемент 1 не входит, их число равно $C_6^5 = 6$.

Конечно, это не очень удобный способ нумерации, и он применяется не часто. Но иногда полезно знать и использовать саму идею такой нумерации: разбиение нумеруемого множества на части и составление его нумерации из нумераций частей. Вот пример использования этой идеи в другой ситуации.

Пример. Понадобилось перебирать наборы из d неотрицательных целых чисел $x = (x_1, \ldots, x_d)$, для которых выполняется неравенство $\sum_{i \in 1:d} x_i^2 < s$ с заданными d и s.

Было бы очень удобно перенумеровать все такие наборы и построить метод, позволяющий по номеру восстановить набор. Обозначим число таких наборов через $N(d,s)$ и будем рассматривать различные d и s. Очевидно, что $N(d,0) = 0$, $N(1,s) = \lfloor\sqrt{s-1}\rfloor + 1$ (например, $N(1,4) = 2$, что соответствует 0 и 1, а число 2 нам уже не годится).

Ясно, что точно так же можно перебрать все значения последнего числа в наборе

$$0 \le x_d \le \lfloor\sqrt{s-1}\rfloor.$$

При каждом фиксированном значении $x[d]$ мы получаем такую же задачу перебора наборов длины $d-1$ с суммой квадратов, меньшей $s - x_d^2$. Поэтому

мы имеем для $d > 1$ и $s > 0$

$$N(d, s) = \sum_{i, i^2 < s} N(d-1, s-i^2).$$

Эта формула позволяет нам легко составить таблицу для $N(d, s)$, которая затем может быть использована для восстановления набора по его номеру.

Действительно, задавшись для примера $d = 4$ и $s = 6$, получаем:

d	0	1	2	3	4	5	6
1	0	1	2	2	2	3	3
2	0	1	3	4	4	6	8
3	0	1	4	7	8	11	17
4	0	1	5	11	15	20	32

Найдем по этой таблице набор с номером 11. Общее число наборов 32 состоит из 17 наборов с $x_4 = 0$, которые получают начальные номера, затем идут 11 наборов с 1 и 4 с 2. Номер 11 лежит в первой группе, и значит, $x_4 = 0$, а все остальное образует набор 11 при $d = 3$ и $s = 6$. Аналогично, $17 = 8 + 6 + 3$, так что 11 попадает не в первую группу, а является третьим номером во второй группе, $x_3 = 1$ и для дальнейшего $d = 2$, $s = 5$. Теперь $6 = 3 + 2 + 1$, так что $x_2 = 0$ и, наконец, $x_1 = 2$ — третье возможное значение.

Числа $N(d, s)$ с ростом d и s увеличиваются очень быстро, например $N(20, 20) = 125261230$, и если вы хотите работать с большими значениями аргументов, то здесь не обойтись без "длинной арифметики" — специальных методов представления длинных целых чисел.

Упражнение 2.22. По аналогии с алгоритмом перебора сочетаний разработайте алгоритм перебора разбиений множества $1 : n$.

Упражнение 2.23. Пусть D_n — множество всех 0–1 последовательностей, длина которых не превосходит n. Подсчитайте $|D_n|$ и предложите способ нумерации и перебора элементов D_n в лексикографическом порядке.

2.6. Бином Ньютона и его комбинаторные использования

Интересное получается зрелище, если в узлах положительного квадранта прямоугольной координатной решетки поместить количества путей от начала координат до этих узлов. Лучше всего повернуть эту решетку так, чтобы начало координат было наверху, а квадрант был

симметрично направлен вниз:

```
                1
            1       1
        1       2       1
    1       3       3       1
1       4       6       4       1
    5      10      10       5
```

Эта схема называется *треугольником Паскаля* *).

То, что в этой таблице стоят числа сочетаний, видно из приводившегося свойства 2). По таблице очень удобно быстро считать число сочетаний для небольших значений параметра.

На треугольнике Паскаля особенно удобно показывать обсуждавшуюся только что нумерацию путей (рис. 2.8). В каждой позиции треугольника запишем слева от числа путей номера путей, приходящих слева, а справа — номера путей, приходящих справа. Например, $_{0:2}4_3$ обозначает, что в данную позицию приходит 4 пути, из них три приходят слева и имеют номера 0, 1 и 2, а один приходит справа и имеет номер 3.

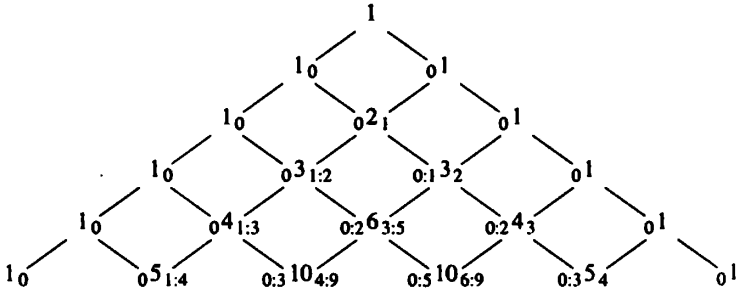


Рис. 2.8. Нумерация путей в треугольнике Паскаля

С треугольником Паскаля тесно связана замечательная формула (возможно, вам уже встречавшаяся), называемая формулой *бинома Ньютона* **):

$$(a+b)^n = \sum_{k=0}^{n} C_n^k a^k b^{n-k}$$

*) Blaise Pascal (1623–1662), знаменитый французский математик, физик, философ и писатель.

**) Sir Isaac Newton (1643–1727). По-видимому, это имя в комментариях не нуждается. Слово *бином* значит *двучлен* — выражение из двух слагаемых.

— при раскрытии степени алгебраического двучлена коэффициенты при степенях в точности совпадают с числом сочетаний. Сама эта формула легко доказывается по индукции. Действительно, при $n = 1$ она очевидно верна. Далее, предположив, что формула верна для $n - 1$, получаем:

$$\begin{aligned}(a+b)^n &= a\cdot(a+b)^{n-1} + b\cdot(a+b)^{n-1} = \\ &= \sum_{k=0}^{n-1} C_{n-1}^k a^{k+1} b^{(n-1)-k} + \sum_{k=0}^{n-1} C_{n-1}^k a^k b^{1+(n-1)-k} = \\ &= \sum_{k=1}^{n} C_{n-1}^{k-1} a^k b^{n-k} + \sum_{k=0}^{n-1} C_{n-1}^k a^k b^{n-k} = \\ &= \sum_{k=0}^{n} (C_{n-1}^{k-1} + C_{n-1}^k) a^k b^{n-k} = \sum_{k=0}^{n} C_n^k a^k b^{n-k}.\end{aligned}$$

Вот в этой выкладке нам и потребовалось доопределение C_n^m на "запредельные" значения.

В связи с формулой Ньютона значения C_n^m часто называются *биномиальными коэффициентами.*

Биномиальное разложение Ньютона служит основой для многих комбинаторных формул. Рассмотрим некоторые из них.

1) Подставив в формулу $a = b = 1$, получаем:

$$\sum_{k=0}^{n} C_n^k = 2^n.$$

Можно эту формулу трактовать геометрически: рассмотрим единичный куб в n-мерном пространстве; как мы помним, он имеет 2^n вершин. Проведем прямую, соединяющую вершину $(0, \ldots, 0)$ и вершину $(1, \ldots, 1)$, и спроектируем на эту прямую все прочие вершины. Вершины с одинаковым числом единичных координат m все проектируются в одну и ту же точку, их будет ровно C_n^m.

2) Подставив в формулу $a = 1$, $b = -1$, получаем:

$$\sum_{k=0}^{n} C_n^k (-1)^k = 0.$$

Следовательно, суммы биномиальных коэффициентов, стоящих на четных и на нечетных местах, равны между собой, и каждая равна 2^{n-1}.

3) Интересные эффекты получаются, если воспользоваться комплексными числами. Подставив в формулу $a = 1$, $b = i$, имеем:

$$\sum_{k=0}^{n} C_n^k i^k = (1+i)^n.$$

Комплексное число $1 + i$ по формуле Эйлера представляется в виде

$$z = 1 + i = \sqrt{2}\left(\cos\frac{\pi}{4} + i\sin\frac{\pi}{4}\right).$$

Применив формулу Муавра*) для комплексного числа $z = \rho e^{i\varphi}$, возводимого в степень n:

$$z^n = \rho^n e^{in\varphi} = \rho^n(\cos n\varphi + i\sin n\varphi),$$

получаем:

$$\sum_{k=0}^{n} C_n^k i^k = 2^{n/2} e^{in\pi/4}.$$

Левая часть этого выражения разделяется на вещественную и мнимую части. Вещественная часть — это знакопеременная сумма слагаемых с четными индексами $S_{\text{чст}} = C_n^0 - C_n^2 + \cdots$, мнимая — аналогичная сумма $S_{\text{неч}}$ для нечетных индексов. Итак, получаем:

$$S_{\text{чст}} = 2^{n/2}\cos\frac{n\pi}{4}, \qquad S_{\text{неч}} = 2^{n/2}\sin\frac{n\pi}{4}.$$

Упражнение 2.24. Докажите самостоятельно следующие формулы:

$$C_{2n}^n = \sum_{j=0}^{n} \left(C_n^j\right)^2,$$

$$C_{m+n}^n = \sum_{j=0}^{n} C_m^j C_n^j, \quad m \le n$$

(в конце курса мы вернемся к подобным приемам получения явных формул).

Упражнение 2.25. Напишите формулы для $(a+b+c)^n$, а также для $(a_1 + a_2 + \cdots + a_k)^n$.

*) Abraham de Moivre (1667–1754), английский математик. Французские энциклопедии говорят: французский математик, живший в Англии. Можно добавить: гугенот, спасавшийся от религиозных преследований.

2.7. Числа Фибоначчи

Итальянский математик Леонардо Пизанский [*)] в своей книге 1202 г. ввел очень интересную последовательность чисел, определяемую следующими соотношениями:

$$F_0 = 0; \qquad F_1 = 1;$$
$$F_n = F_{n-1} + F_{n-2}, \qquad n > 1.$$

Таким образом, получаем

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Числа Фибоначчи имеют много замечательных свойств и связанных с ними приложений. Достаточно взглянуть на поведение отношений $\varphi_n = F_{n+1}/F_n$ (пропустим φ_0)

$$1, 2, 1.5, 1.66667, 1.6, 1.675, 1.61538, 1.61905, 1.61764, \ldots$$

— есть твердая уверенность, что эта последовательность сходится. К какому значению φ может сходиться последовательность? Предположив, что это законно, перейдем к пределу в отношении

$$\frac{F_{n+1}}{F_n} = \frac{F_n}{F_n} + \frac{F_{n-1}}{F_n}$$

и получим $\varphi = 1 + \varphi^{-1}$. Таким образом, φ легко находится из квадратного уравнения [**)]

$$\varphi = \frac{\sqrt{5}+1}{2} \approx 1.618034.$$

Вопрос о законности предельного перехода сам собой решится, если найти явную формулу для числа F_n; мы отложим эту тему почти до конца курса. А сейчас получим полезные оценки роста для F_n. Умножая обе части соотношения для φ на φ^{n+1}, получаем $\varphi^{n+2} = \varphi^{n+1} + \varphi^n$. Таким образом, степени числа φ удовлетворяют тому же рекуррентному соотношению, что и числа Фибоначчи. Отсюда легко получаются следующие неравенства.

Лемма 1. При $n > 0$ выполняются неравенства $\varphi^{n-1} \leq F_n \varphi^n$.

Доказательство основано на индукции с использованием сделанного выше замечания. Достаточно только убедиться в справедливости этих неравенств для малых n. □

А сейчас докажем очень простую формулу.

[*)] Leonardo Pisano (около 1170 — после 1228), известный также как Фибоначчи.

[**)] Это число называется *золотым сечением*. Прямоугольник с отношением сторон $\varphi : 1$ обладает тем свойством, что, если отрезать от него квадрат, пропорции оставшегося прямоугольника будут такие, как у исходного. Почему-то они особенно приятны для глаза и часто встречаются в искусстве.

Лемма 2. При $k > 2$ имеют место равенства

$$F_{2k} = F_{2k-1} + F_{2k-3} + \cdots + F_1,$$
$$F_{2k+1} = 1 + F_{2k} + F_{2k-2} + \cdots + F_0.$$

Доказательство оставляется читателю. Оно использует индукцию, для каждой формулы отдельно. Формулы леммы нам будет удобно объединить в одну универсальную:

$$F_n = 1 + \sum_{1 \le i \le (n-2)/2} F_{n+1-2i}, \qquad (*)$$

в которой исключение из суммирования "младших слагаемых" F_0 и F_1 позволяет выделить в обоих случаях слагаемое 1. Читателю предоставляется возможность доказать и еще одну формулу: $F_{n+2} = 1 + \sum_{i=1}^{n} F_i$. □

Теорема. Любое натуральное число s однозначно представимо в виде суммы чисел Фибоначчи

$$s = F_{i_0} + \cdots + F_{i_r}, \qquad (**)$$

где $i_0 = 0$ и для любого $k \in 1 : r$ выполняется неравенство $i_{k-1} + 1 < i_k$ *).

Доказательство. Прежде всего покажем, что такое представление существует для любого s. Пусть $j(s)$ — номер максимального числа Фибоначчи, не превосходящего s. Положим $s' = s - F_{j(s)}$. Из определения $j(s)$ следует, что $s' < F_{j(s)-1}$, иначе число Фибоначчи не было бы максимальным. Теперь мы получим искомое представление для s как представление для s', дополненное слагаемым $F_{j(s)}$.

Осталось показать однозначность представления. Пусть кроме представления (**) имеется еще одно:

$$s = F_{j_0} + \cdots + F_{j_q}.$$

Не умаляя общности, можно считать, что $j_q < j(s)$ (больше быть не может, а равные старшие слагаемые можно отбросить). Если мы заменим F_{j_q} на $F_{j(q)-1}$, то правая часть разве лишь увеличится. Аналогично заменим с возможным увеличением предпоследнее слагаемое на $F_{j(s)-3}$ и т. д. Но при всем этом увеличении сумма не сможет подняться до значения $F_{j(s)}$, даваемого формулой (**), по крайней мере из-за отсутствия слагаемого 1, следовательно, другое представление для s невозможно. □

Характеристический вектор набора $(i_0, i_1, \ldots, i_r)$ можно рассматривать как запись числа s в появляющейся у нас новой системе счисления — *фибоначчиевой*. При n двоичных разрядах мы можем записать в этой системе числа от 0 до $F_n - 1$. Отсюда следует, что *число* 0–1 *последовательностей длины* n, *которые начинаются с* 1 *и не содержат единиц, идущих подряд, равно* F_n.

*) Д. Кнут [23], который много пишет о числах Фибоначчи, вводит даже специальный знак для такого неравенства: $i_{k-1} \ll i_k$.

Посмотрим, что нужно сделать для увеличения числа на 1 при использовании этой системы счисления.

Прибавление единицы в фибоначчиевой системе

Состояние вычислительного процесса. Набор $x[0:n-1]$ из n нулей и единиц, в котором нет двух единиц рядом и $x[0] \neq 1$.

Начальное состояние. Положить $x[1] := 1$.

Стандартный шаг. Среди значений k, для которых $x[k] = 1 \wedge x[k-1] = 1$, выбрать наибольшее. Если такого k не найдется, закончить процесс. В противном случае положить $x[k-1] := 0$, $x[k] := 0$, $x[k+1] := 1$ и $x[0] := 1$ — на случай, если нам пришлось изменить $x[0]$.

Пример. Прибавим единицу к числу 46. Имеем

0	1	2	3	4	5	6	7	8	9	Комментарий
1	0	1	0	1	0	1	0	0	1	$=46$
1	1	1	0	1	0	1	0	0	1	Начало процесса
1	0	0	1	1	0	1	0	0	1	Изменение для $k=2$
1	0	0	0	0	1	1	0	0	1	Изменение для $k=4$
1	0	0	0	0	0	0	1	0	1	Изменение для $k=6$

Используя операцию увеличения числа на 1, мы можем перебрать все 0–1 векторы, являющиеся фибоначчиевыми разложениями. Попробуйте сами!

В заключение упомянем интересную связь чисел Фибоначчи с треугольником Паскаля: можно так выбрать наклон линий, пересекающих числа треугольника, что при суммировании этих чисел на каждой линии получатся числа Фибоначчи. Первая линия должна пересечь верхнюю единицу. Вторая — правую единицу второго ряда. Третья линия определяет наклон, она пересекает левую единицу второго ряда и правую единицу третьего ряда, их сумма равна 2. Далее аналогично.

Упражнение 2.26. Введите в треугольнике Паскаля систему координат его позиций, определите правило, по которому определяются позиции, перечеркнутые k-й прямой, и докажите, что сумма чисел, перечеркнутых k-й прямой, действительно равна F_k.

Упражнение 2.27 [68]. Сопоставим 0–1 вектору $X = (x_1, \ldots, x_n)$ матрицу $K(X) = K_1 \times \cdots \times K_n$, где матрица K_i равна K_E при $x_i = 1$ и K_Z при $x_i = 0$, а

$$K_E = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \qquad K_Z = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

Докажите, что все матрицы $K(X)$ различны и элементы $K(X)$ не превосходят n-го числа Фибоначчи F_n.

Глава 3

Элементарная теория вероятностей

3.1. Основные определения

Мы часто сталкиваемся в жизни с ситуациями, когда, зная в принципе возможные результаты какого-либо события, не можем предсказать его точный исход: пойдет ли завтра дождь, какой номер выиграет в лотерее, когда перегорит только что купленный сканер и т. п.

Однако мы можем сравнивать разные исходы по тому, насколько правдоподобно их осуществление, вводя для сравнения некоторую количественную меру. Изучением такой меры и занимается *теория вероятностей*. В основе этой теории лежат специальные математические модели, в которых сопоставлению событий по степени их правдоподобия можно придать точный смысл. Теория вероятностей входит в обязательное математическое образование, и вам еще предстоит ее изучать более подробно, но мы не можем ждать этого систематического изложения, так как некоторое первоначальное знакомство с ней потребуется уже сейчас. Разумеется, мы ограничимся только теми сведениями, без которых невозможно обойтись.

В *элементарной* теории вероятностей, которой мы сейчас будем заниматься (это принятый термин), в качестве примеров используются довольно странные модели: игральные карты и кости, бросаемые монеты, известная игра в рулетку, урна с шарами. Собственно, с этих моделей (именно с игры в кости, в переписке Б. Паскаля и П. Ферма [*)]

[*)] Pierre de Fermat (1601–1665), французский математик и юрист. Его имя повсеместно известно благодаря так называемой великой теореме Ферма.

исторически и началась теория вероятностей, но дело не только в истории — такие игровые модели очень удобны для первоначального рассмотрения вопроса.

Объект *Монета* подбрасывается и падает какой-либо стороной вверх. Рассматривается не реальный объект, а модельный, поэтому считается невозможным, что монета при падении станет на ребро (мы все понимаем, что шансы падающей монете встать на ребро очень малы). Стороны Монеты, которые традиционно называются *Герб* и *Решетка* (дворовый вариант — орел и решка), равноправны, с равным шансом можно ожидать появления любой из них (за исключением особо оговариваемого случая, когда объект называется *Фальшивая монета*).

Игральная кость тоже подбрасывается и падает на одну из своих шести граней, результат бросания определяется гранью, оказавшейся наверху. На каждой грани написано число от 1 до 6 (на самом деле изображено соответствующее количество точек), это число на верхней грани и есть результат. Все грани равноправны.

Игральные карты составляют колоду, в которую входят карты различной масти (две красные — Бубны и Черви, две черные — Пики и Трефы) и различного значения (Тузы, Короли, Дамы, Валеты, 10, 9 и далее до 2). Карт каждой масти — 13, каждого значения — 4. Когда вынимают случайную карту из *хорошо перетасованной колоды*, считается, что все карты равноправны.

Естественно сопоставлять шансы на появление различных событий по тому, сколько эти события содержат равноправных элементарных исходов. Так, в событии "вынута карта пиковой масти" содержится 13 исходов, а в событии "вынута фигура" — 12 (туз не фигура).

Однако такой измеритель, как число исходов, неудобен — трудно сопоставлять разные модели. Например, если сравнить появление шестерки на игральной кости (один исход из шести) и туза из колоды карт (4 исхода из 52), то ясно, что, хотя во втором событии элементарных исходов больше, шансов его осуществления меньше.

Можно потребовать, чтобы числовая характеристика осуществимости события всегда лежала в каком-либо определенном диапазоне значений. В качестве такого диапазона принимается отрезок [0, 1], так что характеристика не может быть отрицательной и не превосходит 1. Вот эта характеристика события и называется его *вероятностью*.

Проще всего ввести такие характеристики, определив множество *элементарных событий* S, элементы которого (они называются также *исходами*) считаются равноправными, *равновероятными*. Любое рассматриваемое нами событие A будет подмножеством S, оно будет

составлено из элементарных событий. Отношение $|A|/|S|$ называется *вероятностью события A* и обозначается через $P(A)$.

Событие, вероятность которого равна 1, называется *достоверным* — оно наступает всегда. (Монета опустится Гербом или Решеткой вверх, слово "всегда" трактуется с точки зрения принятой модели.)

Событие с вероятностью 0, называется *невозможным* — оно не наступает никогда. (Модельная Монета не встанет на ребро.)

При сопоставлении событий требуется также, чтобы вероятность любого события A, содержащегося в другом событии B, имела вероятность, не превосходящую вероятности B (события — это множества, и они могут содержаться одно в другом).

Если событие A составляется из событий, которые не могут выполняться одновременно (такие события называются *несовместными*), то вероятность осуществления события A должна равняться сумме вероятностей составляющих событий. (Иначе говоря, если задано разбиение множества исходов A на подмножества A_i, то вероятность события A равна сумме вероятностей событий A_i.)

Всем этим условиям удовлетворяет введенное нами определение вероятности события как *отношения числа входящих в событие — благоприятных — равновозможных исходов к числу всех исходов.*

Например, при бросании двух игральных костей вероятность получить не меньше 11 очков равна $\frac{3}{36}$, так как всего имеется 36 равновозможных исходов и три из них благоприятны: (5, 6), (6, 5) и (6, 6).

Было бы неправильно считать, что так как может выпасть от 2 до 12 очков и благоприятными являются два исхода из этих 11, то вероятность равна $\frac{2}{11}$. Исходы типа "число выпавших очков" не равновозможны, например 12 очкам соответствует всего одна комбинация очков, а 7 очкам — целых шесть (какие?).

Вероятность того, что на первой кости очков выпадет больше, чем на второй, равна $\frac{15}{36}$. Проверьте.

Пример 1. Вероятность выпадения четного числа гербов при любом числе бросаний монеты равна $\frac{1}{2}$. Действительно, при любом n верна формула $\sum_{k\in 0\cdot n}(-1)^k C_n^k = 0$, так что суммы коэффициентов, стоящих на четных и на нечетных местах, равны. Вероятность выпадения четного числа гербов равна отношению одной из этих сумм (благоприятные исходы) к сумме обеих (все исходы).

Пример 2. В эксперименте участвуют две колоды карт: *полная* — из 52 карт и *сокращенная* — из 36 карт (тузы, фигуры и карты с очками от 10 до 6). Случайно выбирается одна из колод (*полная* с вероятностью 0,7,

сокращенная с вероятностью $1 - 0.7 = 0.3$), и из этой колоды берется одна из карт. Какое здесь пространство равновероятных элементарных событий? Совершенно очевидно, что карты неравноправны, они делятся на две группы: "редкие" карты, которые содержатся только в первой колоде, например четверка треф, и "частые" карты вроде дамы пик.

Для обеспечения равной вероятности исходов приходится множество исходов усложнять. Обозначим через C_{52} множество карт полной колоды, через C_{36} — множество карт сокращенной колоды, $N = 1 : 10$, и рассмотрим прямое произведение этих трех множеств $S = N \times C_{52} \times C_{36}$. Каждому элементарному событию (i, c, d), где $i \in N$, $c \in C_{52}$, $d \in C_{36}$, сопоставим *результат исхода*, равный c при $i \leq 7$ и d при $i > 7$. Число элементарных событий в множестве S равно $10 \cdot 52 \cdot 36 = 18720$; чтобы найти вероятность появления какой-либо карты, нужно найти "число благоприятных событий", в которых исходом будет эта карта.

Для редких карт, например для 4♣, нужный нам результат дают исходы, у которых $i \leq 7$ и $c = 4♣$; число таких исходов равно $7 \cdot 1 \cdot 36 = 252$, так что $\mathsf{P}(4♣) = \frac{252}{18720} = \frac{7}{520}$.

Для дамы пик благоприятны исходы, у которых $i \leq 7$ и $c = D♠$, а также те, у которых $i > 7$ и $d = D♠$, первых исходов 252, вторых — $3 \cdot 52 \cdot 1 = 156$; так что $\mathsf{P}(D♠) = \frac{252+156}{18720} = \frac{17}{780}$.

Упражнение 3.1. Монета бросается 127 раз. Какова вероятность того, что герб выпадет больше 63 раз?

Упражнение 3.2. Монета бросается 7 раз. Почему вероятность того, что герб выпадет 3 раза, не равна ни $\frac{1}{8}$, ни тем более $\frac{1}{7}$? Какова эта вероятность на самом деле?

Упражнение 3.3. Найдите вероятность того, что при бросании двух игральных костей на них в сумме выпадет 5 очков.

Событие, которое составлено из всех элементарных событий, входящих одновременно в событие A и в событие B, называется *совмещением* этих событий. Таким образом, множество элементарных событий, входящих в совмещение, — это пересечение соответствующих множеств. Очевидно, вероятность совмещения событий не превосходит вероятности каждого из этих событий.

Событие, которое составлено из всех элементарных событий, входящих в событие A или в событие B, называется *объединением* этих событий. Таким образом, множество элементарных событий, входящих в объединение, — это объединение соответствующих множеств. Как уже говорилось, вероятность объединения несовместных событий равна сумме вероятностей этих событий.

Оба эти определения легко переносятся на несколько событий.

Разбиение множества элементарных событий S на несовместные события $S_1, \ldots, S_m$ называется *полной системой событий*. Каждое событие A можно представить как объединение несовместных событий — совмещений A с элементами разбиения S_i. Очевидно равенство

$$\mathsf{P}(A) = \sum_{i \in 1:m} \mathsf{P}(A \cap S_i).$$

Эта формула называется *формулой полной вероятности*.

События A и B называются *независимыми*, если вероятность их совмещения равна произведению их вероятностей:

$$\mathsf{P}(A \cap B) = \mathsf{P}(A) \times \mathsf{P}(B).$$

Независимость событий сильно упрощает вычисление вероятностей. Ее наличие часто видно из побочных соображений, само название свойства отражает уверенность в отсутствии связи между случайными событиями, хотя формальное определение допускает и такую независимость событий, которая естественной показаться не может.

Пример 3. Рассмотрим колоду в 52 карты, из которой извлекается одна карта. Событие A = "карта красной масти" и событие B = "туз" являются, очевидно, независимыми, и это вполне естественно. Но согласно нашему определению события A и $C = \{2\spadesuit, 2\clubsuit, D\heartsuit, K\heartsuit\}$ также будут независимы. В чем естественность независимости событий A и B? Множество элементарных событий является произведением двух множеств: мастей и значений карты. Множества A и B являются цилиндрическими, они определены соответственно подмножеством мастей и подмножеством значений, и нам естественно считать, что *выбор масти и значения карты происходит независимо*. Во взаимодействии событий A и C таких аргументов у нас нет — их независимость определяется игрой чисел.

Пример (продолжение примера 2). В рассмотренной схеме с двумя колодами разного размера множество элементарных событий очевидно делится на два несовместных события: $S_1 = \{(i, c, d) \mid i \leq 7\}$ и $S_2 = \{(i, c, d) \mid i > 7\}$, при этом $\mathsf{P}(S_1) = 0.7$ и $\mathsf{P}(S_2) = 0.3$. Когда мы вычисляли вероятность того, что будет выбрана дама пик, фактически мы считали вероятности событий $S_k \cap S_{D\spadesuit}$, $k \in 1:2$. Поскольку события S_1 и $\{c = D\spadesuit\}$ независимы, то можно не строить большое множество элементарных событий, а прямо выяснить, что

$$\mathsf{P}(S_1 \cap S_{D\spadesuit}) = \mathsf{P}(S_1 \cap \{c = D\spadesuit\}) = \mathsf{P}(S_1) \cdot \mathsf{P}(\{c = D\spadesuit\}) = \frac{7}{10} \cdot \frac{1}{52}.$$

Равным образом,

$$\mathsf{P}(S_2 \cap S_{D\spadesuit}) = \mathsf{P}(S_2 \cap \{d = D\spadesuit\}) = \mathsf{P}(S_2) \cdot \mathsf{P}(\{d = D\spadesuit\}) = \frac{3}{10} \cdot \frac{1}{36}.$$

Упражнение 3.4. Из колоды в 52 карты вынимаются две карты. Независимы ли события "первая карта — фигура" и "вторая карта — фигура"?

Упражнение 3.5. В той же схеме будут ли независимы события "обе карты одной масти" и "ровно одна из карт — фигура"?

Когда рассматривается большее число событий, скажем $A_1, A_2, A_3, \ldots, A_k$, где $k > 2$, следует различать попарную независимость этих событий и их *независимость в совокупности*. В первом случае

$$\mathsf{P}(A_i \cap A_j) = \mathsf{P}(A_i) \times \mathsf{P}(A_j) \text{ при } i \neq j,$$

во втором случае (более жесткое требование) для любого $I \subset 1:k$ должно выполняться равенство

$$\mathsf{P}\Big(\bigcap_{i\in I} A_i\Big) = \prod_{i\in I} \mathsf{P}(A_i).$$

Легко показать, что независимость событий в совокупности не следует из их попарной независимости.

Пример (С. Н. Бернштейн *)). Рассмотрим игральную кость в форме правильного тетраэдра. Одна грань этой кости выкрашена в белый цвет W, вторая — в черный B, третья — в красный R, а окраска четвертой — смешанная M, в ней есть все три цвета. При каждом бросании кость ложится какой-то стороной вниз. Вероятность того, что на нижней грани окажется белый цвет, равна, очевидно, $\frac{1}{2}$, так как для этого события благоприятны два исхода из четырех; для любого другого цвета она такая же. Вероятность выпадения двух цветов сразу, например красного и черного, равна $\frac{1}{4}$, также независимо от конкретной пары цветов. Таким образом, для событий "выпал красный цвет" и "выпал черный цвет" имеем $\mathsf{P}(R \cap B) = \mathsf{P}(R) \cdot \mathsf{P}(B)$, и по определению эти события независимы (уж таково определение!). С другой стороны,

$$\mathsf{P}(R \cap B \cap W) = \frac{1}{4} \neq \mathsf{P}(R) \cdot \mathsf{P}(B) \cdot \mathsf{P}(W) = \frac{1}{8},$$

и независимости в совокупности нет.

3.2. Условные вероятности и формула Байеса

При работе с зависимыми случайными событиями необходимо учитывать их общее влияние на вероятностное поведение. Влияние проявляется в том, что осуществление одного события может изменить вероятности других событий, и приходится говорить о вероятностях событий при тех или иных условиях. Формально вводится следующее определение.

*) Сергей Натанович Бернштейн (1880–1968), один из крупнейших российских математиков, академик. Был профессором Ленинградского университета.

Пусть A — некоторое событие, имеющее положительную вероятность. Определим для каждого события B *условную вероятность* B *при условии* A как отношение

$$\mathsf{P}(B|A) = \frac{\mathsf{P}(B \cap A)}{\mathsf{P}(A)},$$

так что $\mathsf{P}(B \cap A) = \mathsf{P}(B|A) \cdot \mathsf{P}(A)$. Легко видеть, что так определенные условные вероятности обладают всеми свойствами обычных вероятностей: они меняются от 0 до 1, возрастают с расширением события, условная вероятность объединения несовместных событий равна сумме их условных вероятностей.

Используя условную вероятность, независимость событий A и B можно определить так: события независимы, если $\mathsf{P}(B|A) = \mathsf{P}(B)$.

Используя условные вероятности, мы можем видоизменить формулу полной вероятности, заменяя в ней $\mathsf{P}(B \cap A_i)$ на $\mathsf{P}(B|A_i)\,\mathsf{P}(A_i)$:

$$\mathsf{P}(B) = \sum_i \mathsf{P}(B|A_i)\,\mathsf{P}(A_i).$$

Эта формула во многих случаях сильно облегчает вычисление вероятностей, даже в случае независимых событий.

Пример. Завод выпускает модули памяти на двух линиях. Старая линия выпускает в 2 раза меньше продукции, чем новая, а доля брака у нее в 4 раза больше. Что можно сказать о доле брака в продукции завода?

Модуль, выбранный в продукции случайно, с вероятностью $\mathsf{P}(1) = \frac{1}{3}$ окажется изготовленным на старой линии. Если обозначить вероятность бракованного изделия на старой линии и на новой, соответственно, через P_1 и P_2, то для искомой вероятности P имеем:

$$P = P_1 \cdot \mathsf{P}(1) + P_2 \cdot (1 - \mathsf{P}(1)) = P_2 \cdot \left(4 \cdot \frac{1}{3} + \frac{2}{3}\right) = 2 \cdot P_2 = 0.5 \cdot P_1.$$

Таким образом, пуск новой линии снизил долю брака в два раза.

Из формулы полной вероятности легко получить еще одну очень важную формулу, принадлежащую английскому математику Байесу*). Выражая $\mathsf{P}(B \cap A_i)$ с помощью условных вероятностей, мы видим, что

$$\mathsf{P}(A_i|B)\,\mathsf{P}(B) = \mathsf{P}(B|A_i)\,\mathsf{P}(A_i).$$

Разделив обе части этой формулы на $\mathsf{P}(B)$ и подставляя выражение для $\mathsf{P}(B)$ из формулы полной вероятности, получаем *формулу Байеса*

$$\mathsf{P}(A_i|B) = \frac{\mathsf{P}(B|A_i)\,\mathsf{P}(A_i)}{\sum_j \mathsf{P}(B|A_j)\,\mathsf{P}(A_j)}.$$

*) Thomas Bayes (1702–1761). Иногда встречается транскрипция "Бейес", теперь все реже.

Эта формула имеет следующее истолкование и применение. Предположим, что случайное событие B зависит от некоторого случайного параметра, который определяется заданием полной системы событий $\{A_i\}$. Известны некоторые (*априорные* — заданные до опыта) вероятности событий A_i. Нас интересует, какое из событий осуществилось. Мы проводим эксперимент и узнаем, что осуществилось событие B. Информация об этом дает возможность пересчитать вероятности A_i — получить *апостериорные* вероятности.

Пример 1. Вернемся к примеру с двумя колодами, в 52 и 36 листов. Пусть мы выбираем одну из колод с равной вероятностью, затем достаем из выбранной колоды карту и, зная ее, хотим угадать, какая колода нам попалась. Если эта карта отсутствует в меньшей колоде, то, очевидно, вероятность меньшей колоды равна нулю. А если присутствует, например $T\diamondsuit$, повлияет ли эта информация на вероятности колод?

Имеем

$$\mathrm{P}(B_{36}|T\diamondsuit) = \frac{\mathrm{P}(T\diamondsuit|B_{36})\mathrm{P}(B_{36})}{\mathrm{P}(T\diamondsuit|B_{36})\mathrm{P}(B_{36}) + \mathrm{P}(T\diamondsuit|B_{52})\mathrm{P}(B_{52})} = \frac{\frac{1}{36}\cdot\frac{1}{2}}{\frac{1}{36}\cdot\frac{1}{2} + \frac{1}{52}\cdot\frac{1}{2}} = \frac{52}{88}.$$

Вероятность меньшей колоды в результате эксперимента чуть-чуть возросла!

Пример 2. Зная вероятности появления различных болезней и вероятности отдельных симптомов при той или иной болезни, врач может проводить анализы, которые уточняют диагностику и постепенно изменяют шансы этих болезней у данного пациента. Пусть при заболевании партикулезом с вероятностью 0.75 развивается монокулярный партикулез (МП), а с вероятностью 0.25 — бинокулярный (БП) *), а реакция Перфора (Pf) положительна с вероятностью 0.8 в случае МП и с вероятностью 0.3 в случае БП. Требуется оценить шансы того, что больной партикулезом болен МП, если у него реакция положительна.

Подставляя числа в формулу Байеса, получаем

$$\mathrm{P}(\text{МП} \mid \mathrm{Pf}^+) = \frac{0.8 \times 0.75}{0.8 \times 0.75 + 0.3 \times 0.25} = \frac{0.6}{0.6 + 0.075} = 0.889,$$

и диагноз сильно склоняется в сторону МП.

3.3. Случайные величины

Часто со случайными событиями связываются некоторые их численные характеристики: количество бракованных деталей, продолжительность безаварийной работы прибора, число ошибок при передаче

*) Не ищите эту болезнь в медицинском справочнике.

файла и т. п. В вероятностных моделях такие характеристики формально появляются как функции от элементарных событий.

Пусть $\Omega = \{\omega\}$ — множество элементарных событий, p_ω — вероятность элементарного события ω. Заданная на Ω функция $f : \Omega \to R^1$ называется *случайной величиной*.

Для задания случайной величины важна не ее зависимость от элементарного события, а лишь набор вероятностей, с которыми случайная величина принимает те или иные значения. Этот набор вероятностей называется *распределением* случайной величины, или ее *законом распределения*. Если у двух случайных величин распределение совпадает, они называются *одинаково распределенными*. Мы рассмотрим способы задания законов распределения и некоторые типичные распределения ниже, но еще до этого будет полезно ввести некоторые понятия и характеристики случайных величин.

Случайные величины ξ и η называются *независимыми*, если независимы события $\{\omega \,|\, \xi(\omega) = a\}$ и $\{\omega \,|\, \eta(\omega) = b\}$ для любых значений a и b. В дальнейшем, записывая множество элементарных событий, удовлетворяющих этому условию, будем просто писать в фигурных скобках это условие.

Таким образом, случайные величины ξ и η называются *независимыми*, если для любых значений a и b независимы события $\{\xi = a\}$ и $\{\eta = b\}$, т. е. если

$$\mathsf{P}(\xi = a, \eta = b) = \mathsf{P}(\xi = a) \times \mathsf{P}(\eta = b).$$

Из этого определения следует, что для независимых случайных величин независимы события $\{\xi \in A\}$ и $\{\eta \in B\}$ с произвольно выбранными множествами их значений A и B.

Легко ввести и определение независимости нескольких случайных величин *в совокупности*.

Предложенные определения нужны как математическое обоснование свойств случайных величин, которые на самом деле вводятся и используются по-другому. В практических применениях (в прикладных вероятностных моделях) мы наблюдаем какую-то величину (например, расстояние между двумя точками, которое измеряется с неизбежными ошибками; запись радиосигнала, который поступает с помехами; количество детей, родившихся в городе в прошлый понедельник) и, создавая для этой величины математическую модель, совсем не думаем о пространстве элементарных событий — оно нам нужно только для объяснений некоторых сложных феноменов.

3.4. Математическое ожидание и дисперсия

Величина

$$\mathsf{E}\xi = \sum_{\omega} \xi(\omega) p_\omega = \sum_{a} a \times \mathsf{P}(\xi = a)$$

называется *математическим ожиданием* случайной величины ξ. Математическое ожидание похоже на центр тяжести: если считать вероятности значений случайной величины массами точек, мы получим в точности ту же формулу.

Из этого определения легко следуют основные свойства математического ожидания.

1) Если $\mathsf{P}(\xi = a) = 1$, то $\mathsf{E}\xi = a$.

2) Если ξ и η — две случайные величины, то $\mathsf{E}(\xi + \eta) = \mathsf{E}\xi + \mathsf{E}\eta$.

3) Если $\eta = b\xi$, где b — константа, то $\mathsf{E}\eta = b\,\mathsf{E}\xi$.

Математическое ожидание квадрата отклонения случайной величины от ее математического ожидания называется *дисперсией* этой случайной величины:

$$\mathsf{D}\xi = \mathsf{E}(\xi - \mathsf{E}\xi)^2.$$

Дисперсия характеризует разброс случайной величины вокруг ее математического ожидания. Продолжая аналогии с механикой, мы увидим, что дисперсия — это момент инерции системы точек.

Часто оказывается полезной такая формула для дисперсии:

$$\mathsf{D}\xi = \mathsf{E}\xi^2 - (\mathsf{E}\xi)^2,$$

которая легко получается, если раскрыть скобки в предыдущей формуле. Попробуйте сами вывести еще такую формулу:

$$\mathsf{D}\xi = \mathsf{E}(\xi - a)^2 - (\mathsf{E}\xi - a)^2.$$

Эта формула была очень удобна при статистических вычислениях, когда их выполняли вручную *).

Вот еще несколько свойств дисперсии.

1) Дисперсия неотрицательна. Если $\mathsf{P}(\xi = a) = 1$, то $\mathsf{D}\xi = 0$.

2) Если $\eta = \xi + a$, где a — константа, то $\mathsf{D}\eta = \mathsf{D}\xi$.

3) Если $\eta = b\xi$, где b — константа, то $\mathsf{D}\eta = b^2\mathsf{D}\xi$. В частности, $\mathsf{D}(-\eta) = \mathsf{D}\eta$.

*) Сейчас многие карманные калькуляторы считают среднее арифметическое и выборочную дисперсию — аналоги математического ожидания и дисперсии для экспериментальных данных. Есть эти функции и в программе Calculator в системе Windows, не говоря уже о программе Excel.

4) Если ξ и η — две независимые случайные величины, то

$$\mathsf{D}(\xi + \eta) = \mathsf{D}\xi + \mathsf{D}\eta.$$

Свойства 1)–3) очевидны. Для доказательства свойства 4) придется сделать некоторые выкладки. Обратите внимание на требование независимости — это свойство существенно отличается от аналогичного свойства математического ожидания. Из определения

$$\begin{aligned}\mathsf{D}(\xi + \eta) &= \mathsf{E}(\xi + \eta - \mathsf{E}(\xi + \eta))^2 = \mathsf{E}(\xi - \mathsf{E}\xi + \eta - \mathsf{E}\eta)^2 = \\ &= \mathsf{E}(\xi - \mathsf{E}\xi)^2 + 2\mathsf{E}((\xi - \mathsf{E}\xi)(\eta - \mathsf{E}\eta)) + \mathsf{E}(\eta - \mathsf{E}\eta)^2 = \\ &= \mathsf{D}\xi + \mathsf{D}\eta + 2(\mathsf{E}\xi\eta - \mathsf{E}\xi\mathsf{E}\eta).\end{aligned}$$

Осталось показать, что $\mathsf{E}\xi\eta - \mathsf{E}\xi\,\mathsf{E}\eta = 0$. Действительно,

$$\begin{aligned}\mathsf{E}\xi\eta &= \sum_{a,b} ab\,\mathsf{P}(\xi = a, \eta = b) = \sum_{a,b} ab\,\mathsf{P}(\xi = a)\,\mathsf{P}(\eta = b) = \\ &= \sum_a a\,\mathsf{P}(\xi = a) \sum_b b\,\mathsf{P}(\eta = b) = \mathsf{E}\xi\,\mathsf{E}\eta.\end{aligned}$$

Корень из дисперсии называется *средним квадратичным отклонением*. Оно используется для оценки масштаба возможного отклонения случайной величины от ее математического ожидания.

Величина $m(\xi, \eta) = \mathsf{E}\xi\eta - \mathsf{E}\xi\mathsf{E}\eta$ может служить показателем зависимости случайных величин ξ и η. Если разделить ее на корень из произведения дисперсий обеих случайных величин (считая обе дисперсии положительными), то получится коэффициент, обозначаемый через

$$\rho(\xi, \eta) = m(\xi, \eta)/\sqrt{\mathsf{D}\xi\mathsf{D}\eta};$$

он всегда лежит между -1 и 1 и называется *коэффициентом корреляции* этих случайных величин. Экстремальные значения достигаются, когда случайные величины связаны линейной зависимостью, $\eta = a\xi$, а знак ρ таков же, как у a. Отметим, что даже при функциональной зависимости случайных величин $\rho(\xi, \eta)$ может быть нулевым.

Упражнение 3.6. Найдите математическое ожидание и дисперсию числа очков, выпадающих при бросании игральной кости. Затем найдите математическое ожидание и дисперсию числа очков при двух и при n бросаниях.

Упражнение 3.7. Натуральные числа a и b записаны в двоичном виде. Одна из единиц от случайного сбоя превратилась в нуль, и в результате произведение $a \times b$ уменьшилось. Найдите математическое ожидание уменьшения.

Упражнение 3.8. Математические ожидания и дисперсии или их статистические аналоги — средние значения и средние квадратические отклонения — широко используются во всевозможных прикладных расчетах, от самых бытовых (нормативы питания, одежды, бытовых и транспортных услуг) и до высокотехнологичных (обеспечение пропускной способности сервера, скорость ветвления в атомном реакторе и др.). Один поучительный пример хочется рассмотреть подробно.

При угрозе холерной эпидемии организуется карантин, в котором задерживается большое число подозреваемых лиц, для каждого из которых необходимо сделать медицинский анализ. Подавляющее большинство анализов обычно дает отрицательный результат — холерный вибрион не обнаружен. Было предложено изменить схему анализа: соединять пробы нескольких человек и делать совместный анализ. Отрицательный результат означает отрицательный результат для каждого из группы, положительный результат приводит к повторному анализу для каждого отдельно.

Пусть вероятность инфицирования равна p, размер группы равен k. Найдите математическое ожидание числа анализов на одного человека.

3.5. Схема Бернулли

Рассмотрим последовательность независимых одинаково распределенных случайных величин $\delta_1, \ldots, \delta_n, \ldots$, каждая из которых принимает два значения: 1 с вероятностью p и 0 с вероятностью $q = 1 - p$. Такая вероятностная схема называется *схемой Бернулли* *).

Случайная величина ξ_n, получающаяся при сложении n таких случайных величин δ_i, имеет распределение, называемое *биномиальным распределением*. Его легко найти: для того чтобы $\xi_n = k$, нужно, чтобы ровно k из случайных величин $\delta_1, \ldots, \delta_n$ принимали значение 1, а остальные должны равняться нулю. Вероятность этого события при фиксированных местах единиц и нулей равна $p^k q^{n-k}$, и если учесть все возможные C_n^k расположений этих мест, то получим

$$\mathsf{P}(\xi_n = k) = C_n^k p^k q^{n-k}$$

— k-й член биномиального разложения $(p + q)^n$.

Математическое ожидание и дисперсию для биномиального распределения можно легко подсчитать и непосредственно, но лучше воспользоваться свойствами математического ожидания и дисперсии,

*) Jacob Bernoulli (1654–1705). Знаменитая семья математиков Бернулли была очень большой. Это Якоб I, самый старший в семье.

в силу которых

$$\mathsf{E}\xi_n = \sum_{i \in 1:n} \mathsf{E}\delta_i, \quad \mathsf{D}\xi_n = \sum_{i \in 1:n} \mathsf{D}\delta_i.$$

Очевидно,

$$\mathsf{E}\delta_i = \mathsf{E}\delta_i^2 = p, \quad \mathsf{D}\delta_i = p - p^2 = pq,$$

и следовательно,

$$\mathsf{E}\xi_n = np, \quad \mathsf{D}\xi_n = npq.$$

3.6. Функции распределения

Пусть задана случайная величина ξ. Для каждого x в множестве элементарных событий определим $A_\xi(x) = \{s \mid s \in S, \xi(s) < x\}$ — множество тех событий, где $\xi(s)$ меньше x. Введем в рассмотрение функцию $F_\xi(x) = \mathsf{P}(A_\xi(x))$. Она называется (интегральной) *функцией распределения* случайной величины ξ и используется как основная форма задания вероятностного поведения случайной величины.

Легко убедиться в том, что функция распределения $F(x)$ обладает следующими свойствами.

1) Ее значения лежат между 0 и 1 (это же вероятности!).

2) Функция $F(x)$ неубывающая, так как $A(x) \subset A(y)$ при $x < y$.

В дискретном случае закон распределения чаще задают скачками функции распределения — вероятностями отдельных значений. Однако нужно упомянуть и о другом стандартном случае, когда функция распределения непрерывна и имеет производную. Эта производная называется *плотностью распределения*.

Имеется несколько важных стандартных функций распределения, часто появляющихся в приложениях. Среди них мы встретим и дискретные, и непрерывные.

Пример 1. *Биномиальное распределение.* Рассмотрим две уже встречавшиеся нам случайные величины. Начнем с распределения одного наблюдения в схеме Бернулли — это распределение, сосредоточенное в двух точках: 0 — с вероятностью q и 1 — с вероятностью p. Функция распределения — кусочно-постоянная со скачками высоты q в точке 0 и высоты p в точке 1.

Сумма k таких случайных величин ξ имеет биномиальное распределение, вероятность того, что $\xi = s$, равна $\mathsf{P}(s) = C_k^s p^s q^{k-s}$. Распределение получается более наглядным, если изобразить на графике не интегральную функцию распределения, а сами вероятности (рис. 3.1).

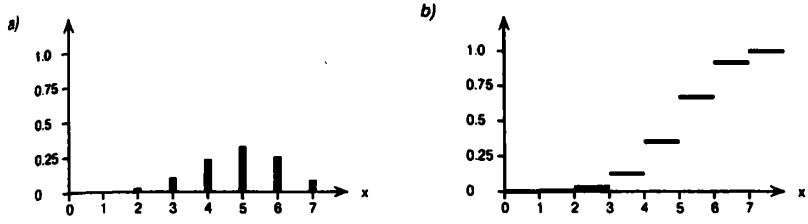


Рис. 3.1. Биномиальное распределение: вероятности (*a*) и функция (*b*)

Для биномиально распределенной случайной величины ξ, как мы уже знаем, $\mathsf{E}\xi = kp$, $\mathsf{D}\xi = kpq$.

Легко догадаться, что если сложить две независимые случайные величины ξ_1 и ξ_2, распределенные биномиально с параметрами (p, k_1) и (p, k_2) соответственно, то сумма будет случайной величиной, распределенной биномиально с параметрами $(p, k_1 + k_2)$.

Пример 2. *Равномерное распределение.* Равномерное распределение на множестве целых чисел $k : l$ — это распределение, в котором все исходы равновероятны, следовательно, $\mathsf{P}(\xi = s) = 1/(l - k + 1)$ при $s \in k : l$ и $F_\xi(x) = 0$ при $x \le k$, $F_\xi(x) = 1$ при $x > l$, $F_\xi(x) = r/(l - k + 1)$ при $r < x \le r + 1$, $k < r < l$.

При увеличении размера области изменения случайной величины, равного $l - k + 1$, отношение шага между значениями к этому размеру уменьшается. В качестве предельной случайной величины рассматривается величина, распределенная равномерно на отрезке. Обычно в качестве этого отрезка принимается отрезок $[0, 1]$. •

Равномерное распределение играет большую роль в программировании как базовое распределение при моделировании случайных наблюдений. Мы коснемся этого вопроса в следующем параграфе и покажем, как из равномерно распределенных случайных величин можно получить случайные величины с другими законами распределения.

Пример 3. *Нормальное распределение.* Это распределение задается формулой

$$F(x, a, \sigma) = \Phi\left(\frac{x - a}{\sigma}\right) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{(z-a)^2}{2\sigma^2}} dz.$$

Параметры a и σ имеют простой смысл: a — математическое ожидание случайной величины с такой функцией распределения, а σ^2 — ее дисперсия. На рис. 3.2 изображено распределение с $a = 0$ и $\sigma = 1$. Программа рисовала этот график в интервале $[-4.0, 4.0]$, но так как $\Phi(3.0) \approx 0.99665$ и $\Phi'(3.0) \approx 0.00433$, то кажется, что кривые обрываются.

Нормальное распределение возникает в схемах, где можно ожидать, что случайная величина получается как сумма многих случайных слагаемых.

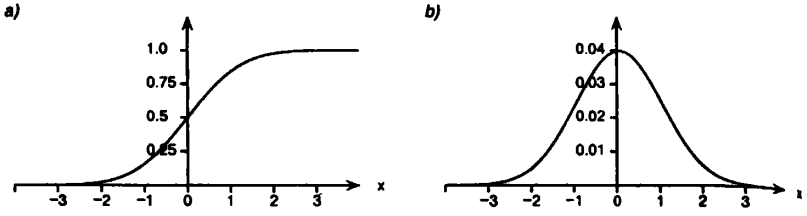


Рис. 3.2. Функция (a) и плотность (b) нормального распределения

Например, рост человека (в *однородной популяции*, где нет смешения различных этнических групп и условия жизни мало меняются от поколения к поколению) имеет нормальное распределение.

Пример 4. *Показательное распределение.* Распределение задается формулой $F(x) = 1 - e^{-\lambda x}$. Единственный параметр λ называется интенсивностью, через него легко выражаются математическое ожидание и дисперсия, $\mathsf{E}\xi = 1/\lambda$, $\mathsf{D}\xi = 1/\lambda^2$.

Показательное распределение устойчиво по отношению к операции взятия минимума: если ξ_1 и ξ_2 — две независимые случайные величины, имеющие показательное распределение с параметрами λ_1 и λ_2, то случайная величина $\eta = \min\{\xi_1, \xi_2\}$ имеет показательное распределение с параметром $\lambda_1 + \lambda_2$. Показательное распределение часто используется в моделях обслуживания (в телефонии, а теперь и в анализе сложных компьютерных систем), где очень удобно следующее его свойство: условное распределение $\xi - z$ при условии, что $\xi > z$, одно и то же при всех z и совпадает с безусловным распределением.

Пример 5. *Распределение Пуассона*[*]. Это распределение дискретно, случайная величина ξ с таким распределением принимает целые неотрицательные значения, и

$$\mathsf{P}(\xi = k) = \frac{\lambda^k}{k!} e^{-\lambda}.$$

Распределение Пуассона, как и биномиальное и нормальное, устойчиво относительно сложения.

Это распределение характерно для схемы *редких событий*: в ситуациях типа схемы Бернулли, если вероятность наступления одного события мала, а самих событий много, число наступивших событий имеет распределение Пуассона. Так, можно ожидать этого распределения для числа ошибок при передаче текста по информационному каналу.

Распределение Пуассона любопытным образом связано с показательным распределением. Рассмотрим последовательность независимых случайных величин ξ_i с показательным распределением с параметром λ. Зададимся неко-

[*] Siméon-Deni Poisson (1781–1840), французский механик, физик и математик. В 1826 г. он был избран почетным членом Петербургской академии наук.

торым положительным T и образуем последовательность накопленных сумм $\eta_k = \sum_{i \le k} \xi_i$. Пусть ζ — наибольший индекс, при котором $\eta_k < T$. Тогда случайная величина ζ имеет распределение Пуассона с параметром λT. В более "практических" терминах это свойство можно пояснить так: если в системе обслуживания промежутки между вызовами имеют показательное распределение, то число вызовов, попавших в любой интервал времени, имеет распределение Пуассона с параметром, пропорциональным длине интервала.

Упражнение 3.9. Случайная величина ξ принимает с равной вероятностью целочисленные значения от 1 до 6. Найти математическое ожидание и функцию распределения случайной величины $\lfloor \xi + 1/\xi \rfloor$.

Упражнение 3.10. Случайная величина η принимает значения от 1 до 5, задана ее (интегральная) функция распределения $F(x)$. Постройте функцию распределения случайной величины $1/\eta$.

Упражнение 3.11. Случайная величина η принимает значения от -1 до 1, задана ее (интегральная) функция распределения $F(x)$. Постройте функцию распределения случайной величины $|\eta|$.

3.7. Случайные числа

Свойства случайных величин можно использовать для создания особой разновидности вычислительных методов, ставших возможными с появлением компьютеров, — методов *статистического моделирования*.

Предположим, что нам удалось создать вероятностную схему, в которой вероятность p некоторого события A равна интересующему нас числу (чтобы создать такую схему, совсем не обязательно знать это число — оно может сложным образом зависеть от известных нам параметров).

Если мы проведем n независимых случайных экспериментов, то, подсчитав количество k осуществлений события A, мы можем принять дробь k/n в качестве хорошего приближения параметра p.

Действительно, отношение k/n — это реализация случайной величины $\eta_n = \xi_n/n$, которая, как это видно из предыдущего параграфа, имеет математическое ожидание p и дисперсию pq/n. При росте n дисперсия приближается к 0, и оценка становится ближе к истинному значению параметра. Вспомним, что масштаб отклонения определяется средним квадратичным отклонением, которое имеет порядок $\sqrt{n}$, так что это убывание не слишком быстрое.

В вычислительных машинах можно имитировать случайные эксперименты. В качестве источника случайности используются специальные программы — *датчики случайных чисел*. Датчик при каждом

обращении к нему вырабатывает некоторое число (обычно целое или вещественное число из фиксированного диапазона), и последовательность этих чисел по своему поведению очень похожа на *последовательность независимых случайных величин, имеющих одинаковое равномерное распределение.*

Например, в качестве такого датчика может быть использована программа *), в которой хранится некоторое число θ с достаточно большой разрядностью. При каждом вызове программа возводит θ в квадрат, увеличивая примерно вдвое число разрядов, и "вырезает" новое значение θ из середины этого результата. Новое значение θ используется и как результат процедуры, и как рабочая информация для следующего вызова процедуры.

Пример. Для вычисления площади $sq(A)$ плоской ограниченной фигуры A можно построить содержащий фигуру прямоугольник R (ориентированный по координатам) и "бросать" в него случайные точки. Отношение числа точек, попавших в A, к общему числу точек будет хорошей оценкой отношения площадей **).

Действительно, при равномерном распределении точек в R вероятность попадания в A равна $p = sq(A)/sq(R)$, а число точек, попавших в A при n попытках, является случайной величиной ξ, имеющей биномиальное распределение с параметрами p и n. Ее математическое ожидание равно np, а дисперсия $np(1-p)$. Для интересующего нас отношения, очевидно, получаем математическое ожидание p и дисперсию $p(1-p)/n$, так что с ростом n отклонения оценки от истинного значения уменьшаются.

Для того чтобы создать необходимое распределение вероятностей, достаточно иметь последовательность независимых случайных величин типа "бросаний монеты", т. е. величин, принимающих значения 0 и 1 с вероятностью $\frac{1}{2}$.

Например, для создания схемы с двумя исходами A_1 и A_2:

$$\mathsf{P}(A_1) = \frac{3}{4}, \qquad \mathsf{P}(A_2) = \frac{1}{4}$$

можно из *датчика случайных двоичных разрядов* получить два двоичных разряда δ_1 и δ_2 и, например, при $\delta_1 = \delta_2 = 1$ выработать исход A_2, а в остальных случаях A_1.

*) Этот способ далеко не лучший. Вы им не пользуйтесь.

**) Мне справедливо заметили, что здесь нужно предполагать, что фигура имеет площадь. Само собой! Но этого мало. Нужно еще предполагать, что проверка попадания очередной точки в фигуру выполняется достаточно просто.

Аналогично для схемы с четырьмя исходами

$$P(A_1) = \frac{3}{16}, \quad P(A_2) = \frac{1}{16}, \quad P(A_3) = \frac{8}{16}, \quad P(A_4) = \frac{4}{16}$$

можно получить четыре двоичных разряда δ_1, δ_2, δ_3, δ_4 и любым способом сопоставить трем из 16 возможных наборов исход A_1, одному — A_2, восьми — A_3, четырем — A_4.

Схему моделирования можно сделать более экономной, если принимать решения *последовательно* и в некоторых случаях прекращать вычисления досрочно, когда результат уже понятен. Так, например, если связать с A_3 восемь наборов, соответствующих $\delta_1 = 1$, то при $\delta_1 = 1$ нет необходимости вырабатывать и анализировать остальные три разряда. Аналогично, при $\delta_1 = 0$ и $\delta_2 = 1$ (четыре набора) можно сделать выбор A_4, при $\delta_1 = 0$, $\delta_2 = 0$, $\delta_3 = 1$ (два набора) возможен выбор A_1. Этот последний выбор не исчерпывает всего, что требуется для A_1, но деление двух оставшихся наборов между A_1 и A_2 завершает формирование схемы.

Нашу схему легко представить графически. Изображенная на рис. 3.3 фигура похожа на дерево и называется *двоичным деревом*. В ней самый верхний кружок называется *корневой вершиной*, нижние, маленькие сплошные, кружки называются *листьями*, а все остальные кружки — *узлами* или *промежуточными вершинами*. Это дерево называется двоичным потому, что из корня и из каждого узла выходит по две стрелки в другие вершины. Мы еще много раз будем возвращаться к деревьям.

Количество случайных двоичных разрядов λ, которые необходимы для формирования случайного исхода, — само случайная величина.

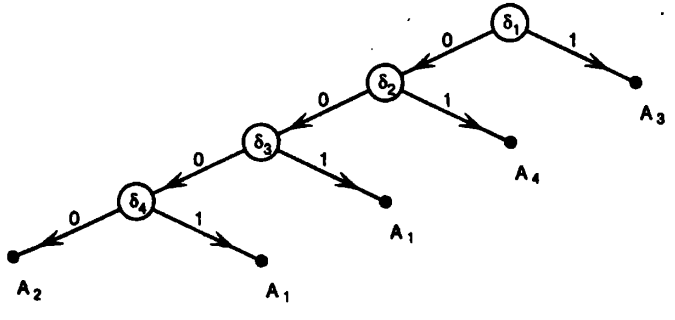


Рис. 3.3. Схема ветвления

Легко подсчитать ее математическое ожидание

$$E\lambda = \frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \frac{1}{8}\cdot 3 + \frac{1}{16}\cdot 4 + \frac{1}{16}\cdot 4 = 1\frac{7}{8}.$$

А что делать, если вероятности исходов не кратны 2^{-k}? Есть несколько вариантов действий.

По первому варианту можно приблизить вероятности двоичными дробями (это можно сделать с любой точностью), а дальше работать с этими приближенными значениями.

Второй вариант больше соответствует вероятностному подходу и способу рассуждений. Пусть все вероятности n_i — дроби со знаменателем r. Найдем k, для которого $r < 2^k$. Теперь предложим схему с k двоичными разрядами, в которой r наборов используются для выработки случайного исхода, а остальные $2^k - r$ наборов объявляются "неудачными" и требуют повторного эксперимента (до тех пор, пока не встретится удачный). Конечно, чем выше доля полезных исходов, равная $r2^{-k}$, тем эта схема будет эффективнее.

Третий вариант использует свойство датчиков случайных чисел: они формируют не отдельные двоичные разряды, а целые наборы их, например в виде числа, распределенного равномерно в [0, 1]. Образуем по данному набору вероятностей p_i *накопленные суммы* s_i: $s_0 = 0$; $s_i = s_{i-1} + p_i$, $i > 0$. Случайный исход будет вырабатываться так: по полученному из датчика случайному числу γ определяется такой индекс i, для которого $s_{i-1} < \gamma \le s_i$. Найденное значение индекса i и определяет исход A_i.

Индекс i можно определять непосредственно просмотром s_i подряд. В случае если k велико, можно применять специальные приемы ускоренного поиска (например, деление множества индексов примерно пополам, мы вернемся к этому вопросу ниже).

Но, пожалуй, самую эффективную схему предложил в 1977 г. А. Уолкер*), и ее полезно знать как пример для подражания.

Идея Уолкера достаточно хорошо видна даже на небольшом примере. Прежде всего заметим, что, если бы все исходы имели одинаковые вероятности, моделировать такое распределение было бы очень просто. В такой ситуации достаточно разделить отрезок [0, 1] на k одинаковых частей, соответствующих этим исходам, и определить, в какую часть отрезка попало значение случайного датчика x. А это выясняется очень просто: нужно взять целую часть произведения $k \cdot x$. Так

*) Alastair Walker [85].

что при исходах 0, 1, 2, 3 значению $x = 0.333$ соответствует исход 1, поскольку $4x = 1.332$.

Можно "подгонять" распределение под равномерное, передавая часть "вероятностной массы" от одних исходов другим. Скажем, если те же четыре исхода должны иметь вероятности, соответственно, 0.25, 0.31, 0.19 и 0.25, то исход 2, получая, как все другие, долю 0.25 и нуждаясь в доле 0.19, может быть своеобразным "донором" и отдать исходу 1, нуждающемуся в доле 0.31, свои лишние шесть сотых. Эта передача воплощается в следующих действиях при генерировании случайного исхода: берется случайное число, например можно использовать непосредственно дробную часть числа $k \cdot x$, и если это число меньше чем $0.19 \cdot 4 = 0.76$, то результатом будет исход 2, а если больше, то исход 1 (здесь случай равенства несуществен, его можно приписать к любой альтернативе). Эта передача обеспечит нужное увеличение доли для исхода 1 и уменьшение — для исхода 2. Исход 1 служит здесь "реципиентом" для "донора", исхода 2.

Такую передачу "вероятностной массы" можно проводить в каждом диапазоне, причем если "владельцами диапазонов" удобно назначать различные исходы, то реципиентами разных диапазонов могут быть одни и те же исходы. Например, взяв ту же схему с четырьмя исходами, назначим реципиентом при исходе 0 исход 3 с долей 0.07, при исходе 1 — исход 2 с долей 0.11, при исходе 2 — исход 3 с долей 0.14, наконец, при исходе 3 — исход 1 с долей 0. Подсчитаем окончательную вероятность получения каждого исхода:

$$\begin{aligned} p_0 &= 0.25 - 0.07 &&= 0.18, \\ p_1 &= 0.25 - 0.11 + 0 &&= 0.14, \\ p_2 &= 0.25 - 0.14 + 0.11 &&= 0.22, \\ p_3 &= 0.25 + 0.07 + 0.14 &&= 0.46. \end{aligned}$$

А создание по этой схеме очередного значения будет таким же простым, как и раньше: только в каждом диапазоне нужно будет решать, какой выбирать исход, основной или дополнительный. Сложность розыгрыша не зависит от количества исходов.

Оказывается, такую схему можно создать для *любого* распределения вероятностей, причем у каждого исхода-донора будет только один исход-реципиент. Схема создается последовательно. На каждом шаге выбирается донор, не полностью использующий свою долю, и реципиент, которому требуется добавка. Конечно, после каждой передачи нужно исключить из рассмотрения донора и уменьшить долю реципиента, так что на следующих итерациях он может сам стать донором.

Пример. Возьмем распределение $p_A = 0.07$, $p_B = 0.31$, $p_C = 0.35$ и $p_D = 0.27$. Для него мы можем получить:

Донор	Барьер	Реципиент	Остаток у реципиента
A	$0.07 \cdot 4$	B	$0.31 - 0.18 = 0.13$
B	$0.13 \cdot 4$	D	$0.27 - 0.12 = 0.15$
D	$0.15 \cdot 4$	C	$0.35 - 0.10 = 0.25$
C	$0.25 \cdot 4$	A	0

Далее при моделировании эти числа используются так: разыгрывается случайное число $x \in [0, 1]$, пусть получилось $x = 0.531$. Это число умножается на k: $0.531 \times 4 = 2.124$. Целая часть произведения определяет строку таблицы, считая от нуля, значит, третью сверху строку с основным исходом D и дополнительным C. Дробная часть произведения 0.124 сравнивается с барьером 0.6, и так как барьер выше, то принимается основной исход.

3.8. Двоичный поиск и неравенство Крафта

При моделировании дискретных распределений мы встретили схему *дихотомического поиска*: разыскивая на прямой линии отрезок, соответствующий нужному значению индекса, разбивали область поиска на две части и устанавливали, в какой из частей лежит интересующий нас индекс. Затем, взяв в качестве новой зоны поиска оставшуюся часть, повторяли это действие до тех пор, пока не получили часть, содержащую только одно значение индекса.

При реализации этой схемы результат достигается на разных разбиениях зоны поиска. Так, могут использоваться разбиения, в которых на каждом шаге отщепляется отрезок, соответствующий только одному значению индекса. Это случай последовательного перебора возможностей. Можно делить множество индексов на каждом шаге и более решительно, например примерно пополам.

Возникает задача о выборе такой последовательности разбиений единичного отрезка, при которой затраты на поиск будут наименьшими. Главной частью в постановке этой задачи является критерий оценки — числовая характеристика, по которой нужно оценивать качество разбиений.

Таким критерием могло бы быть, например, максимальное количество шагов при поиске, но при вероятностном подходе и необходимости выполнять поиск многократно более правильным критерием качества представляется *математическое ожидание затрат на поиск*. Посмотрим, как его записать формально.

В схеме дихотомического поиска если в каждом разбиении сопоставлять получающимся альтернативам значения 0 и 1, то каждому из окончательных индексов i будет сопоставлен *путь поиска* — строка из нулей и единиц, описывающая последовательность выборов. Длина этого пути s_i может служить хорошей оценкой усилий поиска.

В вероятностной схеме появления индексов, когда исход i появляется в результате случайного испытания с вероятностью p_i, длина пути поиска становится случайной величиной σ, математическое ожидание которой равно $\sum_i p_i s_i$. Обратите внимание на то, что мы не имеем права говорить, что вероятность значения s_i у случайной величины σ равна p_i. Почему? Потому, что несколько исходов могут дать одно и то же значение длины.

Итак, получилась такая экстремальная задача: найти схему поиска, при которой математическое ожидание σ было бы минимальным. Очевидно, что при анализе схемы поиска нам достаточно определять не сам набор путей поиска, а только набор длин и его сопоставление различным исходам i. Оказывается, легко можно определить необходимые и достаточные условия, которым должен удовлетворять набор натуральных чисел для того, чтобы он был набором длин путей в некоторой схеме дихотомического поиска.

Теорема (неравенство Крафта[*]). Для того чтобы набор целых чисел s_i, $i \in 1:m$, мог быть набором длин путей поиска в схеме с m исходами, необходимо и достаточно, чтобы

$$\sum_{i \in 1:m} 2^{-s_i} \leq 1.$$

Доказательство. Необходимость. Рассмотрим поисковую схему, которая представляет собой двоичное дерево T с m листьями. Сопоставим каждой вершине k этого дерева, находящейся на расстоянии t от корня, число $a_k = 2^{-t}$. Очевидно, что для корня r_0 мы имеем $a_{r_0} = 1$, так что требуется доказать такое неравенство:

$$a_{r_0} \geq \sum_{k \in F} a_k,$$

где F — множество листьев. Для каждого не-листа $k \in M \setminus F$ имеет место неравенство:

$$a_k \geq \sum_{r \in \mathrm{next}(k)} a_r, \qquad (1)$$

[*] L. G. Kraft. Thesis of Master of Sci. MIT, 1944. См. книгу Ягломов [52].

где next(k) — множество вершин, непосредственно следующих за k. Это множество состоит из одной или из двух вершин. В случае двух вершин неравенство выполняется как равенство, в случае одной вершины — как строгое неравенство. Суммируя неравенства (1) по $M \setminus F$, мы получаем неравенство

$$\sum_{M\setminus F} a_k \geq \sum_{M\setminus\{r_0\}} a_r. \tag{2}$$

Сокращение общих слагаемых (промежуточных вершин) в левой и правой частях (2) и дает искомое неравенство. Отметим, что неравенство выполняется как равенство только в случае, когда множество next(k) каждой вершины k состоит из двух вершин.

Достаточность. Здесь будет использована совсем другая техника. Взяв m чисел s_k, удовлетворяющих условию теоремы, расположим их в порядке возрастания (будем считать, что это упорядочение уже выполнено). Определим, как и раньше, числа $a_k = 2^{-s_k}$ и положим

$$b_1 = 0; \quad b_{k+1} = b_k + a_k, \quad k > 1.$$

Рассмотрим последовательности из нулей и единиц t_k, являющиеся двоичными представлениями дробей b_k, в каждой такой дроби b_k возьмем первые s_k знаков. Покажем, что никакая из последовательностей t_k не является началом другой такой последовательности.

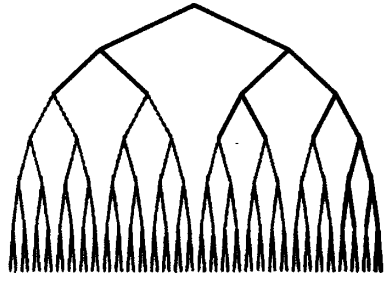
Так как длины последовательностей t_k не убывают с ростом k, нам достаточно показать, что никакая последовательность t_k не является началом последовательности с бо́льшим номером. Так как дроби b_i возрастают, то $b_k < b_{k+1} < \cdots < b_m \leq 1$. Обозначим через $b_r^{(k)}$ число, получающееся из b_r, отсечением первых s_k цифр. Очевидно, что для таких "урезаний" сохраняется то же неравенство, хотя и нестрогое: $b_k < b_{k+1}^{(k)} \leq \cdots \leq b_m^{(k)}$. При этом первое из неравенств осталось строгим, так как $b_k + a_k = b_{k+1} = b_{k+1}^{(k)}$. Следовательно, начало $t_r^{(k)}$ никакой дроби b_r не совпадает с t_k при $r > k$.

Далее, любой набор последовательностей t_k, из которых ни одна не является началом другой, задает некоторую процедуру поиска. Действительно, рассмотрим полное двоичное дерево с достаточно большим числом этажей и наложим на это дерево все последовательности t_k, трактуя каждую из них как путь от корня, который идет влево, если очередной элемент последовательности равен 0, и вправо —

для единицы. В силу сделанного предположения ни одна из вершин, соответствующих концам последовательностей, не лежит на какой-нибудь другой последовательности как промежуточная вершина. □

Пример. Посмотрите конструкцию, использованную в доказательстве достаточности, на примере. Вся информация собрана в таблицу. Рядом на полном дереве с шестью этажами изображены черными линиями все последовательности t_k.

k	s_k	a_k	b_k	t_k
1	2	.010000	.000000	00
2	2	.010000	.010000	01
3	3	.001000	.100000	100
4	3	.001000	.101000	101
5	3	.001000	.110000	110
6	5	.000010	.111000	11100
7	5	.000010	.111010	11101
8	5	.000010	.111100	11110
9	6	.000001	.111110	111110
10	6	.000001	.111111	111111



Этот пример подсказывает нам другое доказательство достаточности в теореме. Фактически оно совпадает с изложенным, но выглядит гораздо проще.

Возьмем упорядоченный по неубыванию набор длин $\{s_k\}$ (в примере это набор $\{2, 2, 3, 3, 3, 5, 5, 5, 6, 6\}$) и соответствующий им набор $\{a_k\} = \left\{\frac{1}{4}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{32}, \frac{1}{32}, \frac{1}{32}, \frac{1}{64}, \frac{1}{64}\right\}$. При построении дерева мы распределяем сопоставленный вершине вес 1 по весам нижестоящих вершин, где листьям соответствует набор $\{a_k\}$, используя два действия: раздачу весов и разбиение нерозданных весов пополам.

Начальный вес 1 "никому не требуется" — его надо разбить на две половинные части $\left\{\frac{1}{2}, \frac{1}{2}\right\}$. Они также не требуются — разобьем и их, получая $\left\{\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right\}$. В наборе $\{a_k\}$ есть две такие доли — мы их отдаем, а оставшиеся делим пополам. Получилось

$$\left\{\frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}\right\}, \qquad \left\{\frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{32}, \frac{1}{32}, \frac{1}{32}, \frac{1}{64}, \frac{1}{64}\right\}$$

(слева — "предложение", справа — "спрос"). Три доли отдаются, одна делится пополам:

$$\left\{\frac{1}{16}, \frac{1}{16}\right\}, \qquad \left\{\frac{1}{32}, \frac{1}{32}, \frac{1}{32}, \frac{1}{64}, \frac{1}{64}\right\}.$$

Оставшиеся доли снова делятся:

$$\left\{\tfrac{1}{32}, \tfrac{1}{32}, \tfrac{1}{32}, \tfrac{1}{32}\right\}, \qquad \left\{\tfrac{1}{32}, \tfrac{1}{32}, \tfrac{1}{32}, \tfrac{1}{64}, \tfrac{1}{64}\right\}.$$

Затем снова три доли отдаются, а одна делится:

$$\left\{\tfrac{1}{64}, \tfrac{1}{64}\right\}, \qquad \left\{\tfrac{1}{64}, \tfrac{1}{64}\right\}.$$

Наконец, обе доли отдаются — и весь вес распределен (часть могла остаться нераспределенной), а набор $\{a_k\}$ исчерпан. Очевидно, что этот процесс приводит к полному распределению (и построению двоичного дерева) при любом допустимом наборе $\{a_k\}$.

Двоичное дерево, которое состоит только из вершин и дуг, входящих в пути, соответствующие последовательностям t_k, и является деревом поиска. Отметим, что если неравенство выполняется как строгое, то дерево будет с "лакунами" — из некоторых промежуточных вершин будет выходить только по одной дуге.

Теперь можно обратиться к вопросу о минимизации математического ожидания. В силу доказанной теоремы набор путей поиска может описываться набором их длин, которые должны удовлетворять условию Крафта. Получается следующая экстремальная задача.

Задача о наилучших длинах кодов. Пусть задан набор вероятностей p_i, $i \in 1:m$. Найти набор чисел s_i, $i \in 1:m$, удовлетворяющий условиям: 1) $\sum_{i\in 1:m} 2^{-s_i} \le 1$; 2) $s_i > 0$, $i \in 1:m$; 3) s_i — целые и минимизирующий значение $\sigma(p, s) = \sum_{i\in 1:m} p_i s_i$. ◻

Обозначим искомый минимум через $S(p)$. Наряду с этой задачей рассмотрим упрощенную задачу, в которой не требуется выполнения условия 3). Минимум в этой второй задаче обозначим через $H(p)$. Так как смягчение условий расширяет возможности выбора минимизирующей точки, то $S(p) \ge H(p)$.

Теорема. $H(p) = \sum_{i\in 1:m} p_i \log_2 \frac{1}{p_i}$.

Доказательство. Обозначим $y_i = 2^{-s_i}$, так что $s_i = \log_2 \frac{1}{y_i}$. В этих обозначениях условие 1) преобразуется к виду:

$$\sum_{i\in 1:m} y_i \le 1, \qquad (*)$$

а целевая функция — к виду $T(p, y) = \sum_{i\in 1:m} p_i \log_2 \frac{1}{y_i}$. Условие 2) перейдет в условие $0 < y_i < 1$. Второе неравенство из этой пары следует из только что полученного ограничения на сумму переменных y_i, так что останется только условие положительности.

Целевая функция T состоит из отдельных слагаемых, соответствующих отдельным переменным. Каждое слагаемое убывает с ростом аргумента. Если бы неравенство $(*)$ выполнялось как строгое, то увеличение любой из переменных уменьшило бы значение целевой функции. Поэтому в точке минимума условие 1) выполняется как равенство. Выразим переменную y_m через остальные переменные:

$$y_m = 1 - \sum_{i \in 1:m-1} y_i,$$

подставим ее в целевую функцию и приравняем нулю производные целевой функции $T(p, y)$ по всем оставшимся переменным:

$$\frac{\partial T}{\partial y_i} = -p_i \cdot \log_2 e \cdot \frac{1}{y_i} + p_m \cdot \log_2 e \cdot \frac{1}{y_m},$$

откуда

$$\frac{p_1}{y_1} = \frac{p_2}{y_2} = \dots = \frac{p_m}{y_m}.$$

Так как обе суммы (p_i и y_i) равны 1, то $y_i = p_i$. Это единственная точка, в которой возможен экстремум функции $T(p, y)$, и следовательно, $\min_y T(p, y) = H(p)$. □

Предложенное доказательство пока "не по зубам" первокурсникам. Здесь используются строгая выпуклость функции $T(p, y)$, гарантирующая совпадение локального минимума с глобальным, и условие локального минимума функции нескольких переменных на ограниченном множестве. Но в курсе математического анализа всему этому обязательно научат.

3.9. Энтропия и ее свойства

В некоторых ситуациях нет оснований сопоставлять вероятностным схемам какие-либо числовые характеристики. Например, если мы изучаем вероятностные свойства текстов, то сопоставлять буквам алфавита их номера или какие-либо другие числа было бы неправильно. Однако хочется сравнивать вероятностные схемы по содержащейся в них неопределенности, и для этой неопределенности было бы удобно иметь какую-нибудь численную оценку. Такая мера неопределенности еще до математики вводилась в статистической физике, где рассматриваются модели вероятностного характера. Например, модели, описывающие поведение комплексов, состоящих из отдельных частиц, в частности газов.

Функция, которая появилась у нас в предыдущем параграфе в качестве оценки для числа экспериментов при поиске, обладает, оказывается, всеми требуемыми свойствами.

Энтропией[*)] случайной схемы называется мера содержащейся в этой схеме неопределенности. Она задается как вполне конкретная функция от вероятностей исходов, и можно, конечно, начать с ее явного вида. Если мы имеем вероятностную схему $\mathscr{P}$ с m исходами и вероятности этих исходов равны $p_1, p_2, \ldots, p_m$, то энтропия этой схемы определяется как[**)]

$$H(\{p_1, p_2, \ldots, p_m\}) = \sum_{i=1}^{m} p_i \log_2 \frac{1}{p_i}.$$

Оказывается, что свойства, которыми обладает функция H и которые представляются совершенно естественными для характеристики неопределенности, на самом деле предопределяют этот конкретный вид функции. Рассмотрим теперь эти свойства.

1) Естественно предполагать, что при фиксированном m мера неопределенности должна непрерывно зависеть от вероятностей. Функция H этим свойством, разумеется, обладает.

2) При перестановке вероятностей в наборе $\{p_1, \ldots, p_m\}$ мера неопределенности не изменяется.

3) Техническое требование: нужно ввести шкалу для измерения неопределенности. Примем за единицу неопределенность вероят-

[*)] Слово entropy образовано от греческой частицы *εν* (в) и *τροπη* (преобразование) по аналогии со словом *энергия*, производимым от *εν* и *εργον* (работа). В 1865 г. это слово придумал немецкий физик Р. Клаузиус (1822–1888), один из создателей термодинамики, выбирая термин для введенной им *количественной меры преобразуемости* тела. Он писал (очень красиво): "Если мы хотим дать название величине S, мы можем сказать, что она представляет собой содержащееся в теле преобразование, так же как мы говорим, что величина U представляет собой содержащиеся в теле тепло и работу. Однако, поскольку я думаю, что имена для таких величин, важных для науки, лучше выбирать из древних языков, чтобы они вводились без изменения во все современные языки, я предложил назвать величину S энтропией тела, от греческого слова *преобразование*. Я намеренно выбрал слово *энтропия*, которое так похоже на слово *энергия*, так как обе величины, которые должны стать известными под этими именами, настолько близки друг другу по их физическому значению, что определенное сходство их названий кажется мне желательным". В информатике этот термин по совету Дж. фон Неймана ввел в обиход Клод Шеннон (1916–2001), выдающийся американский математик и инженер, один из создателей теории информации.

[**)] Вам здесь может не понравиться, что мы вводим функцию от *переменного* числа аргументов, так как, конечно, число исходов m будет изменяться. Но эту функцию нужно рассматривать как $H(\mathscr{P})$ — функцию от вероятностной схемы, заданную на множестве всех возможных схем.

ностной схемы с двумя равновероятными исходами *) Отметим, что $H(\{\frac{1}{2}, \frac{1}{2}\}) = 1$.

4) При фиксированном m наибольшей неопределенностью обладает схема, в которой все исходы равновероятны. Для функции H это требование выполняется, что легко проверить дифференцированием. Обозначим $H(\{\frac{1}{m}, \ldots, \frac{1}{m}\}) = h(m)$.

5) Функция $h(m)$ с ростом m возрастает. Это условие может быть ослаблено (см. книгу А. М. и И. М. Ягломов [52]), но мы считаем, что оно не должно вызвать возражений — предположение, что с ростом числа исходов неопределенность растет, вполне естественно.

6) (Самое важное свойство.) Рассмотрим схему $\mathscr{P}_m$ с m исходами и вероятностями $\{p_1, \ldots, p_m\}$ и схему $\mathscr{R}_k$ с k исходами и вероятностями $\{q_1, \ldots, q_k\}$. Образуем комбинированную схему с $m + k - 1$ исходами следующим образом: выбирается случайным образом один из исходов схемы $\mathscr{P}_m$, и если произошел m-й исход, выбирается случайно один из исходов схемы $\mathscr{R}_k$, а остальные $m - 1$ исходов схемы $\mathscr{P}_m$ считаются окончательными. В этой комбинированной схеме $\mathscr{P}\mathscr{R}$ мы получаем исходы

$$1, 2, \ldots, m-1, (m, 1), (m, 2), \ldots, (m, k)$$

с вероятностями

$$p_1, p_2, \ldots, p_{m-1}, p_m q_1, p_m q_2, \ldots, p_m q_k.$$

Легко видеть, что $H(\mathscr{P}\mathscr{R}) = H(\mathscr{P}_m) + p_m H(\mathscr{R}_k)$. Потребуем выполнения этого свойства для любой меры неопределенности.

Теорема. Единственная функция на множестве всех вероятностных схем, удовлетворяющая условиям 1)–6), — это функция H.

Доказательство. Пусть функция G определена для всех вероятностных схем и удовлетворяет условиям 1)–6). Положим

$$G\left(\left\{\frac{1}{m}, \ldots, \frac{1}{m}\right\}\right) = g(m).$$

Почти все, что требуется, доказывается в двух простых леммах.

*) Для этой единицы вполне подходит название *бит*, так что энтропия измеряется в битах. В битах же измеряется и *информация*, определяемая как уменьшение энтропии в результате сообщения. Биты энтропии и биты памяти меряют принципиально разные вещи, но это не должно вызвать трудности.

Лемма 1. $g(m) = \log_2 m$.

Доказательство. Пусть имеются две схемы с равновероятными исходами: $\mathscr{A}_k$ и $\mathscr{A}_l$. Составим из них комбинированную схему $\mathscr{A}_{kl}$ с kl равновероятными исходами:

$$(1, 1),\ (1, 2),\ \ldots,\ (1, l),\ (2, 1),\ \ldots,\ (2, l),\ \ldots,\ (k, 1),\ \ldots,\ (k, l).$$

Эта схема получается последовательным присоединением к схеме $\mathscr{A}_k$ схемы $\mathscr{A}_l$ (k раз, в продолжение каждого из исходов схемы $\mathscr{A}_k$). Из свойства 6) получаем $g(kl) = g(k) + g(l)$.

Отсюда, в частности, следует, что $g(m^k) = k \cdot g(m)$ и $g(2^k) = k$ для всех k в силу свойства 3). Далее обозначим через s целую часть величины $\log_2 m^k$, ввиду монотонности g имеем

$$g(2^s) \le g(m^k) \le g(2^{s+1}),$$

или

$$s \le kg(m) \le s + 1.$$

После очевидных преобразований получаем

$$0 \le g(m) - \frac{\lfloor k \log_2 m \rfloor}{k} = g(m) - \log_2 m + \frac{\{k \log_2 m\}}{k} \le \frac{1}{k},$$

где (только один раз) фигурные скобки обозначают дробную часть числа. Это неравенство справедливо для любого k. Ясно, что, устремляя $k \to \infty$, мы получаем утверждение леммы. □

Лемма 2. Если набор вероятностей $\{p_1, \ldots, p_m\}$ состоит из рациональных чисел, то $G(\{p_1, \ldots, p_m\}) = H(\{p_1, \ldots, p_m\})$.

Доказательство. Пусть все $p_i = \frac{r_i}{n}$. Комбинируя схему $\mathscr{P}_m$ со схемами $\mathscr{A}_{r_i}$, мы получим схему $\mathscr{A}_n$ с n равновероятными исходами. Из свойства 6) имеем $G(\mathscr{A}_n) = G(\mathscr{P}_m) + \sum_{i \in 1:m} p_i G(\mathscr{A}_{r_i})$ или по лемме 1

$$G(\mathscr{P}_m) = \log_2 n - \sum_{i \in 1:m} p_i \log_2 r_i = \sum_{i \in 1:m} p_i \log_2 \frac{1}{p_i}.$$ □

Завершение доказательства теоремы. Остается воспользоваться свойством непрерывности функции G. Для любого набора вероятностей $\{p_1, \ldots, p_m\}$ рассмотрим сходящуюся к нему последовательность рациональных наборов $\{p_1^{(k)}, \ldots, p_m^{(k)}\}$. По лемме 2 для

каждого из этих наборов $G = H$. Так как обе функции непрерывны, то это равенство выполняется и в предельной точке. □

Нарисовать график энтропии сложно — это функция от многих переменных, но некоторое представление о ней можно получить по графикам функций $H(\{p, p', \cdot, p'\})$, в которых все аргументы от p_2 до p_n одинаковы. Четыре такие функции изображены на рис. 3.4.

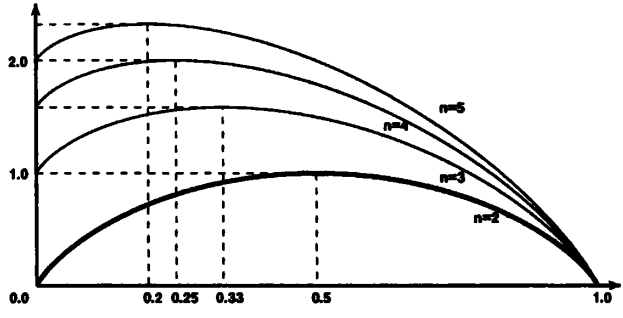


Рис. 3.4. Двумерные графики энтропии

Энтропия оказывается естественным мерилом объема работы в задачах информационного поиска. Теорема, полученная в предыдущем параграфе, показывает, что если проводить эксперименты, в каждом из которых возможны два исхода, каждый с вероятностью $\frac{1}{2}$, то среднее количество экспериментов, необходимых для установления результата случайного испытания с данным априорным распределением вероятностей, не может быть меньше энтропии этого распределения.

Глава 4

Строки переменной длины

4.1. Строки, списки, последовательности

Во многих разделах математики (неожиданно во многих) в качестве объекта исследования рассматривается конечная последовательность однотипных элементов. В отличие от многомерных векторов (или, в программистском понимании, массивов) в таких случаях длина последовательности может варьироваться, и в операциях с последовательностью доступ к элементу по номеру считается либо невозможным, либо затруднительным.

В качестве примеров таких последовательностей можно назвать набор коэффициентов многочлена в математическом анализе, последовательность вершин многогранника или контура в геометрии, список ненулевых элементов разреженного вектора или матрицы в линейной алгебре, цепной список в программировании, слово, строку текста, сам текст или последовательность текстов во многих разделах современной информатики.

В чем различие понятий "строка", "последовательность", "цепной список"? Во-первых, в подразумеваемых деталях их представления, во-вторых, в разрешенных действиях.

Говоря о *последовательности*, мы интересуемся прежде всего порядком в некотором множестве объектов. Можно сказать так: последовательность — это отображение отрезка натурального ряда $1:k$ в некоторое множество A; это отображение создает набор *нумерованных* элементов A: $a_1, a_2, \ldots, a_k$, в котором (в отличие от множества) один и тот же элемент может появиться несколько раз.

Цепной список — понятие сугубо программистское. Это способ организации множественных данных, состоящих из *последовательности* однотипных элементов, причем для организации порядка этих элементов используются *связи* между этими элементами: *односторонние*, если каждый элемент "помнит" следующий за ним (а последний помнит, что он последний), или *двусторонние*, если каждый элемент помнит и следующий за ним, и предшествующий элемент. Цепные списки очень популярны в программировании и часто будут встречаться в этом курсе. У нас нет возможности останавливаться на многочисленных их разновидностях.

Строка занимает промежуточное место между "чисто математической" последовательностью и "чисто программистским" цепным списком. Правда, у строки есть сугубо прикладной смысл — "строка текста". В таком понимании строка математически представляет собой последовательность некоторых элементов, "букв", а программистски — отрезок памяти, "одномерный массив". Для строк характерно сочетание свойств массива, например доступ к букве по ее номеру в последовательности, и свойств цепного списка, например возможность изменения ее размера *).

Мы будем пользоваться в основном термином "строка" как самым коротким и наиболее подходящим, а требования к объекту будут меняться по мере изложения.

Начнем с задач, для которых характерно использование рекурсии. Рекурсия основана на возможности разделения строки на первый элемент и остаток, который рассматривается как такая же строка, но меньшей длины.

Функции и операции, определяемые для таких объектов, обладают некоторой спецификой, которую мы и намерены здесь обсудить. Начнем с определений, относящихся к строкам из элементов произвольного типа. Несмотря на эту произвольность, называть мы их будем для простоты буквами.

4.2. Операции над строками

Итак, пусть задано некоторое конечное множество A, которое будет называться *алфавитом*, а его элементы — *буквами*. Произвольная конечная последовательность букв называется *строкой*, а количество

*) В очень интересной книге [13], целиком посвященной "строковой математике", связанной с молекулярной биологией, отмечается поучительное различие строк и последовательностей: смысл терминов *подстрока* и *подпоследовательность* совершенно различен, *подстрока* — это сплошной фрагмент строки, а *подпоследовательность* — произвольный набор элементов с монотонным возрастанием номеров.

букв в этой последовательности — *длиной строки*. Последовательность нулевой длины называется *пустой строкой*.

1. Нахождение длины строки. Самая простая операция. Ее трудоемкость существенно зависит от формы представления строки в данном алгоритме или программной системе: в одних случаях длина записана в данных о строке явно, и тогда трудоемкость, как говорят, "константна", а в других ее приходится считать, и трудоемкость "линейна" — пропорциональна самой длине.

В некоторых случаях приходится определять *ширину* строки как сумму ширин букв. Пришлось взять другой термин, так как сменилось приложение: ширина строки нужна при ее изображении на бумаге или на экране.

2. Выделение подстроки. Операция выделяет подстроку с заданным числом букв начиная с заданного места (она так важна, что включена почти во все современные языки программирования). Иногда особо определяются операции для выделения подстроки в начале или в конце строки.

Начальная подстрока строки называется ее *префиксом*, а конечная — ее *суффиксом*.

Префикс из одной буквы назовем *головой строки* (head), а оставшуюся часть строки — ее *хвостом* (tail). Операции выделения головы и хвоста строки s обозначим, соответственно, через head s и tail s.

3. Конкатенация строк *). Соединение двух строк в одну стыковкой, одна после другой. Иногда используются термины *катенация* и *сцепка*. Эта операция всем хорошо известна.

4. Обращение строки. Такая естественная операция! Я ее только по причине естественности и упоминаю, но ни разу не встречал ее использования, хотя обращение *цепных списков* использовать приходилось **).

Обращение строки выполняется циклом из последовательных "отцеплений" головы исходного списка и "прицеплений" этого элемента в качестве головы списка-результата.

---

*) От латинского слова catena — цепь.

**) Впрочем, еще можно назвать обращение двоичных разложений целых чисел, встречающееся в так называемом *быстром преобразовании Фурье*.

Студент Г. Корнеев (ИТМО, 1998 г.) назвал мне более естественное приложение этой операции: построение зеркального отражения контура фигуры, когда списки используются для задания кусочно-линейных границ.

5. Сравнение строк по предшествованию. Операция используется для задания упорядочения строк. Как правило, используется *лексикографическое* сравнение строк, основанное на упорядочении букв. Именно, пусть буквы алфавита A можно сравнивать (на алфавите "задано отношение порядка") и про любые две можно сказать, какая из них должна стоять в этом порядке раньше. Используя операцию сравнения букв, мы можем определить результат сравнения строк a и b как

$a = b$, если обе строки пусты,

$a \prec b$, если строка a пустая, а b нет,

$b \prec a$, если строка b пустая, а a нет,

$a \prec b$, если head a предшествует head b,

$b \prec a$, если head b предшествует head a,

результат сравнения tail a и tail b, если head a = head b.

Сформулированное правило сравнения удобно формально, но для "человеческого" восприятия стоит определить его по-другому: чтобы сравнить две строки w_1 и w_2, нужно отбросить у них совпадающие начала и сравнить остатки t_1 и t_2. Если один из остатков пуст, то соответствующая ему строка должна стоять раньше. В противном случае результат сравнения определяется первыми элементами остатков.

Пример. При сравнении слов `колдун` и `колба` отбрасывание совпадающей части оставляет `дун` и `ба`; так как буква `б` предшествует (в русском алфавите) букве `д`, то слово *колба* предшествует слову *колдун*.

При сравнении слов `колбасник` и `колба` отбрасывание совпадающей части оставляет `сник` и пустое слово; соответствующее пустому остатку слово *колба* предшествует слову *колбасник*.

Замечание. Слово "лексикографический" вызывает в памяти словари, лексикографический порядок — такой, как в лексиконе, обычном, привычном для нас, словаре. Нужно предупредить, что упорядочения, используемые в словарях, могут варьироваться в зависимости от целей словаря; в программных системах, где используется лексикографический порядок, такие вариации учитываются, но только частично. Вот несколько причин, из-за которых порядок может варьироваться.

- Способ учета различий между прописными и строчными буквами. В обычном русском словаре правилен порядок:

  небо, Нева, нега, недра, Нил,

  притом что, упорядочивая по кодам символов, мы получим:

  Нева, Нил, небо, нега, недра.

- Взаимное расположение латиницы и кириллицы. Русскому читателю привычно, чтобы кириллица предшествовала латинице:

  Волга, Нева, Фонтанка, Loire, Nile,

  а порядок, основанный на кодах символов, будет другим, так как у кириллицы коды больше, чем у латиницы. Кстати, если вам встретилось упорядочение

  нерпа, навес, нерв, недра, немец,

  то, скорее всего, в вашем тексте перепутана кириллица и латиница (определите сами, какие буквы могли быть набраны в латинице).

- Необходимость учитывать при упорядочении положение ударения. Например:

  пода́рок, по́дать, пода́ть, подда́ть.

- Специальное расположение особых знаков, добавляемых к основному алфавиту (буквы с диакритическими знаками, лигатуры, особые буквы), и наличие в языке букв, состоящих из нескольких букв алфавита. Так, в испанском языке кроме обычных букв и буквы *ñ*, следующей за *n*, за *l* следует буква *ll*, и правилен такой порядок слов:

  peludo, pellejo, pensado, peña.

6. Поиск образца в строке. Эта операция очень важна при компьютерной обработке текстов, с ней хорошо знакомы все, кто работал с каким-нибудь текстовым редактором: просматривается текст (его можно рассматривать как одну большую строку) и в нем разыскивается заданная подстрока — *образец*. Разговор об этой операции мы продолжим в п. 4.5.

7. Подстановка. В текстовых редакторах подстановка всегда сосуществует с поиском по образцу, найденный образец часто нужно заменить другим *). Здесь нам нечего сказать, кроме того, что такая операция существует. Главные хитрости, связанные с ней, относятся к способу хранения текстов в компьютере, который должен обеспечивать простое многократное исправление текста.

8. Преобразование. Это действие можно рассматривать как математическое развитие "текстовой" подстановки: задается отображение элементов строки-аргумента в некоторое множество B, а дальше просматривается строка и каждому ее элементу сопоставляется в образе строки образ этого элемента. Но не совсем так: во-первых, при подстановке задается частичное отображение, и тексты, которым в отображении ничего не соответствует, остаются без изменений. Во-вторых, при подстановке один фрагмент текста заменяется другим, а слово

*) При этом автоматическая замена провоцирует ошибки: очень трудно предугадать, где может встретиться, казалось бы, совершенно уникальный образец.

преобразование подразумевает замену элемента на элемент. Разумеется, возможны и используются многочисленные промежуточные формы подстановок-преобразований.

9. Фильтрация. Это еще одна операция из того же семейства. Все символы делятся на два подмножества, элементы одного оставляются без изменений, элементы другого исключаются из строки. Например, рассматривается последовательность натуральных чисел, и из нее отбираются только простые.

10. Слияние. Это операция с несколькими аргументами (чаще всего с двумя). Она характерна скорее для последовательностей, а не для сплошных строк. Ее важной особенностью является одновременный просмотр двух списков для получения из них одного (слитного) списка. Пусть на множестве элементов списков (алфавите) задан порядок. В ходе операции сравниваются начальные элементы списков, и меньший из них записывается в результат, удаляясь из своего списка. Мы еще вернемся к этой операции позднее, в п. 4.8.

11. Поиск максимального совпадения. Пусть заданы два списка $a[1:m]$ и $b[1:n]$ над одним алфавитом, и на алфавите определен порядок. Требуется найти такие монотонно возрастающие последовательности индексов $i_1, \ldots, i_s$ и $j_1, \ldots, j_s$, чтобы $1 \le i_1$, $i_s \le m$, $1 \le j_1$, $j_s \le n$, $a[i_k] = b[j_k]$ для любого $k \in 1:s$ и длина этих последовательностей s была бы максимальна.

Потребность в такой операции возникает, например, когда нужно составить список поправок к измененному текстовому файлу, — старый и новый файлы представляют собой два списка, а строки текста выступают как элементы наших списков. В параграфе 4.6 мы рассмотрим алгоритм поиска такой пары последовательностей, имеющий трудоемкость порядка $O(mn)$. На нем будет удобно показать технику работы со списками как вспомогательным средством.

4.3. Функции от строк

Как уже говорилось, часто встречаются функции от строк, в определении которых используется разделение строки на голову и хвост.

Функция $f(s)$, заданная на множестве всех строк S, называется *аддитивной*, если существует такая функция $\varphi : A \to R^1$, что

$$f(s) = \varphi(\mathsf{head}\ s) + f(\mathsf{tail}\ s),$$

когда строка непустая, и $f(s) = 0$ для пустой строки.

Пример 1. Длина строки (число элементов в ней) является аддитивной функцией:

$$\mathsf{length}(s) = 1 + \mathsf{length}(\mathsf{tail}\ s).$$

Пример 2. В предыдущем примере мы фактически приняли длину каждого элемента строки s равной единице. Если как-то определена *ширина* $\lambda(a)$ каждого элемента a, то ширина строки $L(s)$ определяется как сумма ширин входящих в строку элементов и является аддитивной функцией:

$$L(s) = \lambda(\mathsf{head}\ s) + L(\mathsf{tail}\ s).$$

Функция $f(s)$, заданная на множестве всех строк S, называется *мультипликативной*, если существует такая функция $\psi : A \to R^1$, что

$$f(s) = \psi(\mathsf{head}\ s) \times f(\mathsf{tail}\ s),$$

когда строка непустая, и $f(s) = 1$ для пустой строки.

Пример 3. Вероятность совмещения независимых в совокупности событий, заданных списком s, равна

$$\mathsf{P}(s) = \mathsf{P}(\mathsf{head}\ s) \times \mathsf{P}(\mathsf{tail}\ s)$$

и $\mathsf{P}(s) = 1$, когда список s пуст. Такой способ нахождения вероятностей характерен для демографических расчетов: например, при определении так называемой *продолжительности жизни* вероятность P_k дожития человека до возраста k принимается равной произведению $P_k = \prod_{i \in 0:k-1} p_i$, где p_i — вероятность прожить еще один год, находясь в возрасте i лет (в этой формуле, конечно, скрыты очень смелые упрощения).

Функция $f(s)$, заданная на множестве всех строк S, называется *АМ-тивной*, если существуют такие функции $\varphi, \psi : A \to R^1$, что

$$f(s) = \varphi(\mathsf{head}\ s) + \psi(\mathsf{head}\ s) \times f(\mathsf{tail}\ s),$$

когда строка непустая, и $f(s)$ как-то определена для пустой строки.

Пример 4. Значение полинома $P(x; \{a_0, \ldots, a_n\}) = \sum_{k \in 0:n} a_k x^k$ в точке x можно рассматривать как функцию от строки $\{a_0, \ldots, a_n\}$. Схема вычисления значения полинома хорошо известна (схема Горнера): полином представляется в виде $P(x; \{a_0, \ldots, a_n\}) = a_0 + x \cdot P(x; \{a_1, \ldots, a_n\})$. Тут сразу видно разбиение строки на голову a_0 и хвост $\{a_1, \ldots, a_n\}$.

Интересно участие этой полиномиальной формулы в экономических расчетах: когда требуется привести к единой системе измерения несколько платежей, сделанных в разное время, то, поскольку ценность денег в разные моменты времени различна *), каждый платеж пересчитывается в сегодняшние

*) Будущий рубль (доллар, евро, любые деньги) при нормальном развитии экономики стбит меньше, чем сегодняшний, в настоящее время это всем хорошо известно, а в советские времена нужно было объяснять даже экономистам. Уменьшение ценности определяется возможностью вложить сегодняшний рубль в экономический оборот и получить дополнительную прибыль.

деньги умножением на *коэффициент приведения* β^{t-t_0}, где t — момент платежа, t_0 — сегодняшний момент, а β — коэффициент пересчета на единицу времени (*коэффициент дисконтирования*).

Функция $f : S \to D$, где D — некоторое множество, называется *марковской* *), если существует такая функция $\Theta : A \times D \to D$, что

$$f(s) = \Theta(\text{head } s, f(\text{tail } s)),$$

и задано *начальное значение* для пустой строки $f(s) = r_0 \in D$.

Конечно же, марковскими являются все упомянутые нами функции: и аддитивные, и мультипликативные, и АМ-тивные функции. Но хочется привести и специфические примеры.

Пример 5. В современной полиграфии, основанной на компьютерах, оказались возможными технологии, которые раньше казались немыслимыми. Одна из них называется *кернингом* (от английского глагола to kern) и означает возможное изменение стандартного расстояния между парами соседних символов (например, уменьшение расстояния между Г и Л). В этой технологии ширина строки определяется не так, как мы это делали раньше, она равна сумме ширин отдельных символов и поправок на расстояния между соседними буквами. Например,

$$W(\text{ГЛУХОЙ}) = w(\text{Г}) + \delta(\text{Г},\text{Л}) + w(\text{Л}) + \cdots + w(\text{О}) + \delta(\text{О},\text{Й}) + w(\text{Й}).$$

Для последовательного вычисления значений функции W от префиксов строки s самих этих промежуточных значений недостаточно: на каждом шаге нужно знать последний символ префикса. Таким образом, вначале мы имеем пару $d_1 = (w_1, \text{Г})$, где $w_1 = w(\text{Г})$. По d_1 и Л вычисляем $d_2 = (w_1 + \delta(\text{Г},\text{Л}) + w(\text{Л}), \text{Л})$ и т. д.

Пример 6. Важнейшие примеры марковской функции от списка — максимальное и минимальное из значений, сопоставленных элементам этого списка. Для функции максимума, очевидно,

$$\Theta_{\max}(a, b) = \max\{a, b\}, \qquad r_0 = -\infty.$$

Аналогично, функция взятия минимума определяется с помощью

$$\Theta_{\min}(a, b) = \min\{a, b\}, \qquad r_0 = \infty.$$

А вот пример немарковской функции. *Размахом* строки называется разность ее максимума и минимума. Очевидно, что размах не является марковской

*) По имени академика А. А. Маркова-старшего (1856–1922), одного из виднейших ученых Петербургской математической школы, профессора Петербургского университета. В конце курса мы будем рассматривать введенные им вероятностные схемы, называемые *цепями Маркова*.

функцией строки. Однако пара (max, min) будет марковской функцией, а max — функцией от нее.

Марковские функции хороши тем, что их значения можно вычислять рекурсивно. На примере функции min покажем, что это вычисление можно организовать по-разному. С одной стороны, можно определить функцию, скажем, listmin1, полагая

$$\text{listmin1 } s = \begin{cases} \text{head } s, & \text{если length } s = 1, \\ \min\{\text{head } s, \text{listmin1}(\text{tail } s)\} & \text{в противном случае.} \end{cases}$$

Очень естественно — минимум по всему списку есть минимум из первого элемента и минимума по остатку. Это опасная естественность. Если следовать ей и двигаться в вычислениях от начала списка к концу, то происходит накопление незаконченных вычислений. Значительно эффективнее другая схема. Определим функцию от списка listmin2, выразив ее через еще одну функцию lm2, зависящую от двух аргументов:

$$\text{listmin2 } s = \text{lm2 } (\text{head } s, \text{tail } s),$$
$$\text{lm2 } (a, s) = \begin{cases} a, & \text{если length } s = 0, \\ \text{lm2 } (\min\{a, \text{head } s\}, \text{tail } s) & \text{в противном случае.} \end{cases}$$

Здесь важно, что функция lm2 также вычисляется рекурсивно, но незаконченных вычислений уже не будет.

Вообще, марковские функции можно создать для чего угодно, так как сам список может рассматриваться как марковская функция от себя самого. Поэтому когда речь должна идти об удобстве вычисления данной функции от списка, то подразумевается, что нужна марковская функция со значениями в множестве приемлемых размеров.

Упражнение 4.1. Рассматривается список чисел $\{a_i\}$, $i \in 1 : n$. Предложите марковскую функцию, с помощью которой можно вычислить по этому списку $\max\{a_i + a_{i+1} + a_{i+2} \mid i \in 1 : n - 2\}$.

Упражнение 4.2. Постройте функцию, которая по строке вероятностей дожития из примера 3 вычисляет продолжительность жизни. Какому из приведенных классов принадлежит эта функция?

Далее мы рассмотрим отдельно операции над текстовыми строками, а затем несколько важных приложений техники строк: множества на прямой, кусочно-постоянные и кусочно-линейные функции *).

*) В реально читаемом мною курсе для этого последнего материала времени обычно, к сожалению, не хватает, хотя здесь уместно было бы сказать и о длинной арифметике, и об аналитических вычислениях с помощью компьютеров, и о некоторых задачах вычислительной геометрии.

4.4. Скользящие суммы

Материал этого параграфа можно рассматривать как дальнейшее развитие идеи марковской функции. Нас будут интересовать функции, определенные на отрезках последовательности и удобные для вычисления. Начнем с важного частного случая.

Пусть задана числовая последовательность $p_1, p_2, \ldots, p_n$. Образуем из этой последовательности отрезки некоторой фиксированной конечной длины r:

$$P_1 = \{p_1, \ldots, p_r\}, \quad P_2 = \{p_2, \ldots, p_{r+1}\}, \quad \ldots, \quad P_{n-r+1} = \{p_{n-r+1}, \ldots, p_n\},$$

и для каждого отрезка P_k вычислим сумму его членов S_k. Легко догадаться, что при достаточно большом r значения S_2, S_3 и т.д. выгодно не считать каждый раз заново, а получать из предыдущего значения простым пересчетом $S_{k+1} = S_k - p_k + p_{k+r}$. Такой метод подсчета называется методом *скользящего суммирования*, а сами значения — *скользящими суммами*.

Скользящее суммирование широко используется при обработке длинных рядов статистических наблюдений, если требуется уменьшить влияние локальных флуктуаций и лучше увидеть основную тенденцию изменений.

По аналогии с суммой можно вычислять в той же "скользящей" технике и некоторые другие функции, в частности значения полиномов, в которых отрезки последовательности P_k служат коэффициентами:

$$\sigma_k(x) = \sum_{i \in 0:\, r-1} p_{k+i} x^{r-1-i}.$$

Здесь мы также можем получить формулу пересчета, чуть более сложную: $\sigma_{k+1} = x \cdot \sigma_k - p_k \cdot x^r + p_{k+r}$.

Упражнение 4.3. Рассмотрим последовательность 3, 9, 4, 1, 5, 8, 12, 7, 4, 4, 8. Вычислите для нее суммы $\sigma_k(10)$ при $r = 4$ (это очень легко), а затем $\sigma_k(7)$.

При достаточно больших r и x вычисление значений $\sigma_k(x)$ в компьютере может оказаться сложным из-за опасности переполнений. Но в некоторых случаях достаточно использовать не сами значения полиномов, а остатки от деления их на некоторое число. Выбрав некоторое целое d, определим $\rho_k(x)$ как остаток от деления $\sigma_k(x)$ на d и получим для этих значений формулу пересчета

$$\rho_{k+1} = (x \cdot \sigma_k - p_k \cdot R(x, r) + p_{k+r}) \bmod d,$$

где $R(x, r) = x^r \bmod d$.

Упражнение 4.4. Вернитесь к предыдущему упражнению и вычислите $\rho_k(17)$ при $d = 23$.

Эта функция используется для изготовления *дактилоскопических снимков* последовательности, своего рода ее *отпечатков пальцев* (fingerprints). Предположим, что нам задана текстовая строка $t[1:n]$ и нас интересуют ее подстроки длины r.

Можно каждой букве сопоставить некоторое число, например ее код в используемой кодировке, и по получившемуся набору чисел сопоставить каждой подстроке $T_k = t[k : k + r - 1]$ ее значение $\rho_k(x)$. Это значение можно, подобно отпечаткам пальцев в криминалистике, использовать как очень компактную характеристику подстроки: хотя мы и не можем гарантировать, что разные подстроки получат разные "дактилограммы", но шансы очень велики, особенно если позаботиться о правильном выборе параметров x и d — они должны быть взаимно просты, и желательно, чтобы d было простым.

Про эти дактилограммы мы скоро вспомним.

Упражнение 4.5. Используя функцию $K(X)$ из упражнения 2.27, разработайте скользящую функцию от битовой строки со значениями в множестве неотрицательных целочисленных матриц 2×2.

4.5. Поиск образца в строке

В оставшейся части главы мы рассмотрим несколько важных отдельных задач, связанных со строками. Вынужденно наше изложение будет отрывочным, слишком обширен накопившийся материал.

4.5.1. Задача точного поиска

Первой из таких задач будет *задача о поиске с точным совпадением*. Эта задача решается, например, в текстовом редакторе, когда пользователь разыскивает в редактируемом тексте образец, но практическое ее использование данным примером не ограничивается. Поэтому в научной литературе задаче о поиске уделяется очень большое внимание.

Задача о точном поиске образца в строке. Заданы две строки: строка текста $t[1 : n]$ (от text) и строка поискового образца $p[1 : m]$ (от pattern). Говорят, что образец входит в текст точно с позиции j, если $t[j : j + m - 1] = p[1 : m]$. Требуется найти все точные вхождения образца в текст. □

При самом наивном подходе, когда строка сканируется (просматривается подряд) для поиска первой буквы образца, а при обнаружении начального совпадения проверяется совпадение следующих букв, трудоемкость поиска (количество сравнений в наихудшем случае) пропорциональна $m \cdot n$. Есть, однако, более экономные методы.

Для начала мы можем вспомнить о "дактилограммах" строк, введенных в предыдущем параграфе. Используя их, получим следующий метод Карпа–Рабина ([68], см. также [13, 28]).

Выбрать подходящие x и D, снять "дактилограмму" ρ_p с образца, а затем просмотреть строку t, вычислить дактилограммы всех ее подстрок длины m, сравнить их с ρ_p и в случае совпадений выполнить полноценное сравнение образца с подстрокой.

Пример. Будем искать в строке

```
раз_два_три_четыре_и_четыре_на_четыре_на_четырнадцать_четыре
_и_потом_еще_четыре
```

образец `четырнадцать` (пробелы заменены подчеркиваниями, и знаки препинания убраны для наглядности). При $x = 137$ и $D = 12793$ мы получаем $\rho_p = 12418$ и единственную подстроку с тем же значением. Отметим, что три "внутренних" совпадения отпечатков подстрок показали нам, что в этом тексте есть три пары совпадающих подстрок длины 12 (какие?).

Авторы метода обосновывают его рассуждениями вероятностного типа, частыми в подобных исследованиях: если считать, что все значения дактилограмм получаются с равной вероятностью (которая в таком случае равна $1/D$), то математическое ожидание числа совпадений с образцом равно n/D, а ожидаемая трудоемкость затрат на их дополнительную проверку не превосходит mn/D. Слабое место таких обоснований в том, что мы работаем не с произвольными, случайно сформированными наборами символов, а с текстами, имеющими внутренние (не изученные) закономерности. Тем не менее, полученные так оценки хорошо подтверждаются экспериментально*).

Другой способ уменьшать затраты связан с более экономной проверкой и со специальной предварительной обработкой образца, которая, как показано в [13], сама по себе может свести затраты времени на поиск до $O(m+n)$, а дополнительной памяти — до $O(m)$.

Д. Гасфилд [13] предлагает в качестве основной такую предварительную обработку.

Нужно для каждого $i \in 2:m$ вычислить три числа:

- z_i – наибольшее k, при котором $p[i:(i+k-1)]$ совпадает с $p[1:k]$;
- r_i – наибольшее e, при котором найдется $b \le i$, такое что $p[b:e]$ совпадает с $p[1:(e-b+1)]$;
- l_i – какое-либо b из определения r_i.

Определяемая этими числами подстрока $p[l_i:r_i]$ совпадает с префиксом p и идет дальше всего по строке. По ней и определяется $z_i = l_i - i + 1$.

*) В работе [68] предлагается несколько вероятностных процедур, усиливающих эффект случайности за счет случайного выбора x среди простых чисел, принадлежащих заданному диапазону, а также одновременное вычисление нескольких дактилограмм.

Зная $z[2:i-1]$, $L=l_{i-1}$ и $R=r_{i-1}$, мы легко можем вычислить z_i и пересчитать L и R. Возможны несколько случаев. Если $i>R$, то подстрока, найденная для $i-1$, для i уже не годится и ее нужно вычислить заново. Непосредственно сравниваем строки $p[i:m]$ и $p[1:m]$, находим длину совпадающей части s и полагаем $z_i=s$, $L=l_i=i$, $R=r_i=i+s-1$.

Если $i\le R$, то символ $p[i]$ лежит в подстроке $p[L:R]$, совпадающей с префиксом, и следовательно, $p[i:R]$ совпадает с суффиксом этого префикса. Длина суффикса равна $S=R-i+1$, а начальная позиция $i'=i-L+1$. Самое главное: у нас уже вычислено значение $z_{i'}$.

Если $z_{i'}<S$, то нужно принять $z_i=z_{i'}$, а L и R не изменяются.

Если $z_{i'}\ge S$, то вся подстрока $p[i:R]$ совпадает с префиксом. Возможно, что это совпадение идет и дальше, так что нужно продолжить, сравнивая строки $p[R+1:m]$ и $p[S+1:m]$. Установив совпадение вплоть до позиций $R+k$ и $S+k$, полагаем $L:=i$, $R:=R+k$, $z_i=k+1$.

Упражнение 4.6. Докажите, что описанный алгоритм имеет линейную трудоемкость, т.е. что количество операций в нем имеет оценку сверху, линейно зависящую от длины строки.

Упражнение 4.7. Рассмотрим следующий алгоритм [13]. Соединим образец p и текст t в одну строку s, вставив между ними отсутствующий в них символ-разделитель (например, используя знак плюс в качестве разделителя $s=$`{bit+at_the_last_it_biteth_like_a_serpent}`). Затем, применив наш алгоритм к строке s, найдем значения i, для которых $z_i=m$. Докажите, что это будет алгоритм поиска образца в строке с линейной трудоемкостью. Оцените минимальный размер памяти, требуемый в этом алгоритме.

Основное алгоритмическое богатство этой задачи составляют два классических метода: Бойера–Мура и Кнута–Морриса–Пратта (Boyer–Moore и Knuth–Morris–Pratt, для краткости в дальнейшем БМ и КМП, см. [28]). Оба метода имеют линейную трудоемкость, оба являются замечательными улучшениями самого первого "наивного метода" и интересны различием подхода.

В методе БМ на каждом шаге образец "прикладывается" к текстовой строке так, чтобы символ $p[m]$ соответствовал некоторому символу $t[s]$, и сравнение образца с текстом идет *с конца*, пока не обнаружится различие или полное совпадение. Идея в том, что по завершении этого сравнения можно будет приложить образец не в следующей позиции, а существенно дальше.

Для определения нового положения образца относительно текста используются два правила. Первое из них называется правилом плохого символа. Когда в проверяемой подстроке текста $t[s-m+1:s]$ находится несовпадение с образцом, например для буквы $t[k]$, то образец

можно иногда продвинуть больше чем на одну позицию. Если буква текста $t[k]$ (она и есть плохая) не встречается в образце, как буква а в образце `колокол`, то образец можно продвинуть полностью за позицию k. Если же буква входит в образец, то можно совместить по ней образец с текстом, как в случае буквы к. Предпоисковая подготовка образца создает необходимый справочник для определения величины сдвига: для этой подготовки достаточно просмотреть образец один раз (эти детали мы опустим).

```
имея закольцованный % текст
колокол             % образец Р, несовпадение с буквой а
      колокол       % новое положение Р
имея закольцованный % текст
 колокол            % Р, несовпадение с буквой к
   колокол          % новое положение Р (по последнему вхождению к)
имея закольцованный % текст
      крыльцо       % Р, несовпадение с буквой о
       крыльцо      % новое положение Р (минимальный сдвиг)
```

Второе правило называется правилом хорошего окончания. Если окончание образца на некоторую длину совпадает с текстом, но затем появляется различие, то образец надо продвинуть до предыдущего появления этого фрагмента в образце

```
имея закольцованный % текст
   колокол          % Р, совпадение по окончанию кол
      колокол       % сдвиг Р до предыдущего вхождения окончания
```

Схема БМ многократно улучшалась. Ее называют *сублинейной*: хотя в ней количество сравнений и пропорционально числу букв текста, но обычно коэффициент в этой пропорции существенно меньше единицы. (Так, в первых 80 символах текста `мой дядя самых честных...` поиск образца `повеса` требует 15 сравнений, а образца `молодой повеса` — всего 5 сравнений.)

В схеме КМП [71] используется возможность сдвига образца вдоль текста в случае, когда совпали первые k символов. Сдвиг происходит не более чем на k позиций, размер сдвига $s(k)$ определяется предварительной обработкой образца, при которой вычисляются все $s(k)$. К сожалению, мы не имеем возможности останавливаться на этих вопросах подробнее.

4.5.2. Суффиксное дерево

В связи с задачами точного поиска была введена в обиход интересная информационная конструкция, называемая суффиксным деревом.

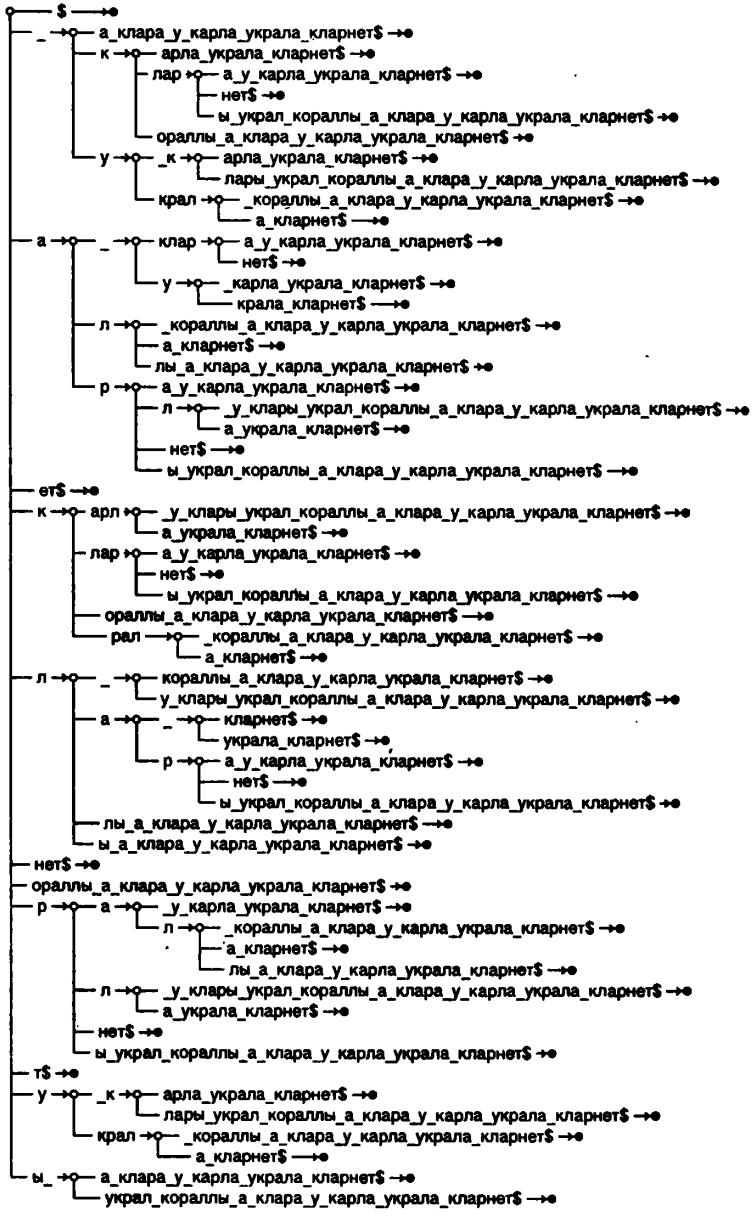


Рис. 4.1. Суффиксное дерево

Суффиксное дерево строится для экономного хранения всех суффиксов заданной строки. Возьмем для примера строку

```
карл_у_клары_украл_кораллы_а_клара_у_карла_украла_кларнет$
```

В конце строки приписан в качестве уникального символа знак $, который при лексикографическом сравнении будет считаться предшествующим всем остальным символам алфавита.

Суффиксное дерево для этой строки показано на рис. 4.1. Вы видите на рисунке белые и черные кружки. Черные кружки обозначают листья дерева и соответствуют суффиксам строки. Белые кружки — остальные вершины — называются, как раньше, *узлами*. Левый верхний кружок — корень дерева. На линиях, ведущих (по стрелке) из кружка в кружок, есть надписи — *метки*. Двигаясь по пути из корня к какому-нибудь узлу, можно составить ("прочесть") текст — это суффикс, соответствующий данному листу.

Дерево составляется так, чтобы из каждого узла выходило не менее двух линий и на линиях, выходящих из каждого узла, все метки начинались с разных символов.

Суффиксное дерево очень удобно для поиска в строке образца. Действительно, взяв образец, нужно проверить, выходит ли из корня линия с нужным началом. Если нет, то образец в строку не входит. Если да, то нужно идти по этой линии, "читая" образец. Доходя до следующего узла, если образец "не дочитан", нужно снова искать линию с нужной буквой для продолжения.

Например, если искать в строке на рис. 4.1 слово `карлик`, мы сначала находим выходящую из корня линию `к...` Прочтя метку `к`, попадаем в новый узел и ищем в нем для продолжения линию `а...` Двигаясь по ней, прочитываем дополнительно метку `арл` и ищем линию `и...` Такой линии нет, и следовательно, образец в текст не входит.

Легко видеть, что время, затрачиваемое на поиск вхождения образца, пропорционально длине этого образца, так что суффиксные деревья удобно использовать для многократного поиска образцов в одной и той же строке, задаваемой заранее.

Хранение суффиксного дерева требует довольно много памяти, но размер памяти зависит от длины строки линейно. Благодаря этому есть алгоритмы, которые строят суффиксное дерево за линейное время. К сожалению, описание этих алгоритмов потребовало бы слишком много места.

4.5.3. Задачи приближенного поиска

Упомянем еще, что кроме точного поиска рассматриваются разнообразные варианты *задачи приближенного поиска образца*, в которых не ищется обязательно точное вхождение образца, а допускаются некоторые приближения. Вы можете встретиться с такими "приближениями" в текстовых редакторах в ситуации, когда программа проверки

грамотности предлагает, чем можно заменить слово, отсутствующее в ее словаре.

Разнообразие вариантов связано с различием постановок задач. В некоторых случаях ограничивается количество букв, которые нужно изменить, удалить или добавить для получения данной строки*). Возможны и другие определения расстояния, например учитывающие схожесть букв (и правдоподобность ошибки) или положение отклонения внутри слова.

Задачи поиска образца в строке имеют огромное значение в вычислительной молекулярной биологии при анализе длинных молекул (расшифровке кодов ДНК).

4.5.4. Регулярные выражения

Во многих текстовых редакторах и специальных программах сейчас имеется более изощренная система поиска, использующая для задания образца так называемые *регулярные выражения*.

Регулярные выражения позволяют экономно создавать образцы поиска с помощью операций над другими образцами (хотя это далеко не единственное и не самое важное их применение) **).

Идея в следующем. Вначале вводятся атомарные операнды.

- Буква исходного алфавита. Она порождает множество из одной строки, состоящей из этой буквы.
- ϵ — пустая строка. Этот символ считается порождающим множество, состоящее из одной пустой строки.
- $\varnothing$ — пустое множество строк.

В описываемых ниже операциях будут участвовать эти атомарные операнды и порождаемые ими множества строк. Наряду с ними в операциях будут участвовать результаты операций (в точной аналогии с арифметическими выражениями, в которых участвуют как атомарные операнды-числа, так и другие арифметические выражения).

Основных операций всего три, и две из них нам уже знакомы.

Объединение. Результатом этой операции является объединение множеств строк, соответствующих аргументам.

*) Количество таких исправлений называется расстоянием Левенштейна [13] по имени В. И. Левенштейна, который ввел его в 1965 г. [31].

**) В учебнике Ахо и Ульмана [53], где этому подходу отводится почетное место, указано, что регулярные выражения появились в статье С. Клини (Kleene, Steven Cole (1909–1994)) в 1956 г., а затем активно использовались К. Томсоном (K. Thompson, 1968) в его системе QED и в UNIX в знаменитой команде `grep`.

Конкатенация. Результатом этой операции является множество всех строк, составленных конкатенацией строк из множеств, соответствующих аргументам.

Замыкание (или *замыкание Клини*). Результатом этой операции является конкатенация произвольного числа строк из множества, соответствующего (единственному) аргументу.

Объединение обозначается вертикальной чертой: `a|b`; конкатенация, подобно умножению в арифметике, изображается выписыванием аргументов рядом; замыкание обозначается знаком * после аргумента. Таким образом, используя скобки для фиксации порядка действий, можно определить множество всех последовательностей из 0 и 1, разделенных запятыми, выражением `((0|1),)*(0|1)`.

В практических системах поиска по регулярным выражениям (например, в операционной системе UNIX, языках AWK, Perl, Java) наборы операций более развиты и удобны. Например, в них можно определять диапазоны символов, так что `[A-Za-z]` определяет множество всех латинских букв. Наряду с операцией * используется операция + для конкатенации не менее одного операнда. Приведем несколько примеров из языка AWK:

`"(бит|байт)"`	— `бит` или `байт`;
`"б(и|ай)т"`	— то же более экономно;
`"(б|Б)(и|ай)т"`	— `бит` или `байт` с большой или маленькой буквы;
`"( |^)(б|Б)(и|ай)т"`	— то же с начала слова (т.е. после пробела или в начале строки);
`"бит.+бит"`	— два разделенных вхождения `бит` в одной строке;
`"бит.*[.,].*бит"`	— то же, но между вхождениями должна быть точка или запятая;
`"( |^)при[и-лн-т])"`	— слова, начинающиеся с `при` и букв и, й, к, л, н, о, п, р, с или т.

Мы вернемся к поиску по регулярным выражениям в конце курса, когда речь пойдет о конечных автоматах.

4.6. Задача о максимальном совпадении двух строк

В этом параграфе мы рассмотрим еще одну важную задачу, которая может возникать в нескольких обличиях. Для начала представьте себе, что вы хотите переслать своему адресату исправления к некоторому длинному текстовому файлу. Это нормальная ситуация: зачем заново пересылать исправленный файл, если длина исправлений существенно меньше? Когда мы,

вдохновившись расстоянием Левенштейна, упомянутым выше, найдем расстояние между старым и новым файлом, эти файлы как бы разобьются на их общую часть (хочется сказать "пересечение") и определяющую расстояние "симметрическую разность" (можно ограничиться только вставками и удалениями). Минимизировать разность — это то же, что максимизировать общую часть.

Задача о максимальном совпадении двух строк. Пусть заданы две последовательности (строки) $a[1:m]$ и $b[1:n]$. Требуется найти их максимальное совпадение, т.е. последовательность, которая является подпоследовательностью их обеих и имеет наибольшую длину. ◻

Наряду с исходной задачей мы будем решать задачи с меньшими строками, $a[1:k]$ и $b[1:l]$, где $k \in 1:m$ и $l \in 1:n$. Задачу с такими параметрами будем называть задачей (k, l), а максимальную длину последовательности в ней обозначим через $v(k, l)$ (так что исходная задача становится задачей (m, n) и требуется найти $v(m, n)$).

Проще решать все эти задачи вместе, находя решения друг через друга. Сформируем из искомых значений матрицу $v[1:m, 1:n]$. Начнем с того, что решения при $k = 1$ и при $l = 1$ выписываются сразу:

$$v[1,1] = \begin{cases} 1, & \text{если } a[1] = b[1], \\ 0 & \text{в противном случае;} \end{cases}$$

$$v[k,1] = \begin{cases} 1, & \text{если } v[k-1,1] = 1 \text{ или } a[k] = b[1], \\ 0 & \text{в противном случае;} \end{cases}$$

$$v[1,l] = \begin{cases} 1, & \text{если } v[1,l-1] = 1 \text{ или } a[1] = b[l], \\ 0 & \text{в противном случае.} \end{cases}$$

Для произвольного элемента матрицы, "отодвинутого" от левого верхнего края, получаем:

$$v[k,l] = \begin{cases} 1 + v[k-1,l-1], & \text{если } a[k] = b[l], \\ \max\{v[k-1,l], v[k,l-1]\} & \text{в противном случае.} \end{cases}$$

Вряд ли эта формула нуждается в больших комментариях: если последние элементы строк совпадают, то совпадающее значение выгодно включить в последовательности, а для оставшихся начал строк решить задачу $(k-1, l-1)$. Если же они не совпадают, то нужно попробовать оба способа: обойтись без последнего элемента и выбрать тот, который дает лучший результат.

Выписанные формулы можно прямо использовать для вычислений, получая последовательно один столбец за другим.

Пример. Пусть заданы две строки, $s = abcdefafgcbd$ длины $m = 12$ и $t = badcfegecdb$ длины $n = 11$. Результаты вычисления $v[\cdot,\cdot]$ представлены

в таблице, причем максимальная цепочка совпадений выделена курсивом, а остальные скачки —жирным шрифтом.

		1	2	3	4	5	6	7	8	9	10	11
		b	*a*	*d*	*c*	*f*	*e*	*g*	*e*	*c*	*d*	*b*
1	*a*	0	**1**	1	1	1	1	1	1	1	1	1
2	*b*	***1***	1	1	1	1	1	1	1	1	1	**2**
3	*c*	1	1	1	**2**	2	2	2	**2**	2	2	2
4	*d*	1	1	***2***	2	2	2	2	2	2	**3**	3
5	*e*	1	1	2	2	2	**3**	3	3	3	3	3
6	*f*	1	1	2	2	***3***	3	3	3	3	3	3
7	*a*	1	**2**	2	2	3	3	3	3	3	3	3
8	*f*	1	2	2	2	3	3	3	3	3	3	3
9	*g*	1	2	2	2	3	3	***4***	4	4	4	4
10	*c*	1	2	2	**3**	3	3	4	4	***5***	5	5
11	*b*	1	2	2	3	3	3	4	4	5	5	**6**
12	*d*	1	2	**3**	3	3	3	4	4	5	***6***	6

Такая таблица хороша для того, чтобы понять, что и как считается, но в реальных вычислениях лучше стараться обойтись без нее. Ну что же, задача кажется простой — можно иметь один текущий столбец таблицы и его последовательно пересчитывать. Однако это не вполне так: после расчета таблицы мы получаем только значения v, а если требуются еще и сами подпоследовательности, то нужно выполнить *обратный ход*, а для него недостаточно иметь последний сохранившийся столбец.

Сейчас нашей целью будет выделение из таблицы необходимой информации и ее экономное сохранение. Взгляните внимательно на таблицу. В ней много одинаковых значений, но особо выделяются *точки скачков*, от которых одинаковые значения распространяются вправо и вниз, пока не будут заменены бо́льшими значениями. Все элементы, выделенные жирным шрифтом, — это как раз такие скачки. Будем сохранять только скачки (val), запоминая от каждого его координаты (строку и столбец) и номер предыдущего скачка (prec), из которого получился данный. Получим список скромного размера:

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
j	1	2	2	3	3	4	4	5	6	7	9	10	10	11	11
i	2	1	7	4	12	3	10	6	5	9	10	4	12	2	11
val	1	1	2	2	3	2	3	3	3	4	5	3	6	1	6
prec	0	0	2	2	4	2	6	6	6	9	10	6	11	2	11

При вычислении очередного столбца каждому элементу можно сопоставить скачок, от которого данный элемент получает свое значение. Например, для последнего столбца $v[\cdot, 11] = (1, 2, 2, 3, 3, 3, 3, 3, 4, 5, 6, 6)$ получим $r[\cdot, 11] = (2, 14, 14, 12, 12, 12, 12, 12, 10, 11, 15, 15)$. Для "обратного хода" — поиска максимальной подстроки — теперь достаточно взять скачок, соответствующий последнему элементу столбца, в нашем случае это скачок 15, а от него пройтись по списку предыдущих скачков:

$$15 \to 11 \to 10 \to 9 \to 6 \to 2.$$

Детали алгоритма оставляются читателю в качестве упражнения.

Что же касается самой задачи, то для нее разработано много вариантов приведенного алгоритма, использующих особенности данных, например то, что алфавит содержит не очень много букв или малы различия двух строк. Так, в статье Хиршберга [67] даны алгоритмы трудоемкости $O(pn)$ и $O(pe \log n)$, где p — размер максимальной подпоследовательности, а $e = m - p$.

Рассмотренный в этом параграфе метод сведения экстремальной задачи к совокупности решаемых совместно экстремальных задач называется методом *динамического программирования*. Мы будем возвращаться к этому методу еще много раз. Прием сокращенного хранения таблицы очень полезен и широко используется в практических задачах. В качестве примера такого использования можно назвать задачи оптимального раскроя (линейных и плоских материалов).

Очень интересный метод был предложен в 1987 и 1992 гг. рядом авторов (см. подробности авторства в [13]). Метод связан с описанной на с. 42 задачей о максимальной возрастающей подпоследовательности. Но нам понадобится та версия задачи, о которой говорится в упражнении: исходная последовательность будет содержать повторы. Рассмотрим предлагаемый метод на примере. Пусть требуется найти наибольшую общую подпоследовательность в строках

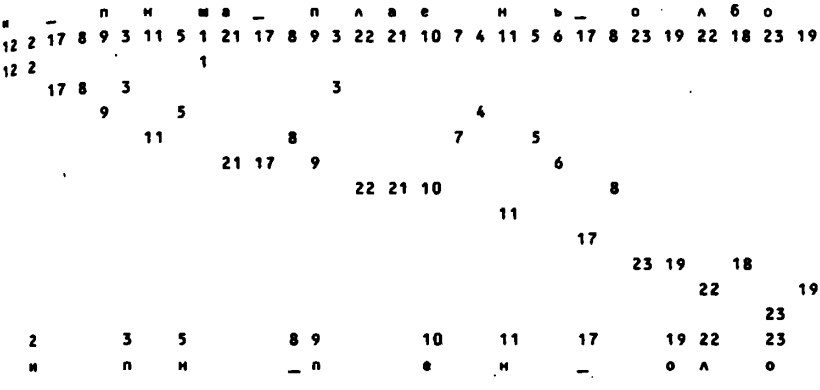
```
          1         2
  123456789012345678901234
s: шипенье_пенистых_бокалов
t: и_пунша_пламень_голубой
```

Сначала выпишем для каждого символа, входящего в строку s, все позиции, в которых этот символ встречается. Будет удобно перечислить их по убыванию:

ш	1	и	12	2	п	9	3	е	10	7	4	н	11	5	ь	6
_	17	8	с	13	т	14		ы	15			х	16		б	18
о	23	19	к	20	а	21		л	22			в	24			

Затем для каждого символа строки t выпишем эти последовательности: для и, для пробела, для п и т.д. (в некоторых случаях пустую) и составим из них

общую последовательность позиций. Затем найдем в этой последовательности наибольшую возрастающую подпоследовательность:



Упражнение 4.8. Докажите, что описанный алгоритм действительно находит наибольшую общую подпоследовательность двух строк. Оцените его трудоемкость.

4.7. Задача Кнута–Пласса о выключке абзаца

Вот еще одна задача, в которой используется техника динамического программирования.

Как вы знаете, печатный текст всегда разбивается на фрагменты, набираемые отдельно, — абзацы. Каждый абзац набирается в виде нескольких полиграфических строк, обычно таким образом, чтобы правый край всех этих строк был выровнен, а в начале первой строки слева был дополнительный отступ, который так и называется *абзацным отступом*. Для того чтобы добиться выравнивания строк в абзаце, используют несколько приемов: разрешается разбивать слова в некоторых местах и переносить часть слова на следующую строку, изменять межсловные пробелы, менять размер текста в последней строке. Эта операция, называемая *выключкой абзаца*, в прежние "докомпьютерные" времена требовала у наборщиков много времени, и при ручном наборе ее результат существенно зависел от квалификации наборщика. Теперь текстовые редакторы (например, MS Word) выполняют эту операцию автоматически, но качество результата не всегда достаточно высокое.

Когда Д. Кнут разрабатывал систему полиграфической обработки текста TEX, он вместе со своим аспирантом М. Плассом предложил метод вычисления наилучшей выключки. Этот метод основывается на решении задачи о минимизации суммарного штрафа за отклонение выключки от идеальной.

Штраф — это некоторая функция от строки-абзаца с указанными в ней местами разбиений. Штраф состоит из нескольких слагаемых — количества переносов (и их качества, которое определяется по специальным таблицам), отклонения размера межсловных пробелов в меньшую или бо́льшую сторону от нормативной величины, слишком большого или слишком малого размера последней строки [*]. Выбранная целевая функция является аддитивной, и это свойство позволяет вычислять наилучшую выключку достаточно быстро.

Вместе с тем аддитивная функция позволяет учесть не все дефекты набора. С ее помощью, например, не удается оштрафовать так называемые коридоры — совпадения межсловных пробелов в идущих подряд строках текста, которые образуют белую полосу, "разваливающую" текст абзаца.

4.8. Слияние

Начиная с этого параграфа и до конца главы мы будем рассматривать применения техники списков в различных вычислительных задачах. Уже упоминавшаяся операция слияния двух списков в один будет хорошим началом.

Напомним, что слияние массивов выполняется следующим образом. Возьмем первые элементы двух упорядоченных массивов и выберем тот, который должен быть раньше. Этот выбранный элемент запишем в массив результата и уменьшим тот массив, из которого элемент взят (конечно, речь идет не о физическом уменьшении, мы просто уменьшаем область изменения индексов массива), после чего применим ту же операцию к оставшимся массивам. Когда один из массивов исчерпается, остатки другого припишем к массиву результата.

Пример. Сливая массивы (3, 7, 9, 15) и (4, 6, 8), мы сравниваем числа 3 и 4, записываем 3 в результат и продолжаем работать с массивами (7, 9, 15) и (4, 6, 8). Теперь, сравнивая 7 и 4, записываем в результат 4 и продолжаем с (7, 9, 15) и (6, 8), и т. д.

После того как из второго массива мы возьмем 8, последний элемент, остаток первого массива просто приписывается к результату.

Легко видеть, что трудоемкость операции слияния пропорциональна суммарной длине сливаемых массивов: на каждом шаге сравниваются два значения и одно навсегда переносится в результат, так что шагов столько, сколько элементов в массивах.

---

[*] Вы можете увидеть результат и в этом издании, так как оно было сверстано в TEXe. Что же касается текстовых редакторов, то качество их выключки постепенно улучшается, в частности под влиянием работы Кнута – Пласса.

Запишем теперь операцию слияния списков Merge (так она называется по-английски) в программоподобном стиле — алгоритмы гораздо проще описывать, используя стиль (не обязательно точные детали) алгоритмических языков. Сначала напишем, потом объясним.

```
Merge(A,B) result R ===
  <прочесть первый элемент из списка A>
  <прочесть первый элемент из списка B>
  <создать пустой список результата R>
  bContinue := True;
     // булева переменная для управления циклом
  while bContinue do
    if <есть элемент из A> then
      if <есть элемент из B> then
        if <элемент из A предшествует элементу из B> then
           <записать элемент из A в R>
           <прочесть очередной элемент из A>
        else
           <записать элемент из B в R>
           <прочесть очередной элемент из R>
        fi
      else
        <скопировать в R остаток A>
        bContinue := False
      fi
    else
      if <есть элемент из B> then
        <скопировать в R остаток B>
      fi;
      bContinue := False
    fi
  od;
```

Здесь в угловых скобках мы пишем *псевдооператоры* — действия, для которых программная реализация еще не выбрана. Мы стараемся вводить новые переменные, смысл которых уже понятен, и пишем для них комментарии, от двойной косой черты до конца строки.

Из Паскаля взят знак присваивания. Условное выражение начинается с `if` и кончается `fi` (перевернутым `if`). Тело цикла начинается с `do` и кончается `od`. За точками с запятой мы не следим и ставим их, когда удобно.

Такая система записи удобна для последовательной разработки алгоритма. Например, сейчас мы можем уточнить одно из действий следующим образом:

```
CopyRem(P,R) ===
  // копирование в R остатка P
while <существует очередной элемент из P> do
  <записать элемент из P в R>
  <прочесть очередной элемент из P>
od;
```

Возможны варианты процедуры Merge, позволяющие сливать не два, а больше списков.

Упражнение 4.9. Опишите процедуру слияния m списков в один, в которой имеется массив списков.

Упражнение 4.10. Опишите вариант процедуры слияния с удалением кратных элементов (это значит, что, если один и тот же элемент встретился в нескольких сливаемых списках, он попадает в результат только один раз).

Процедура слияния настолько проста, что иногда ее проще не делать заранее, а выполнять "на лету", в ходе работы алгоритма, использующего результат слияния. Такой список, создаваемый по мере надобности, в программном отношении выглядит как обычный список. Программные объекты такого рода называются *виртуальными*.

Слияние служит образцом для многих других аналогичных операций, использующих просмотры списков, и некоторые из них мы сейчас рассмотрим.

4.9. Операции над множествами на прямой

Множество на прямой, если оно состоит из конечного числа интервалов, может быть задано списком интервалов. Все обычные операции над множествами становятся при этом операциями над списками, например объединение двух множеств A и B вычисляется следующим образом (считаем границы всех интервалов целочисленными):

1. Создать пустой список результата C.
2. Если оба списка пусты, завершить работу.
3. Если один из списков пуст, присоединить другой список к C и завершить работу.

4. Выбрать в обоих списках интервал с наименьшим левым краем, удалить его из своего списка и создать из него интервал D. Пока в списках есть интервалы, пересекающиеся с D, объединять их с D, удаляя из своих списков. Затем включить D в список C и вернуться к пункту 2.

Упражнение 4.11. Описать аналогичным образом операции пересечения двух множеств, дополнения, симметрической разности.

4.10. Длинная арифметика

Обычно машинные вычисления с целыми числами проводят в специальных форматах, которые хорошо поддерживаются системой машинных команд. Однако в некоторых случаях приходится делать вычисления с очень большими числами, и здесь вспоминаются школьная тетрадка в клетку и строгие правила позиционирования разрядов.

В позиционной системе мы представляем целое число как полином от степеней некоторого числа (это может быть 10, 2, 8 или 16), например

$$1\,008\,350\,074 = 4 \cdot 10^0 + 7 \cdot 10^1 + 5 \cdot 10^4 + 3 \cdot 10^5 + 8 \cdot 10^6 + 1 \cdot 10^9.$$

Эту запись можно трактовать как строку

$$(4, 7, 0, 0, 5, 3, 8, 0, 0, 1)$$

либо, при желании учесть особенности "разреженности", как строку

$$((4, 0), (7, 1), (5, 4), (3, 5), (8, 6), (1, 9)).$$

Выясним, как выглядят при таком задании чисел хотя бы простейшие арифметические операции. Мы предпочтем сейчас работать с более простой первой формой записи.

Сложение. Результатом сложения двух строк, представляющих положительные целые числа, будет строка, получающаяся при другой операции — *сложении с переносом* этих двух строк и еще одного числа — *переноса*. Перенос следует принять равным 0. Эту дополнительную операцию мы сейчас тоже определим:

```
add(num1,num2) = addWithCarry(num1,num2,0)
```

Сложение с переносом. Результатом сложения с переносом двух строк, представляющих положительные целые числа, будет строка, получающаяся конкатенацией младшего разряда результата и всего остального. Младший разряд R получается сложением голов (первых элементов) обеих этих строк и переноса C. Все остальное — это результат сложения остатков строк с переносом, полученным при разложении R на младший разряд и переносимую часть остатков строк.

```
addWithCarry(num1,num2,C) =
  catenate(remainder(R),
           addWithCarry(tail(num1),tail(num2),carry(R)) )
R = head(num1) + head(num2) + C
```

Мы не будем рассматривать здесь все арифметические операции, отметим только особую роль умножения длинных целых, для которого существуют специальные "быстрые алгоритмы".

4.11. Кусочно-постоянные функции

Кусочно-постоянная функция на отрезке может быть задана как список интервалов постоянства функции.

Пример. Кусочно-постоянная функция $f(x)$, определенная на $[0, 1]$,

$$f(x) = \begin{cases} 3, & 0 < x < 0.5, \\ 2, & 0.5 < x < 0.8, \\ 4, & 0.8 < x < 0.92, \\ 3, & 0.92 < x < 1 \end{cases}$$

задается списком $((0, 0.5, 3), (0.5, 0.8, 2), (0.8, 0.92, 4), (0.92, 1, 3))$ (мы пренебрегаем здесь вопросом о задании значений на границах областей). Функция f имеет четыре сплошные области постоянства, равенство значений в первой и в четвертой области мы игнорируем.

Можно сэкономить память и упростить действия, если в записи скачки задавать только одну, например левую, границу интервала.

В терминах операций над списками можно выразить такие операции над функциями, как умножение функции на число, прибавление константы, нахождение суммы, разности, минимума и максимума двух функций. Для многих приложений полезны операции проверки монотонности функции и построения наименьшей монотонно неубывающей функции, мажорирующей данную (*мажоранты функции*).

Одноместные операции. Рассмотрим сначала те операции, которые выполняются одним простым просмотром списка-аргумента. Типовая структура таких операций — это цикл по списку:

```
<создание пустого списка результата ListRes>
curItem := List.Head;
while <curItem существует> do
  <создание по curItem элемента результата curItemNew>
  <присоединение curItemNew в конец ListRes>
  <переход к следующему curItem>
od;
```

В рамках этой структуры операции умножения на число и добавления константы не должны вызвать никаких проблем. Однако при рассмотрении более общего действия *пересчет значения функции* даже небольшое усложнение этого действия вызывает дополнительные трудности. Рассмотрим, например, пересчет значения функции *с округлением* (при таком пересчете теряется взаимная однозначность отображения).

```
<создание пустого списка результата ListRes>
curItem := List.Head;
<создание по curItem элемента результата curItemNew>
<переход к следующему curItem>
while <curItem существует> do
  <вычисление значения новой функции vNew по curItem>
  if vNew = <значение в curItemNew> then
    <добавление к интервалу curItemNew интервала curItem>
  else
    <присоединение curItemNew в конец ListRes>
    <создание по curItem элемента результата curItemNew>
  fi;
  <переход к следующему curItem>
od;
<присоединение curItemNew в конец ListRes>
```

Упражнение 4.12. Как изменится эта схема, если в записи скачка хранится только левый конец интервала?

Построение наименьшей мажоранты данной функции отличается только тем, что иначе решается вопрос о создании в мажоранте нового скачка: скачки исходной функции, нарушающие монотонность, должны пропускаться.

Пример. Для функции, заданной списком скачков

$$[(0, 0.2),\ (0.3, 7.1),\ (0.5, 3.4),\ (0.53, 3.2),\ (0.57, 5.9),\ (0.6, 8),\ (0.7, 1.9)].$$

монотонная мажоранта задается, очевидно, списком

$$[(0, 0.2),\ (0.3, 7.1),\ (0.6, 8)].$$

Наибольшую монотонную *миноранту* для данной функции, т. е. наибольшую монотонную функцию, не превосходящую данной, строить не так просто. В только что приведенном примере миноранта задается списком скачков [(0, 0.2), (0.7, 1.9)]: при анализе очередного скачка список уже отобранных скачков может сильно измениться.

Упражнение 4.13. Придумать способ построения монотонной миноранты, использующий операцию переворачивания списка.

Двуместные операции. Выполнение двуместных операций, таких как сложение и вычитание функций, вычисление максимума и минимума функций, требует (в лучшем случае) одновременного просмотра списков, соответствующих обоим аргументам; идея этого просмотра вполне аналогична уже известной нам операции слияния.

```
<создание пустого списка результата ListRes>
curItemA := ListA.Head; curItemB := ListB.Head;
while <оба списка непусты> do
  <создание по curItemA и curItemB нового элемента
   результата curItemNew>
  <возможное присоединение curItem в конец ListRes
   и сохранение curItemNew как curItem>
  <возможное обновление curItemA>
  <возможное обновление curItemB>
od;
```

Упражнение 4.14. Выписать полностью перечисленные операции над кусочно-постоянными функциями.

Упражнение 4.15. Пусть $f(x)$ и $g(x)$ — определенные на $[0, 1]$ неубывающие кусочно-постоянные функции. Как вычислить

$$\int_0^1 |f(x) - g(x)|\,dx\ ?$$

Упражнение 4.16. Пусть $f(x)$ и $g(x)$ — неубывающие кусочно-постоянные функции. Доказать, что функция $f(g(x))$ также неубывающая и кусочно-постоянная. Разработать метод ее построения в виде списка скачков.

В приложениях встречаются самые разнообразные применения техники работы с такими заданиями кусочно-постоянных функций. Вот одно из них.

Пример. *Построение рюкзачной функции.* Странное название (knapsack function) связано со следующей "практической" *задачей о рюкзаке*: вор, забравшийся на склад, может положить в свой рюкзак вес, не превышающий W. Какие товары он должен выбрать, чтобы унести максимальную стоимость?

Эта максимальная стоимость как функция от W и называется рюкзачной функцией.

Оставляя теперь в стороне само это сомнительное приложение, перенумеруем товары от 1 до m и каждому товару сопоставим вес w_i и цену c_i. Для рюкзачной функции $v(w)$ имеем:

$$v(w) = v_m(w) = \max\left\{ \sum_{i\in 1:m} c_i k_i \;\middle|\; \sum_{i\in 1:m} w_i k_i \le w \right\},$$

где все $k_i \in 0:1$. Аналогично определим для каждого $r \in 1:m-1$ свою рюкзачную функцию $v_r(w)$:

$$v_r(w) = \max\left\{ \sum_{i\in 1:r} c_i k_i \;\middle|\; \sum_{i\in 1:r} w_i k_i \le w \right\}.$$

Функции v_r, очевидно, неубывающие и кусочно-постоянные, их можно хранить в виде последовательности скачков.

В таком же виде их можно и вычислять. Введем операцию сдвига функции v на пару (x, y) следующим образом: $v' = \mathsf{offset}(v, (x, y))$, если

$$v'(w) = \begin{cases} 0, & w < x, \\ y + v(w - x), & w \ge x. \end{cases}$$

Обозначим минимальную мажоранту функций f и g через $\mathsf{major}(f, g)$. Обе эти функции легко реализуются с функциями, заданными наборами скачков.

Теперь, задавшись функцией $v_0 \equiv 0$, мы можем последовательно найти функции v_r следующим образом:

$$v_r = \mathsf{major}(v_{r-1}, \mathsf{offset}(v_{r-1}, (w_r, c_r))).$$

Задача о рюкзаке нам еще встретится, хотя и в другой постановке (и в этой постановке она будет решаться сложнее).

Аналогично кусочно-постоянной, кусочно-линейную функцию можно задавать списком участков линейности и аналогичным образом определять операции над такими функциями в терминах списков. К упомянутым уже операциям здесь можно добавить построение вогнутой мажоранты или выпуклой миноранты данной функции. Дальнейшее развитие той же техники приводит нас и к кусочно-полиномиальным функциям.

Глава 5

Сжатие и защита информации

5.1. Введение

В главе 2 обсуждался *базовый* метод представления текстовой информации, в котором каждому символу сопоставляется байт или пара байтов. Этот метод не всегда хорош. В некоторых случаях он слишком "затратен", можно было бы обойтись меньшим местом, а места жалко. Здесь мы рассмотрим некоторые специальные способы кодирования.

В связи с желанием экономить место появились, сейчас уже многочисленные, способы *сжатия* информации — более компактной ее записи. Сейчас эти способы хорошо известны по так называемым *архиваторам* — сервисным программам, сжимающим файлы.

В этой главе мы рассмотрим некоторые, наиболее важные и принципиальные, способы сжатия. Это будет первой частью главы.

Вторая часть посвящена, так сказать, противоположной тенденции. Иногда для придания информации некоторых дополнительных качеств ее преобразуют. При этом объем информации несколько увеличивается. Обычно эти дополнительные качества связываются с облегчением восприятия и переработки информации — этот аспект вопроса нас сейчас интересовать не будет — либо защитой информации от случайных помех и (очень важно!) от так называемого *несанкционированного доступа*, т.е. от случаев, когда лицо, не имеющее на это права, может информацию прочесть, уничтожить или исказить. Эта тематика, конечно же, связана с многочисленными сейчас "компьютерными преступлениями".

5.2. Код Шеннона–Фано и алгоритм Хаффмена

При кодировании нормальных текстов обычно принимается какой-нибудь стандартный размер кодовых последовательностей. Например, мы уже встречали кодировку ASCII, в которой каждый символ занимает один байт — 8 битов, и двухбайтовую кодировку Unicode. В первых моделях телеграфа появилась и долго использовалась и пятибитовая кодировка. Каждая из таких кодировок имеет ограниченное число кодовых возможностей (хотя в Unicode это ограничение слабо чувствуется).

Когда нужно получить дополнительные коды, часто используется идея специального расширяющего символа, он называется *Escape*-символом *): пара из такого символа и продолжения рассматривается как единый символ в расширенном кодовом пространстве. Упоминавшаяся на с. 26 вполне современная кодировка UTF-8 также относится к такому классу кодировок с расширением. Расширение дает простой выход из положения, но при массовом использовании и нестандартных наборах символов оказалось неэффективным.

К. Шеннон и Р. Фано предложили в наиболее чистом виде конструкцию *кода переменной длины*. В этом коде у каждого символа своя длина кодовой последовательности, как в азбуке Морзе **). Но в отличие от азбуки Морзе, где конец кодовой последовательности определяется "третьим символом" — паузой, здесь нужно побеспокоиться о том, как определять завершение кода отдельного символа. Предлагается такое ограничение на код: *никакая кодовая последовательность не является началом другой кодовой последовательности*. Это свойство называется *свойством префикса*, а код, обладающий таким свойством, называется *префиксным кодом*.

В предположении, что кодируемые символы появляются в тексте независимо ***), нужно стремиться уменьшать среднее число битов на

*) To escape — выходить, высвобождаться из правил и ограничений; отсюда *эскапада* — веселая выходка.

**) В мои школьные годы (очень давно) код Морзе был популярен. Говорят, что теперь он известен даже меньше, чем перфокарты. Сэмюэль Морзе, американский изобретатель и художник (!), придумал способ передачи информации с помощью двух знаков — точек и тире. Каждой букве соответствовала своя последовательность, например е — точка, т — тире, а — точка и тире.

***) Предположение о независимости несущественно, если мы кодируем каждый символ отдельно, но, если ставится цель закодировать данный текст как можно короче, зависимости между символами, которые в настоящих текстах очень сильны, становятся важными. Позднее мы вернемся к этому вопросу.

один символ, т.е. математическое ожидание длины кодовой комбинации случайно выбранного символа, которое равно

$$\sigma = \sum_i p_i s_i,$$

где p_i — вероятность, а s_i — длина кодовой последовательности i-го символа.

Получается экстремальная задача.

Задача о префиксном коде. Минимизировать математическое ожидание σ по всем наборам длин $\{s_i\}$, удовлетворяющим неравенству Крафта. □

Набор, для которого σ минимально, называется *оптимальным*.

Шеннон и Фано предложили строить код, близкий к оптимальному, следующим способом: разбить все символы на две группы с приблизительно равными суммарными вероятностями, коды первой группы начать с 0, а второй группы — с 1; внутри каждой группы делать то же самое, пока в каждой группе не останется только по одному символу.

Элегантный алгоритм для точного решения этой задачи предложил Д. Хаффмен. Алгоритм основывается на нескольких очевидных свойствах оптимального набора $\{s_i\}$.

Лемма 1. Пусть $\{p_i\}$ — набор вероятностей символов и $\{s_i\}$ — длины оптимальных кодовых комбинаций. Если $p_1 \geq p_2 \geq \cdots \geq p_n$, то $s_1 \leq s_2 \leq \cdots \leq s_n$.

Доказательство. Здесь достаточно сослаться на уже встречавшуюся нам задачу о перестановке, минимизирующей скалярное произведение двух векторов (см. с. 42). □

Лемма 2. В обозначениях и предположении леммы 1 две самые длинные кодовые комбинации имеют одинаковую длину, т. е. $s_{n-1} = s_n$.

Доказательство. Действительно, пусть $s_n > s_{n-1}$ и, следовательно, n-я кодовая комбинация — самая длинная. Так как никакая кодовая комбинация не является началом никакой другой, то, сократив n-ю комбинацию до длины $(n-1)$-й, мы получим снова уникальную кодовую комбинацию и более короткую, чем раньше, что невозможно для оптимальной кодировки. □

Упражнение 5.1. В связи с доказательством леммы 2 мне задали вопрос, не будет ли нарушено при изменении длин кодовых комбинаций неравенство Крафта. Что бы вы ответили на этот вопрос?

Лемма 3. Рассмотрим наравне с исходной задачей P сокращенную задачу P', которая получается объединением двух самых редких символов в один символ, — в предположении леммы 1 это два последних символа с суммарной вероятностью $p'_{n-1} = p_{n-1} + p_n$. Минимальное значение целевой функции в задаче P' отличается от значения в задаче P на p'_{n-1}, а оптимальный кодовый набор для задачи P получается из решения для задачи P' удлинением на один бит кодовой последовательности для объединенного символа.

Доказательство. Действительно, каждому кодовому набору для задачи P' можно так, как сказано в утверждении леммы, сопоставить кодовый набор для задачи P с равными длинами самых редких символов. Это удлинение и приводит к приращению целевой функции на p'_{n-1}.

Теперь можно рассуждать так: в задаче P мы ищем минимум на множестве кодов, которое мы обозначим через C_n, а в задаче P' — аналогично на множестве C_{n-1}. В соответствии с леммой 2 в задаче P можно искать минимум на множестве C'_n, собственном подмножестве C_n, состоящем из таких наборов, в которых две самые длинные кодовые последовательности имеют одинаковую длину. Имеется взаимнооднозначное соответствие между C_{n-1} и C'_n, и значения целевых функций на соответствующих элементах двух множеств отличаются на постоянное слагаемое. Следовательно, и минимумы отличаются на это же постоянное слагаемое, так что минимуму в задаче P' соответствует минимум на C'_n, а он будет минимумом и для задачи P. □

Алгоритм, основанный на этих леммах, описывается в несколько строк: если в алфавите два символа, то нужно закодировать их 0 и 1, а если больше, то соединить два самых редких символа в один новый символ, решить получившуюся задачу и вновь разделить этот новый символ, приписав 0 и 1 к его кодовой последовательности.

При описании реализаций этого метода многие авторы пишут о том, что список символов должен быть упорядочен по вероятностям символов, и это упорядочение нужно обновлять после каждой склейки. Так поступать, конечно, неразумно. После начального упорядочения (символы располагаются по возрастанию их вероятностей) дополнительных упорядочений производить не требуется. Достаточно организовать еще один список для новых символов *). Добавление идет в конец этого списка, и возрастание вероятностей в нем обеспечивается само по себе. А при выборе двух символов с наименьшими

*) Этот прием предложили в 1960 г. студенты третьего курса матмеха А. Н. Терехов и П. Сёке.

вероятностями мы используем виртуальный список — результат слияния старого и нового списков. Поскольку размер нового списка известен, на один элемент короче исходного, такой список реализуется обычным массивом.

Пример. Пусть алфавит состоит из пяти символов — a, b, c, d, e, вероятности которых равны, соответственно,

$$0.37(a),\ 0.22(b),\ 0.16(c),\ 0.14(d),\ 0.11(e).$$

Объединяя d и e в один символ, получаем:

старый список: $0.16(c)$, $0.22(b)$, $0.37(a)$,
новый список: $0.25(de)$.

На следующем шаге:

старый список: $0.37(a)$,
новый список: $0.25(de)$, $0.38(bc)$.

Затем:

старый список: пуст,
новый список: $0.38(bc)$, $0.62(ade)$.

Сопоставим ade код 0, а bc — код 1 и выполним "обратный ход". Расщепим символ ade на a и de с кодами 00 и 01. Затем символ bc на b и c с кодами 10 и 11. Наконец, расщепив de, получим для исходного алфавита

$$00(a),\ 10(b),\ 11(c),\ 010(d),\ 011(e).$$

Обратите внимание на то, что в кодируемом тексте символы (в том числе составные) могут иметь равные вероятности, а из-за этого может существовать несколько различных оптимальных кодовых деревьев. У всех этих деревьев математическое ожидание числа битов на один символ будет одинаковым. (Докажите!)

Упражнение 5.2. Напишите программу для алгоритма Хаффмена.

Упражнение 5.3. Разработайте вариант метода Хаффмена для случая, когда каждый кодовый символ принимает не 2, а 3 (или произвольное число k) значений. Что изменится в методе?

Метод Хаффмена легко допускает всевозможные "усовершенствования". Например, при заметном преобладании какого-либо из символов, вызывающем его многократные повторы, можно сначала заменить последовательности повторов более экономной записью, а затем уже использовать метод Хаффмена.

Из рассмотрения задач поиска и кодирования следуют некоторые практические рекомендации качественного характера: полезно называть длинными именами те объекты в программах, которые редко используются, и короткими именами — используемые часто. Нужно располагать ближе и удобнее то, что используется чаще, и т. п.

5.3. Сжатие текстов

Первоначально возможность сжатия текстов была интересна только специалистам по телеграфной и радиосвязи (включая очень рано появившиеся работы по экономной передаче телевизионных изображений). С появлением персональных компьютеров программы сжатия текстов вошли в обиход практически всех потребителей вычислительной техники, а в последних версиях операционных систем, например в MS DOS начиная с версии 6.0, сжимающие программы становятся уже системной частью. Некоторые из схем сжатия информации запатентованы, и выпускаются реализующие их электронные устройства.

5.3.1. Сжатие по алгоритму Хаффмена

Для нас естественно начать обсуждение способов сжатия информации с уже знакомого кода Хаффмена. Вместо вероятностей, разумеется, используются частоты появления букв конкретного файла. Можно прямо посмотреть пример.

Пример. Исходный текст (223 знака, прописные буквы для простоты заменены строчными, знаки препинания и переводы строк удалены, межсловный пробел заменен подчеркиванием, в каждой строке по 50 символов):

```
ехали_медведи_на_велосипеде_а_за_ними_кот_задом_на
перед_а_за_ним_комарики_на_воздушном_шарике_а_за_н
ими_раки_на_хромой_собаке_волки_на_кобыле_львы_в_а
втомобиле_зайчики_в_трамвайчике_жаба_на_метле_едут
_и_смеются_пряники_жуют
```

По этому тексту получаем следующую статистику:

```
е 17  х 2  э 24  л 7  и 20  _ 40  н 12  д 7  в 9
н 11  о 13  с 4  п 3  з 6  к 11  т 7  р 7  у 3
ю 2  й 3  б 4  ы 2  ь 1  ч 2  ж 2  ш 2  я 2
```

или после сортировки по убыванию частот:

```
_ 40  э 24  и 20  е 17  о 13  н 12  н 11  к 11  в 9
л 7  д 7  т 7  р 7  з 6  с 4  б 4  п 3  й 3
у 3  х 2  ш 2  ы 2  ч 2  ж 2  ю 2  я 2  ь 1
```

Применение алгоритма Хаффмена дает:

```
ль 3  жю 4  ыч 4  хш 4  йу 6  пяь 6  сб 8  жюыч 8  зхш 10
йупяь 12  тр 14  лд 14  сбжюыч 16  взхш 19  нк 22  мйупяь 24
отр 27  лдсбжюыч 30  евзхш 36  инк 42  анйупяь 48
отрлдсбжюыч 57  евзхш_ 76  инканйупяь 90
отрлдсбжюычевзхш_ 133
```

и кодовые последовательности:

_ 111 а 010 и 000 е 1100 о 1000 м 0110 н 0010
к 0011 в 11010 л 10100 д 10101 т 10010 р 10011 з 110110
с 101100 б 101101 п 011110 й 011100 у 011101 х 1101110 ш 1101111
ы 1011110 ч 1011111 ж 1011100 ю 1011101 я 0111110 ь 0111111

Применяя эти кодовые последовательности для кодирования текста, получаем такой двоичный текст (длина строки — 64 бита):

```
1100110111001010100000111011011001010111010110010101000111001001
0111110101100101001000101100000011110110010101110011101011111011
0010111001000001100001110011100010010111110110010101011000011011
1001001001111011001001111001010111101011111101100101110010000011
1110011100001100101001100000110001110010010111110101000110110101
0101110111011110010100001101111101111010100110000011110011101011
1110110010111001000001100001111001101000110001110010010111110111
0100111000011010000111001111011001000101101010001111001111101010
0010100001100011100100101110011100001011011011110101001100111010
0011111111101010111101111101011101011010100101000011010001011010
0101001100111110110010011100101111100000110001111101011110010100
1101001101101001001110010111111000001111001111011100010101101010
1100100101110110110010010101001100111110010101011101100101110001
1110110001101100101110110010101100011111011101111010011011111000
1000000110001111011100011101101110110010
```

Получилось немного меньше 119 байтов, так что исходный текст сокращен почти вдвое (правда, если не считать самой кодовой таблицы).

По методу Хаффмена получается максимальное сжатие, и, казалось бы, ни о каком дальнейшем улучшении уже не может быть и речи. Вспомним, однако, что минимизируется математическое ожидание последовательности независимых символов, — код Хаффмена никак *не использует информацию о закономерностях чередования символов* в кодируемом тексте, закономерностях сложных и плохо описываемых математическими моделями.

Поэтому в качестве альтернативы были разработаны и используются *адаптирующиеся методы*, в которых упомянутые закономерности выявляются "сами собой". Один из таких методов мы сейчас и рассмотрим.

5.3.2. Сжатие по методу Зива–Лемпеля

В этом методе кодируемый текст разбивается на небольшие строки, каждая из которых составлена из одной из предыдущих строк (предопределена пустая строка, имеющая номер 0) и еще одного символа.

По методу Зива–Лемпеля*) тот же пример кодируется следующим образом. Сначала записаны номера строк, затем сами строки, затем представление каждой из этих строк в виде пар (номер предыдущей строки, дополняющий символ). В начале образовалось семь строк длины 1. Восьмая буква совпала с первой строкой, поэтому восьмая строка составлена из первой строки и буквы д. Девятая строка состоит из одной буквы, а десятая — из восьмой строки и буквы и. Дальше смотрите сами.

```
1  2  3  4  5  6  7  8  9  10  11 12 13 14 15 16 17  18 19 20
е  х  а  л  и  _  м  ед в  еди _и a_ ве ло с  ип еде _а _з a_и
0е 0х 0а 0л 0и 0_ 0м 1д 0в 8и  6н 3_ 9е 4о 0с 5п 8е  6а 6з 12и

21 22 23 24 25 26  27 28  29  30 31 32  33  34   35 36 37 38 39
ни и_ к  о  т  _за д  ом  _на п  ер ед_ a_з a_ни м_ ко ма р  ик
5и 5_ 0к 0о 0т 19а 0д 24м 11а 0п 1р 8_  12з 20и 7_ 23о7а 0р 5к

40  41 42  43  44 45 46  47  48  49  50  51 52    53  54 55
и_н a_в оз  ду  м  н  ом_ ма  ри  ке  _a_ з  a_ним и_р ак и_на
22и 12в 24з 27у 0м 0и 28_ 44а 38и 23е 18_ 0з 34м  22р 3к 40а

56 57  58 59 60 61  62  63 64  65  66   67  68 69 70 71 72 73
_х ро  мо й  _с об  ake _в ол  ки  _на_ коб ы  ле _л ь  вы _в_
6х 38о 7о 0й 6с 24б 54е 6в 24л 23и 29_  36б 0ы 4е 6л 0ь 9м 63_

74 75  76  77 78 79  80  81  82   83  84 85 86 87  88 89 90
ав то  моб ил e_ за  йч  ики _в_т ра  ив ай ч  ике _ж аб a_на
3в 25о 58б 5л 1_ 51а 59ч 39и 73т  38а 7в 3й 0ч 39е 6ж 36 20а

91 92 93  94   95  96  97 98 99  100 101 102 103 104 105 106
_и ет ле_ eду  т_  и_с ме ю  тс  я   _п  ря  ни  ки_ ж   у
6и 1т 69_ 8у   25_ 22с 7е 0ю 25с 0я  6п  38я 45и 65_ 0ж  0у

107 108  109
ат  '\n' EOF
98т 0    0
```

Полезно взглянуть на рис. 5.1, где показаны длины получающихся строк и их изменяющаяся средняя длина. Вы видите, что только около семидесятой строки средняя длина строки превосходит 2 и при записи каждого номера строки в один байт получается небольшой выигрыш. Правда, при увеличении длины кодируемого текста выигрыш растет, а закодированную информацию можно еще дополнительно сжимать.

*) J. Ziv, A. Lempel [87]. Приводимый здесь алгоритм называется LZ78; он описан во второй из двух ссылок.

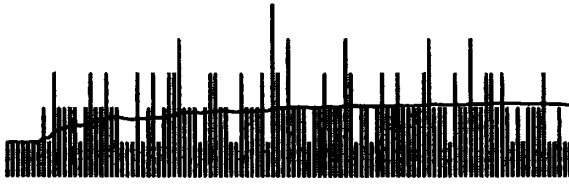


Рис. 5.1. Длины кодовых строк и поведение средней длины

В настоящее время повсеместно применяется модификация этого метода, предложенная Велчем *). В этой модификации, известной как LZW-метод, исходные символы текста получают свои кодовые номера априори, например номерами могут быть ASCII-коды символов, а встретившиеся в тексте цепочки символов заносятся в кодовую таблицу для возможного дальнейшего использования. Рассмотрим метод LZW на том же примере (начало процесса):

```
   е                          букву 'е' сохраняем как префикс
е  х  [е]  ех   256   строка 'ех' не встречалась, регистрируем
                      ее с кодом 256, код буквы 'е' выводим в
                      результат, букву 'х' запоминаем как новый
                      префикс
х  в  [х]  хв   257   аналогично
в  л  [в]  вл   258   то же
л  и  [л]  ли   259   то же
и  _  [и]  и_   260   то же
_  м  [_]  _м   261   то же
м  е  [м]  ме   262   то же
е  д  [е]  ед   263   то же
д  в  [д]  дв   264   то же
в  е  [в]  ве   265   то же
е  д                  строка 'ед' в кодовой таблице уже есть,
                      продолжаем процесс с увеличенным префиксом
ед и  263  еди  266   регистрируем 'еди', выводим код строки 'ед'
и  _       и_         строка 'и_' есть
и_ н  260  и_н  267   регистрируется 'и_н', выводим код 'и_'
н  а  [н]  на   268   снова как раньше
а  _  [а]  а_   269   то же
_  в  [_]  _в   270   то же
в  е                  строка есть
ве л  265  вел  271   регистрируем 'вел', выводим код 'ве'
л  о  [л]  ло   272   как раньше
о  с  [о]  ос   273   то же
```

*) T. A. Welch [86].

Здесь в первой колонке мы видим префикс рассматриваемой строки (в начале процесса он пуст), во второй колонке — очередной символ входного текста. Если текущая строка — конкатенация префикса и символа — уже включена в кодовую таблицу, она становится новым префиксом, в противном же случае она регистрируется, префикс (непустой) выводится в результат, а символ заменяет его как новый префикс.

При сравнительно небольших затратах на одну кодовую комбинацию получается достаточно большой диапазон номеров, например, при 12 битах (полтора байта) можно закодировать примерно 4000 строк и не требуется никаких дополнительных затрат на таблицу исходных символов.

Важное преимущество LZ-методов и в том, что они адаптивны, т. е. сами приспосабливаются к особенностям текста и не требуют предварительного просмотра, как метод Хаффмена. Правда, существуют и адаптивные варианты кодирования по Хаффмену. Эти и другие методы можно найти в обзорах по сжатию данных, например в [75].

Сейчас массовое использование персональных компьютеров сделало популярными многочисленные "архивирующие программы", среди них наиболее известны `arj`, `zip`, `zoo`, `rar`, а в среде UNIX — программы `compress`, `gzip`, `compact`. Популярна сжатая система хранения информации на диске. В системе программирования JAVA предусмотрено хранение в сжатой форме исполняемых файлов. Алгоритмическую основу архивирующих программ образуют методы Хаффмена и Зива–Лемпеля с различными модификациями и усовершенствованиями *).

5.3.3. Метод Барроуза–Уилера

Метод Барроуза–Уилера **) появился только в 1994 г. Изложение принципов его работы будет опираться на достаточно большой пример. Возьмем фразу

```
карл-у-клары-украл-кораллы-а-клара-у-карла-украла-кларнет$
```

(здесь удобнее обозначать пробелы не подчеркиванием, а минусом).

*) Очень интересный обзор методов сжатия был сделан студентом ИТМО В. В. Семенюком [45]. Эта работа кроме изложения последних методов сжатия из литературы содержит и изложение нового конкурентоспособного метода, разработанного автором.

**) M. Burrows, D. J. Wheeler [58]. Я взял этот метод из упомянутой книги В. В. Семенюка, но переделал изложение, согласуя его с другими частями курса.

Для нее на с. 104 мы построили суффиксное дерево. Напомним, что это дерево определяет лексикографическое упорядочение суффиксов рассматриваемой строки. Возьмем для каждого суффикса предшествующий ему символ (для полной строки в качестве "предшествующего" символа возьмем $) и составим из этих символов новую строку, которую назовем S_2. Ее длина будет такой же, как у исходной строки:

```
тыуааулалыа-арлррралккллн$-----ууараркккалркаккоаааае----ар
```

Покажем, как по этой строке можно восстановить исходную строку. Прежде всего, ясно, что по ней можно восстановить строку S_1, составленную из первых символов суффиксов. Действительно, по S_2 определяется буквенный состав S_1, а расположение букв в строке S_1 подчиняется алфавитному порядку. Так что S_1 не просто восстанавливается, а всего за два просмотра S_2 — за линейное время. Для удобства выпишем S_1 под S_2:

```
тыуааулалыа-арлррралккллн$-----ууараркккалркаккоаааае----ар
$----------аааааааааааеккккккккллллллллнорррррррртуууыы
```

Теперь будем восстанавливать исходную строку. Суффиксу $ в строке S_2 соответствовал последний символ исходной строки, поэтому символ т распознается просто. Ему предшествует однозначно определяемое е, перед которым идет н, а перед ним р. Но у нас в строке S_1 восемь символов р, и чтобы понять, какой символ следует перед н, мы должны разобраться, какой же из них последний. Вспомним, что восемь символов р — это начальные символы лексикографически упорядоченных суффиксов. Пара строк (S_2, S_1) показывает нам, что четыре из этих суффиксов начинаются с ра, два с рл, один с ри и один с ры. И именно в таком порядке суффиксы и расположены. Отсюда ясно, что нам нужен предпоследний суффикс, и в строке S_1 мы должны выбрать предпоследнее р. Ему предшествует символ а. Несколько уточняя рассуждения, мы видим, что нам нужно предпоследнее а. Действительно, суффиксы, начинающиеся с а, идут в строке S_2 в лексикографическом порядке, так как у них первая буква одинакова, а продолжения расположены в лексикографическом порядке в S_2. Предпоследнему а предшествует буква л. Это шестая буква л, ей предшествует к и т. д.

На самом деле строка S_1 нам нужна только для понимания механизма. Для декодирования же лучше (и проще) вычислить заранее, к какому символу следует переходить от любого i-го символа строки S_2.

Итак, будем разбирать наш механизм. Каждому символу строки S_1 однозначно соответствует некоторый суффикс, строка S_1 задает нумерацию суффиксов, расставленных в лексикографическом порядке. Наращивание этих суффиксов дополнительным символом (из строки S_2) образует также суффиксы. Суффиксы, наращиваемые одним и тем же символом, будут, очевидно, расположены также в лексикографическом порядке — у них головы совпадают, а хвосты упорядочены. Разобьем множество суффиксов на подмножества

"с одинаковыми головами", сохраняя в каждом подмножестве лексикографический порядок:

```
$ ---------- аааааааааа е ккккккк ллллллл н о ппппппп т уууу ыы
к акккккуууу ----ллпппппп т ааллопр --ааааалы е р аааалльы $ --кк --
```

Естественно, что их общий порядок совпал с порядком в S_1, и этот порядок и нужно сопоставить элементам строки S_2. Это нужно делать так, как описано на с. 14: перенумеровать суффиксы в каждом из подмножеств разбиения суффиксов по начальному символу и из локальных нумераций составить общую. Нумерация строится за те же, что и раньше, два просмотра строки S_2. При первом просмотре для каждого символа алфавита подсчитаем количество его появлений в строке S_2 и образуем накопленные суммы $a[s]$:

s	-	а	е	к	л	н	о	р	т	у	ы
1	10	12	1	8	9	1	1	8	1	4	2
1	2	12	24	25	33	42	43	44	52	53	57

При втором просмотре для каждого i определяем его номер $p[i]$. Именно, встретив на i-м месте символ s, полагаем $p[i]$ равным $a[s]$, а $a[s]$ увеличиваем на единицу. В нашем примере мы получим:

т	ы	у	а	а	у	л	а	л	ы	а	-	л	р	л	р	р	р	л	к
52	57	53	12	13	54	33	14	34	58	15	2	35	44	36	45	46	47	37	25
к	л	л	н	$	-	-	-	-	-	у	у	а	р	а	р	к	к	к	а
26	38	39	42	1	3	4	5	6	7	55	56	16	48	17	49	27	28	29	18
л	р	к	а	к	к	о	а	а	а	а	е	-	-	-	-	л	р		
40	50	30	19	31	32	43	20	21	22	23	24	8	9	10	11	41	51		

Эти номера и будут позициями перехода от каждого суффикса к предыдущему.

Остается вопрос: как этот замечательный факт использовать? Понятно, что хочется передавать вместо исходной строки строку S_2. Но почему это будет экономнее?

Здесь нам поможет прием, который называется MTF*). Будем выписывать символы строки S_2 по мере их появления, кодируя каждый новый символ нулем. Каждый уже встречавшийся символ кодируется его местом в строке. Суть метода в том, что символы в строке все время переставляются — использованный помещается наверх. При этом номера символов соответственно

*) От Move To Front. Так его назвали американские авторы [57]. В [45] отмечается, что в 1980 г. этот метод опубликовал Б. Я. Рябко [44], назвав его *методом стопки книг*. Я вспоминаю, что в мои студенческие годы (много раньше) один из моих коллег использовал аналогичную идею при управлении своей библиотекой, в которой все книги умещались на одной полке. Новые книги и книги, которые для чего-нибудь понадобились, ставились слева, а книги, выходившие за правый край, сдавались в "Старую книгу". Я не следую названным статьям буквально, а использую только идею.

изменяются. Вот так:

```
т  0  т              к  1  клр-ыут        л  8  лакру-$ныт
ы  0  ыт             л  2  лкр-ыут        р  4  рлаку-$ныт
у  0  уыт            л  1  лкр-ыут        к  4  крлау-$ныт
а  0  ауыт           н  0  нлкр-ыут       а  4  акрлу-$ныт
а  1  ауыт           $  0  $нлкр-ыут      к  2  карлу-$ныт
у  2  уаыт           -  6  -$нлкраыут     к  1  карлу-$ныт
л  0  луаыт          -  1  -$нлкраыут     о  0  окарлу-$ныт
а  3  алуыт          -  1  -$нлкраыут     а  3  аокрлу-$ныт
л  2  лауыт          -  1  -$нлкраыут     а  1  аокрлу-$ныт
ы  4  ылаут          -  1  -$нлкраыут     а  1  аокрлу-$ныт
а  3  аылут          у  9  у-$нлкраыт     а  1  аокрлу-$ныт
-  0  -аылут         у  1  у-$нлкраыт     е  0  еаокрлу-$ныт
л  4  л-аыут         а  8  ау-$нлкрыт     -  8  -еаокрлу$ныт
р  0  рл-аыут        р  8  рау-$нлкыт     -  1  -еаокрлу$ныт
л  2  лр-аыут        а  2  ару-$нлкыт     -  1  -еаокрлу$ныт
р  2  рл-аыут        р  2  рау-$нлкыт     -  1  -еаокрлу$ныт
р  1  рл-аыут        к  8  крау-$нлыт     л  7  л-еаокру$ныт
р  1  рл-аыут        к  1  крау-$нлыт     р  7  рл-еаоку$ныт
л  2  лр-аыут        к  1  крау-$нлыт
к  0  клр-аыут       а  3  акру-$нлыт
```

По строке появления символов `тыуал-ркн$ое` и номерам позиций легко восстановить строку. А сами номера имеют очень специфичные частоты и эффективно кодируются, например методом Хаффмена:

```
 0  1  2  3  4  5  6  7  8  9
12 19  9  4  5     1  2  5  1
```

Упражнение 5.4. Восстановите исходную строку по строке

```
тсайувьаяктнг--оянзн$лед-ме---бп-оаеа-ишлр
```

5.3.4. Еще о сжатии

Нужно учитывать, что сжатие может применяться в различных ситуациях, где требования, предъявляемые к выбираемым схемам, сильно меняются. Прежде всего, могут различаться статистические свойства сжимаемого материала: например, файлы текстов, программ, звука и картинок имеют совершенно разные свойства.

Кроме того, могут различаться условия на время, которое разрешается потратить на кодирование и декодирование. Обычно мы используем быстрое сжатие и быструю распаковку. Но бывают случаи, когда паковать позволяется долго, а распаковывать нужно быстро.

Пример. У меня потребность в таком сжатии встретилась, когда мы со студентом матмеха А. Ю. Бураго разрабатывали программную систему для обучения языку ФОРТ*) на очень маленькой, даже по тем временам, машине — в ней было всего 32 Кбайта памяти. Из этой памяти 8К занимал сам ФОРТ, и учебник делился на отдельные уроки, каждый примерно из 20 экранов по 2К символов в каждом экране. Очевидная арифметика $2 \times 20 = 40$ показывает, что без сжатия было не обойтись.

Понятно, что в подобной ситуации важно время "распаковки" одного экрана, а время "запаковки" почти несущественно.

Мы представили экран в виде последовательности слов, причем слово понималось как любая последовательность непробелов, и для каждого слова запоминали, сколько перед ним пробелов, считая при этом экран сплошной строкой длиной в 2000 символов. Составив общий для всех экранов одного урока словарь (конечно, в алфавитном порядке, но различая строчные и прописные буквы), мы сопоставили каждому слову его порядковый номер в словаре и представили информацию об экране как последовательность пар чисел (число пробелов, номер слова). Не буду описывать, как кодировалась эта информация в те времена, сейчас я закодировал бы ее по методу Хаффмена, кодируя отдельно длины пробелов и номера слов. (А почему?)

Словарь мы также сжали, используя так называемое сжатие слева. Словарь (кроме однобуквенных слов, занявших все номера от 0 до 255, и двухбуквенных, перечисленных непосредственно) был разбит на группы по восемь слов. (Почему по восемь? Потому, что восемь в сравнении с четырьмя давало еще большую экономию места, а 16 в сравнении с восемью место сокращает мало, но время декодирования увеличивает вдвое.) В каждой группе каждое слово запишем (тогда мы делали немного иначе) как число символов, совпадающих с началом предыдущего слова, и конец слова, заканчивающийся символом конца строки (принято использовать нулевой код, здесь мы его обозначим звездочкой). Например, как это делается в группе, показанной в таблице:

персик	0	персик*	перуанка	3	уанка*
персика	6	а*	перуанки	7	и*
персиков	6	ов*	песок	2	сок*
персоль	4	оль*	пест	3	т*

Такой способ сжатия очень выгоден для длинных "профессиональных" слов, различающихся окончаниями. Получающийся "текст" словаря можно также закодировать кодом Хаффмена, причем и здесь выгодно кодировать отдельно

*) Для тех, кто впервые слышит про этот язык: FORTH — очень компактный язык программирования, в конструкции которого большую роль играют стеки. Много общего с ФОРТом имеет ПостСкрипт (PostScript), широко используемый сейчас язык графического вывода. Что же касается нашего электронного учебника, то в дальнейшем по его материалам мы выпустили брошюру [9].

числа и буквы. Частоты букв в сжатом словаре, и в несжатом тоже, могут существенно отличаться от частот в тексте. Почему?

Упражнение 5.5. Допустим, каждая группа слов в словаре должна содержать не ровно восемь слов, а не более восьми. Предложите алгоритм для разбиения словаря на k групп с наибольшим сжатием текста. (Совет: используйте динамическое программирование.)

Аналогичные задачи сейчас характерны для записи информации на компакт-дисках. В последние годы для нужд этого приложения были разработаны чрезвычайно эффективные методы сжатия аудио- и видеофайлов. Характерный пример: на обычном компакт-диске нормально умещается около 70 минут музыки. Вместе с тем формат `mpeg-3` позволяет записать на таком же диске больше 10 часов *).

При кодировании графической информации — тоновых изображений и особенно последовательностей изображений, необходимых для "кино", — используют другие специальные приемы сжатия, часто с некоторой потерей информации. Мы не будем углубляться в эту тему, для справки лишь упомянем форматы `jpeg` и уже названный `mpeg`.

5.4. Избыточное кодирование

При выборе системы кодирования приходится оценивать ее по различным характеристикам, а не только по длине кодированного текста, которую приходится несколько увеличивать, уступая другим надобностям. Рассмотрим некоторые из них.

5.4.1. Преобразование в видимый формат

При передаче произвольной 8-битовой информации по 7-битовым каналам приходится как-то преобразовывать текст, чтобы данные не исказились. Часто используется процедура, переводящая 8-битовые коды в "видимые" символы — заглавные латинские буквы, цифры, знаки препинания. Конечно, можно просто писать шестнадцатеричные коды соответствующих символов, как это делается, например, в `rtf` — одном из форматов Микрософта: `\'e0` для `а`, `\'e1` для `б` и т.д. Но такое кодирование невыгодно — каждый символ заменяется тремя.

Сейчас получили большую известность экономные способы кодирования, которые используются в передаче произвольных бинарных

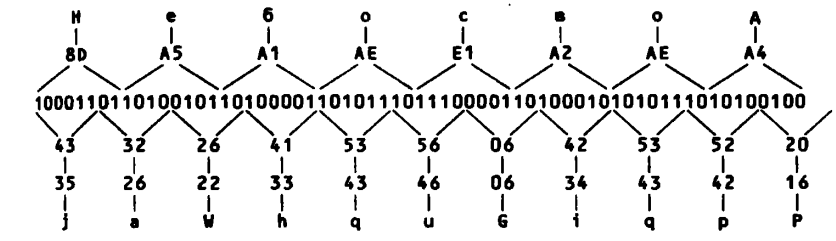
*) Сейчас мир заполнен уже "mp3-музыкой" и "mp4-фильмами".

файлов (например, программ) по 7-битовым каналам телекоммуникаций.

Например, в 6-битовом формате MIME64 *) исходный "текст" произвольной природы, т. е. состоящий из произвольных 8-битовых байтов, делится на кусочки по 6 битов, каждый из них трактуется как число в диапазоне 0 : 63, а это число заменяется "цифрой" — символом из строки "видимых символов":

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
      1       2       3       4       5       6       7       8
```

Вот небольшой пример перевода ASCII-текста в формат MIME64:



В первой строке записан исходный текст, во второй каждый символ переведен в две шестнадцатеричные цифры, в третьей каждая из них представлена в двоичном виде. В четвертой строке кусочки разбиения переведены в восьмеричные цифры, в пятой из них сделаны десятичные числа, и наконец в шестой строке текст, готовый для передачи, составлен из наших "цифр".

Декодирование выполняется в обратном порядке **).

5.4.2. Помехоустойчивость

При передаче данных (по сетям связи и внутри компьютера) возможны ошибки. Между тем ошибка даже в одном двоичном разряде может полностью исказить передаваемый текст, если это программа или сжатый по Хаффмену текст.

Упражнение 5.6. Попробуйте изменить один бит в рассмотренном выше примере и посмотрите, что получится.

*) Multipurpose Internet Mail Extensions — многоцелевые расширения интернетовской почты. Примерно так же работает и другой способ кодирования в процедуре UUENCODE.

**) Мне иногда приходится пользоваться самодельной декодирующей программой. Такая необходимость возникает при появлении закодированного текста вне обычного электронного письма.

Для уменьшения числа ошибок информацию часто передают с дополнительными проверочными данными — *контрольными суммами*.

Самый простой вариант контрольной суммы используется при передаче отдельных байтов. В записи байта выделяется один специальный разряд, *бит чётности* *) — в некоторых случаях это восьмой бит, но чаще девятый, специально добавляемый в аппаратуре и невидимый для пользователя. Назначение этого бита — дополнять число единиц в данном байте по выбору до четного или до нечетного числа. При каждой передаче байта внутри компьютера передаются 9 битов, и четность переданного байта контролируется.

Наряду с этим контролем, *встроенным в аппаратуру*, предусмотрены разнообразные системы контроля *в программных средствах* передачи и обработки информации. Например, при передаче информации в системе электронной почты и в Интернете предусматриваются контрольные суммы у каждого передаваемого "пакета". Сейчас контрольное суммирование используется и для распознания заражения компьютера вирусами — у зараженных программ изменяется контрольная сумма.

Иногда при передаче данных используется специальное *защитное кодирование*, которое позволяет не только проверять правильность передачи данных, но и восстанавливать правильный текст в случае редких сбоев (отметим, что при случайных помехах абсолютной защиты нет и быть не может, речь идет только о повышении надежности передачи).

Очень прост и нагляден код Хэмминга **), который позволяет найти ошибку в последовательности битов при условии, что ошибок не больше одной. Рассмотрим этот способ кодирования. Если требуется передать n битов, а передается N, где $N \geq n$, битов, то для возможности обнаружения ошибки нужно, чтобы в переданном тексте, кроме n битов основного текста, хватало места и для распознания одной из $N + 1$ возможностей положения места ошибки (нулевая возможность — текст передан без ошибок). Таким образом, $2^{N-n} \geq N + 1$. Видно, что необходимое приращение числа контрольных битов логарифмически зависит от длины кодовой последовательности: $N - n \approx \log_2 N$. Вот табли-

*) Мы здесь поставили букву *ё* после того, как некоторые студенты стали говорить на экзамене о *бите честности*.

**) Richard Wesley Hamming (1915–1998), современный американский математик, он одним из первых начал заниматься созданием кодов с исправлением ошибок. Его имя сохраняется также в названии *расстояния Хэмминга*, меры различия двух 0–1 векторов, равной числу позиций, в которых эти векторы различаются.

ца, показывающая максимальное значение n для некоторых значений $N - n$:

$N - n$	2	3	4	5	6	7	8	9
n	1	4	11	26	57	120	247	502

Покажем, что при достаточном добавлении битов можно сделать очень простой код, обнаруживающий один ошибочный бит. Пусть задано некоторое n и N выбрано так, что выполнено приведенное неравенство. Передаваемые биты перенумеруем от 1 до N и, зарезервировав номера 1, 2, 4, 8 и т.д. (степени числа 2), все остальные места сопоставим местам в исходной последовательности. Сформируем матрицу контрольного суммирования, составленную из нулей и единиц, в этой матрице N столбцов и $N - n$ строк, каждый ее j-й столбец содержит двоичное представление числа j. Эта матрица будет умножаться на передаваемый вектор. Выберем значения в резервных столбцах так, чтобы все компоненты вектора-произведения равнялись нулю по модулю 2 (дальше вы увидите, что все это очень просто).

Сформированный таким образом вектор передается по каналам связи и при получении умножается на ту же матрицу. Вектор остатков от деления его компонент на 2 может рассматриваться как двоичное разложение некоторого целого числа k. Если оно равно 0, информация передана без ошибок. Если больше нуля, то число k дает номер бита с ошибкой.

Давайте начнем выяснение того, почему эта конструкция работает, с небольшого примера.

Пример. Пусть $n = 10$ и, следовательно, $N = 14$. Составим таблицу, включающую матрицу суммирования и связь начальных номеров битов i с передаваемыми j:

i			1		2	3	4		5	6	7	8	9	10	s_0	s_1	s_r
j	1	2	3	4	5	6	7	8	9	10	11	12	13	14			
$a[1]$	1		1		1		1		1		1		1		0	0	1
$a[2]$		1	1			1	1			1	1			1	1	0	0
$a[3]$				1	1	1	1					1	1	1	1	0	1
$a[4]$								1	1	1	1	1	1	1	1	0	0
b			0		0	1	1		1	0	1	0	1	0			
β	0	1		1				1									
b_t	0	1	0	1	0	1	1	1	1	0	1	0	1	0			
b_r	0	1	0	1	1	1	1	1	1	0	1	0	1	0			

Пусть требуется передать, например, последовательность 0011101010. Поставив эти биты на их места в расширенной последовательности, образуем "частичный" вектор b и, умножая его на соответствующую часть матрицы a, получаем столбец s_0. По этому столбцу находим значения резервных битов (строка β). Готовый к передаче набор записан в строке b_t, а его контрольные суммы (нулевые) — в столбце s_t. Предположим, что при передаче этой последовательности вместо нее была получена последовательность b_r с ошибкой в пятой позиции. Эта ошибка изменит результат суммирования в первой и третьей строках, так как именно в этих строках столбец 5 имеет единицы. Мы получим набор сумм $(1,0,1,0)$ — двоичное разложение числа 5 (в машинной нумерации разрядов).

В общем случае происходит то же самое: резервированные биты используются для того, чтобы сделать нулевыми все "контрольные суммы". При этом каждый контрольный бит входит только в одну сумму и его значение определяется независимо от всех остальных, количество этих битов достаточно (по выбору $N - n$) для двоичного представления всех чисел от 1 до N. Ошибка при передаче любого j-го бита, $j \in 1:N$, добавляет к набору контрольных сумм (нулевому) в точности j-й столбец матрицы, т. е. в точности двоичное разложение числа j.

Размер передаваемого пакета N выбирается исходя из вероятности p ошибки при передаче одного бита: вероятность хотя бы одной ошибки в пакете будет равна $1 - (1-p)^N \approx Np$ и нужно, чтобы эта вероятность была достаточно малой. Существуют варианты метода Хэмминга, предназначенные для исправления и больше чем одной ошибки, но их рассмотрение в наши задачи не входит.

5.5. Криптография

Все мы знаем о различных формах *криптографии* [*], или *тайнописи*, — специальных способах шифровки сообщений для того, чтобы оградить информацию от нежелательного (говорят, от несанкционированного) разглашения. По "Золотому жуку" Эдгара По и "Пляшущим человечкам" Артура Конана Дойла мы знакомились с "подстановочным шифром", в котором каждая буква документа заменяется однозначно определяемой другой буквой или специальным знаком из другого алфавита.

Криптография широко применяется и становится все более популярной из-за распространения информационных систем. Чем же привлекательна крип-

[*] Этот параграф написан совместно с С. В. Кузнецовым.

тография? Тем, что зашифрованные сообщения можно пересылать по открытым каналам связи, не боясь "прочтения" этой информации третьими лицами.

Все криптографические системы, вне зависимости от их сложности, имеют следующие составные части.

(Исходное) Сообщение — то, что мы хотим защитить от несанкционированного чтения/использования. В качестве *сообщения* может выступать как произвольный текст, так и программное обеспечение, различная медиа-информация.

Зашифрованное сообщение — это сообщение, измененное с целью скрыть его исходный смысл и сделать его "нечитаемым". Процесс преобразования сообщения в зашифрованное сообщение называется *шифрованием*, обратный процесс — *дешифрованием*.

Криптографический алгоритм — некий математический алгоритм, используемый для шифрования или дешифрования исходного сообщения.

Ключ — вспомогательная информация, используемая криптографическим алгоритмом. Она секретна, и априори считается, что, только обладая ею, можно восстановить исходное сообщение.

Зашифрованное сообщение может быть "взломано", т. е. прочтено без знания ключа, несколькими способами. Один из них — криптоанализ, основное назначение которого состоит в анализе зашифрованного сообщения и в поиске в нем различных особенностей, например повторяющихся частей. Другим вариантом является подбор ключа для дешифрования до тех пор, пока в результате дешифрования не получится "читаемое" сообщение.

5.5.1. Симметричное шифрование

Наиболее известными и распространенными криптографическими алгоритмами были так называемые *симметричные* алгоритмы, в которых один и тот же ключ используется для шифрования и дешифрования сообщений (и именно поэтому они так называются). Главный недостаток симметричных систем в том, что стороны, обменивающиеся информацией, должны договориться об использовании одного и того же ключа, а также хранить его в тайне от остальных. А значит, при обмене информацией со многими сторонами необходимо запоминать, где какой ключ используется, и ключ не может быть передан через обычные открытые средства связи из-за опасности его перехвата.

Среди наиболее известных и распространенных алгоритмов можно выделить DES (Data Encryption Standard), AES (Advanced Encryption

Standard), RC2, RC4, RC5, IDEA (International Data Encryption Algorithm) и Blowfish *).

Многие из используемых алгоритмов опираются на два очень простых приема: подстановка и перестановка.

Подстановочные шифры разгадываются легко, достаточно пары строк текста, чтобы оценить частоты появления отдельных букв, а после нескольких угадываний остаток текста читается без труда **).

Очень эффективный и вычислительно простой метод кодировки получается при побитовом сложении кодируемого текста с псевдослучайной последовательностью битов. Именно, пусть имеется последовательность битов $\{a_i\}$, полученная в последовательности независимых случайных испытаний, где 0 и 1 появляются равновероятно, или любой другой последовательности с похожими свойствами. Пусть $\{b_i\}$ — последовательность битов, составляющая передаваемый текст. Образуем последовательность $\{c_i\}$, где $c_i = a_i \not\equiv b_i$, которая и посылается адресату. Если последовательность $\{a_i\}$ ему известна, то, применяя ту же операцию, он легко получит исходный текст ***).

Пример. Возьмем в качестве последовательности $\{a_i\}$ разложение числа e^{-1} в виде двоичной дроби [23]:

```
.0101 1110 0010 1101 0101 1000 0001 1000 1011 0011 1011 1100
 1101 1111 0001 1010 1011 1010 1101 1110 1100 0111 1000 0010
 1001 0000 0101 0100 1111 1001 0000 1101 1101 101
```

*) DES был разработан фирмой IBM и утвержден правительством США в 1977 г. как официальный стандарт. Стандарт AES принят недавно для официальной замены DES в связи с тем, что выросла мощность компьютеров и повысилась вероятность "взлома". Немного сравнительной статистики об этих алгоритмах можно найти в конце параграфа.

Блочные шифры с ключом переменной длины были созданы Р. Ривестом для RSA Data Security Inc. Алгоритм RC4 широко используется в практических приложениях, в частности при передаче шифрованных сообщений в Интернете и по мобильным телефонам.

Алгоритм IDEA был разработан для простого программного и аппаратного воплощения; его безопасность основывается на использовании трех не совместимых типов арифметических операций над 16-битовыми словами. IDEA распространен в Европе и используется в программе шифрования электронных писем PGP (Pretty Good Privacy).

Алгоритм Blowfish разработан Брюсом Шнайером (Bruce Schneier), автором известной книги "Прикладная криптография", в 1993 г. Он существенно быстрее DES.

**) Иногда приходится заниматься такой работой при каких-нибудь неполадках в настройках электронной почты, если письмо приходит в необычной кодировке. Вот реальный пример из моей переписки — попробуйте его расшифровать, зная, что адресата зовут Иосифом Владимировичем:

```
=E9=CF=D3=C9=C6 =F7=CC=C1=C4=C9=CD=C9=D2=CF=D7=C9=DE,
=D1 =C4=CF=D3=D4=C1=CC=C1 =CE=D5=D6=CE=D9=CA =F7=C1=CD =C6=C1=CA=CC.
=EE=C1=D4=C1=DB=C1.
```

***) С. Синх [81] утверждает, что именно такой код используется в "горячей линии Белый дом—Кремль".

или 16-ричной дроби (три последних бита отбросим):

```
.5E2D 5818 B3BC DF1A BADE C782 9054 F90D D
```

Этой последовательности (135 битов) достаточно для кодирования текста из 16 байтов. Закодируем фразу "Petersburg, 2003":

Текст	P	e	t	e	r	s	b	u	r	g	,	_	2	0	0	3
ASCII	50	65	74	65	72	73	62	75	72	67	2C	20	32	30	30	33
B	05	56	47	56	27	37	26	57	27	76	C2	02	23	03	03	33
A	5E	2D	58	18	B3	BC	DF	1A	BA	DE	C7	82	90	54	F9	0D
C	5B	7B	10	4E	94	8B	F9	4C	9E	C8	05	80	B3	57	FA	3E

Другим семейством простейших криптографических алгоритмов являются перестановочные шифры. Основная идея заключается в изменении порядка чтения символов передаваемого текста. Так, типичным примером являются "шифры-решетки", очень популярные в средневековой конфиденциальной переписке. Суть заключается в том, что обе стороны обладают своеобразным бумажным трафаретом, в котором сделаны прорези достаточных размеров для написания одной или нескольких букв. Создавая зашифрованное сообщение, отправитель накладывает трафарет на лист бумаги и пишет нужный текст в прорезях. После заполнения свободных мест трафарет перемещается, закрывая уже написанный текст и открывая новое пространство. После того как текст вписан, все место, оставшееся свободным, заполняется произвольными символами. Для прочтения этого сообщения необходимо наложить тот же трафарет и точно так же перемещать его.

В этом примере ключ состоит из трафарета и порядка его перемещения. Поэтому, очевидно, данный алгоритм является алгоритмом симметричного шифрования.

Описанные выше перестановочные и подстановочные шифры достаточно легко "взломать", т.е. подобрать нужный ключ или же применить различные приемы криптоанализа. Однако на их основе можно сформировать достаточно сложные и "стойкие" шифры. Например, алгоритм DES, который оперирует входными данными из блоков по 64 бита. Мы опишем один из шагов алгоритма DES, чтобы показать, что ничего, кроме подстановки и перестановки, в нем не используется.

Пример. **Один из шагов алгоритма DES**

1. Входной блок данных делится пополам на левую (L') и правую (R') части. После этого формируется выходной массив так, что его левая часть (L'') представлена правой частью (R') входного, а из 32-битового слова R' с помощью битовых перестановок формируется 48-битовое слово.
2. Полученное 48-битовое слово "XOR-ится" с 48-битовым "раундовым" ключом. Раундовые ключи получаются из исходного ключа путем отбрасывания восьми битов.

3. Результирующее 48-битовое слово разбивается на восемь 6-битовых групп, каждая 6-битовая группа посредством специальных таблиц заменяется на 4-битовую группу. Из полученных 4-битовых групп составляется 32-битовое слово.

4. Полученное слово "XOR-ится" с L'; в результате получается R''.

Можно убедиться, что проведенные операции могут быть обращены и расшифровывание осуществляется за число операций, линейно зависящее от размера блока. После нескольких таких "перемешиваний" можно считать, что каждый бит выходного блока (зашифрованного) зависит от каждого бита исходного сообщения.

5.5.2. Шифрование с открытым ключом

В 70-х годах прошлого столетия появился новый тип криптографических алгоритмов — *алгоритмы шифрования с открытым ключом* (или *асимметричное шифрование*). В таких системах все ключи "живут" парами: один используется для шифрования, другой — для дешифрования. Самое интересное — зашифрованное сообщение не может быть дешифровано без знания второго ключа, даже если знать ключ, который использовался для шифрования.

Таким образом, у каждой стороны в криптографической системе с открытым ключом имеется два ключа: *публичный*, для открытого распространения, и *секретный* (private), который знает только его владелец. Для того чтобы A послал зашифрованное сообщение стороне B, необходимо использовать публичный ключ B, потому что лишь B обладает секретным ключом для дешифрования этого сообщения.

Одним из наиболее популярных алгоритмов шифрования с открытым ключом является алгоритм RSA, названный по именам его авторов [*]. Для создания шифра по их методу выбирается большое целое число n — произведение двух простых множителей, p и q. При кодировании используется некоторое целое число, традиционно обозначаемое e (от encode), а при декодировании нужно иметь другое число, обозначаемое d (от decode). Хотя d определяется по e однозначно, но для его определения нужно иметь разложение n на p и q; при очень больших n эта задача оказывается достаточно трудной.

Теперь перейдем к деталям. Пусть $n = p \cdot q$, где p и q — два простых числа. Положим $\varphi = p \cdot q - p - q + 1$. Приведем следующие факты из теории чисел.

Лемма 1. Пусть a — число, взаимно простое с n. Тогда

$$a^{\varphi} = 1 \mod n.$$

Доказательство. Пусть $\Phi(n)$ — множество всех чисел, не превосходящих n и взаимно простых с ним. Заметим, что $|\Phi(n)| = \varphi$. Умножение числа

[*] Ron L. Rivest, Adi Shamir, Leonard Adleman [78].

из этого множества на a по модулю n есть биекция: если $b \cdot a = c \cdot a$, то, поскольку a взаимно просто с n, разность $c - b$ должна делиться на n. Таким образом,

$$\prod_{b \in \Phi(n)} b = \prod_{b \in \Phi(n)} (b \cdot a) = a^{\varphi} \prod_{b \in \Phi(n)} b \mod n,$$

откуда и следует утверждение леммы. □

Лемма 2. Для каждого числа e, взаимно простого с φ, существует единственное $d \in 1 : n$, для которого $e \cdot d = 1 \pmod{\varphi}$.

Доказательство. Вспомним алгоритм Евклида. При установлении того, что общий наибольший делитель e и φ равен 1, легко находятся (см. дальше пример) коэффициенты d и k, для которых $d \cdot e + k \cdot \varphi = 1$. Это верно для любых взаимно простых e и φ. Однозначность определения d получается легко доказательством от противного. □

Пример. Пусть $e = 47$ и $\varphi = 104$. Имеем $\varphi = 2e + 10$. Далее, в соответствии с алгоритмом Евклида $e = 4 \cdot 10 + 7$, затем $10 = 1 \cdot 7 + 3$, $7 = 2 \cdot 3 + 1$. Последнее равенство переписывается так: $1 = 7 - 2 \cdot 3$. Так как $3 = 10 - 1 \cdot 7$, то $1 = 3 \cdot 7 - 2 \cdot 10$. Так как $7 = e - 4 \cdot 10$, то $1 = 3 \cdot e - 14 \cdot 10$. Так как $10 = \varphi - 2 \cdot e$, то $1 = 31 \cdot e - 14\varphi$.

Перейдем к описанию персонального шифра. Для каждого *получателя* выберем достаточно большое число n, которое имеет два простых делителя p и q. Например, $n = 1\,093\,709 = 997 \cdot 1097$, хотя реально используются гораздо более длинные числа, которые разложить на множители очень трудно. По этим сомножителям находится $\varphi = pq - p - q + 1$. В нашем случае $\varphi = 1\,091\,616$.

Получатель (хозяин шифра) выбирает какое-либо число e, взаимно простое с φ, и вычисляет "обратное" для него d. Если выбрать $e = 397$, то $d = 145\,777$. Числа n и e сообщаются отправителям, это открытая часть системы шифрования.

Отправляемый текст перестраивается таким образом, чтобы он состоял из отдельных чисел в диапазоне от 1 до n. Каждое кодируемое число $x \in 1 : n$ возводится в степень e по модулю n. Возведение числа в большую степень сильно упрощается, если используется двоичное разложение показателя степени. Например, чтобы возвести число x в степень $397 = 110\,001\,101_2$, нужно последовательно вычислить x^2, x^4, x^8, x^{16}, x^{32}, x^{64}, x^{128}, x^{256} и перемножить нужные пять (каких?) сомножителей. Результат y передается получателю.

Получатель, имея число d, вычисляет y^d по модулю n. Так как $y^d = (x^e)^d = x^{de}$, то, если $x \in \Phi(n)$, результат возведения в степень равен x *).

*) Случай, когда числа x и n не взаимно просты, очень неприятен, и не столько тем, что обратное преобразование не однозначно, сколько тем, что знание простого делителя раскрывает шифр, делая его дальнейшее использование сомнительным. Приходится утешаться тем, что вероятность попасть на число, кратное простому сомножителю, ничтожно мала.

В отличие от алгоритмов симметричного шифрования, которых много, существует лишь несколько алгоритмов шифрования с открытым ключом; из них мы отметим следующие.

1. "Рюкзак" Меркля–Хеллмана[*]. Алгоритм, очень любопытный с математической точки зрения. По шифруемой битовой строке $x = \{x_i\}$, $i \in 1:n$, составляется сумма $s = \sum_i a_i x_i$, которая и является шифрованным текстом. Числа a_i известны, они и являются открытым ключом. Секретный ключ состоит из взаимно простых чисел W и M, b_i, специальным образом выбранного набора чисел $i \in 1:n$ и перестановки π порядка n. Числа a_i связаны с b_i соотношением $a_i = Wb_{\pi(i)}$. Восстановление набора x по значению s по открытому ключу очень сложно, а по секретному ключу просто, благодаря особому свойству "сверхвозрастания" последовательности b_i: каждое следующее b_i больше суммы всех предыдущих, а M больше суммы всех b_i.

Получающаяся задача называется *задачей о рюкзаке*. Задачу о рюкзаке как экстремальную задачу мы рассматривали на с. 119. Вариант, используемый при шифровании, несколько отличается: целевой функции в нем нет и суммарный вес отбираемых в рюкзак предметов должен иметь в точности заданный вес. При произвольных весах (задаваемых с большим числом знаков) это сложно, а при условии сверхвозрастания просто.

2. RSA. Этот алгоритм нам уже знаком. Он использует ключи с длинами от 512 битов. Запатентован, но срок патента истек в 2000 г., так что можно ожидать большого распространения этого алгоритма.

3. El Gamal. Получил свое название по имени автора Тахира Эль-Гамаля (Taher El Gamal). Использует ключи с длинами от 512 до 10 254 битов. В отличие от RSA не запатентован, но его использование было ограничено в силу различных причин правового порядка[**].

5.5.3. Цифровые подписи, конверты, дайджесты

Описанный подход в криптографических системах с открытым ключом можно "обратить" и получить не менее удобную и полезную систему аутентификации пользователя, называемую *цифровой подписью*. Механизм очень прост: необходимо придумать произвольную фразу, например "Меня зовут Иван Петров", и затем зашифровать ее с использованием *секретного ключа*. Полученное зашифрованное сообщение является цифровой подписью и, как правило, отсылается вместе с исходным сообщением. Каждый может удостовериться, что

[*] Ralph Merkle, Martin Hellman, 1978. Эти авторы считаются основоположниками асимметричной криптографии.

[**] Дело в том, что авторы другого алгоритма шифрования Диффи и Хеллман подали на Эль-Гамаля в суд за нарушение прав патента. Однако срок патента алгоритма Диффи–Хеллмана истек в апреле 1997 г., и конфликт сам собой затих.

получил сообщение от того, чья цифровая подпись прикреплена, простым дешифрованием с использованием публичного ключа.

Пусть передается текст (число) x, который после преобразования превратился в "подпись" $z = x^d$. Получатель этой подписи имеет вторую часть ключа, т.е. e; он выполняет прямое преобразование и, так как $z^e = (x^d)^e = x^{de} = x$, получает исходный текст.

На практике протоколы цифровых подписей безусловно сложнее, и они опираются на еще один тип криптографических алгоритмов — на дайджесты, о которых речь пойдет далее.

Основной недостаток систем шифрования с открытым ключом — низкая скорость. Самая "быстрая" реализация алгоритма RSA в тысячи раз медленнее любого стандартного алгоритма симметричного шифрования. Поэтому для шифрования длинных сообщений, а также в системах реального времени предпочитают использовать симметричное шифрование. На практике же используются оба типа алгоритмов, скомбинированные так, чтобы нейтрализовать их недостатки. Такая схема называется *цифровым конвертом* (Digital Envelope). Ее основная идея состоит в следующем:

1. Сначала генерируется ключ, который будет использован для симметричного шифрования. Такой ключ называется *ключом сессии*, поскольку он создается для каждой сессии пересылки сообщений заново и не может быть использован после нее.
2. Исходное сообщение шифруется с использованием ключа сессии.
3. Ключ сессии шифруется с помощью публичного ключа получателя, полученное сообщение и называется *цифровым конвертом*.
4. Зашифрованное сообщение и цифровой конверт посылаются по нужному адресу.

Получатель сначала дешифрует цифровой конверт с помощью своего секретного ключа. Затем с помощью ключа сессии он дешифрует само сообщение.

С помощью различных криптографических алгоритмов можно пересылать секретную информацию по открытым каналам связи, но как можно быть уверенным, что сообщение не изменится в процессе передачи? Как можно понять, было ли сообщение модифицировано в процессе пересылки, поскольку само зашифрованное сообщение "нечитаемо"? На все эти вопросы помогут ответить так называемые *цифровые дайджесты*.

Алгоритмы-дайджесты в некотором роде похожи на алгоритмы шифрования тем, что тоже получают на входе обыкновенное "читаемое" сообщение и на выходе дают некоторый "нечитаемый" набор байтов: короткие (фиксированной длины) дайджест-сообщения (также называемые *хешами* (hash)) обычно намного короче исходного сообщения. Их дешифрование не предусматривается.

Дайджесты выступают в роли своеобразных "контрольных сумм" сообщения, поскольку если в исходном сообщении изменить хотя бы один бит, то дайджест измененной информации будет отличаться от исходного дайджеста.

Они используются при передаче информации в качестве индикатора правильности принятой информации. Например, в стандарте цифровой подписи DSS (Digital Signature Standard) используется следующий механизм:

1. Получают дайджест исходного сообщения.
2. Зашифровывают дайджест с помощью секретного ключа ("подписывают" дайджест).
3. Посылают "подписанный" дайджест и исходное сообщение получателю.

Таким образом достигаются две цели. Во-первых, получатель может удостовериться, что полученное сообщение не претерпело изменений в процессе передачи (для этого нужно дешифровать "подписанный" дайджест с помощью публичного ключа отправителя, получить дайджест полученного сообщения и сравнить эти два дайджеста. Совпадение означает, что исходное сообщение не изменено и получено именно от того лица, чей публичный ключ использовался для дешифрования). Во-вторых, подтверждается подлинность отправителя информации *).

5.5.4. Немного о длине ключей и правовых аспектах

По ходу этого изложения мы часто упоминали, что важна длина ключа, используемого при шифровании или дешифровании. Действительно, очевиден тот факт, что если в криптографической системе будет "легко" угадать ключ (как в симметричном шифровании, так и в системах с открытым ключом), то она не может быть использована для передачи действительно конфиденциальной информации.

Приведем немного статистики на примере алгоритмов DES и AES. Как раз здесь будет видно, почему был осуществлен переход от "устаревшего" алгоритма к новому. Существуют AES-ключи трех длин: 128 битов, 192 бита и 256 битов. Это означает, что при полном переборе нужно будет пробовать $3.4 \cdot 10^{38}$ 128-битовых ключей, $6.2 \cdot 10^{57}$ 192-битовых и $1.1 \cdot 10^{77}$ 256-битовых. Для сравнения: DES-ключи имеют длину 56 битов, а значит, возможных ключей существует в 10^{21} раз меньше, чем 128-битовых AES-ключей.

Шифры хранились и хранятся как высочайшие государственные секреты **), поэтому не все уровни шифров доходят до всеобщего обсуждения.

*) Существуют различные более сложные методы использования дайджестов, например в протоколе SSL (Secure Socket Layer) ключевым моментом является процедура аутентификации сообщений MAC (Message Authenticity Check), которая вычисляет по данному сообщению M удостоверяющий код $MAC(M)$.

**) Во время Второй мировой войны англичане располагали способом дешифровки немецкой криптосистемы и оперативно читали тексты радиоперехватов. Получив и прочитав заранее сообщение о налете на город Ковентри, они не сочли возможным предупредить его жителей, считая более важным сохранение своей тайны. Интересно, что активное участие в работе английской группы принимал известный английский математик Алан Тьюринг (Alan Turing, 1912–1954).

и некоторые важные методы остаются "за семью печатями". Например, запрещен экспорт за пределы США и Канады систем, использующих при шифровании ключи длиннее 56 битов. Такое ограничение вызвано желанием государства "держать под контролем" потоки информации, выходящие за пределы страны, а, как мы уже говорили, дешифрование сообщений может занять достаточно большое время даже у сверхмощных компьютеров при использовании длинных ключей.

Другим примером является соглашение между Software Publishers Association (SPA) и правительством США, которое дает алгоритмам RC2 и RC4 специальный статус, разрешающий экспортировать шифры с длиной ключа до 40 битов. При этом 56-битовые ключи разрешено использовать заграничным отделениям американских компаний.

Глава 6

Информационный поиск и организация информации

6.1. Зачем здесь этим заниматься?

Сейчас компьютеры используются (раньше я писал — *все чаще*, теперь нужно говорить — *в основном*) для хранения больших количеств информации, и среди задач, в которых эта информация применяется, одна из важнейших — поиск нужной информации. Действия по поиску информации входят в качестве важной и трудоемкой составляющей во многие алгоритмы. Поэтому нам важно знать наиболее эффективные конструкции, обеспечивающие те или иные возможности быстрого поиска. Не зная этих конструкций, невозможно правильно оценить трудоемкость многих алгоритмов. Разумеется, у нас нет возможности рассмотреть здесь задачи поиска во всей полноте, мы ограничимся некоторыми начальными представлениями.

Итак, пусть имеется некоторая *база данных* — совокупность равноправных *записей*, в каждой из которых есть *информационная* и *идентификационная* части. На информационную часть не накладывается никаких ограничений. Идентификационная часть записи уникальна, ее значение отлично от всех остальных, обычно ее называют *ключом записи*. Первоначально можно предположить, что ключ — это просто целое число.

Нас интересует, как организовать в базе данных поиск записи с нужным значением ключа. Здесь слово *поиск* подразумевает и проверку наличия записи с таким ключом, и установление места этой записи в памяти.

Трудность в том, что записей в базе очень много, а возможных значений ключа и того больше, поэтому быстрый поиск требует специальной организации данных, которая должна обеспечивать возможность оперативного добавления новых записей с новыми ключами, причем так, чтобы даже при многократном добавлении новых записей поиск оставался достаточно быстрым.

Зачем этими вопросами заниматься в дискретном анализе? Дело в том, что одной из важных задач современного дискретного анализа является оценка трудоемкости различных вычислительных схем. А эта трудоемкость существенно зависит от того, как организованы данные и работа с ними.

6.2. Простейшие механизмы — массивы, файлы и цепные списки

Хотя вопросы этого параграфа принадлежат, конечно, проблематике программирования и представления данных в компьютерах, необходимо упомянуть, хотя бы вкратце, об основных конструкциях, используемых при работе с массовой информацией.

Наиболее простой из таких конструкций является "массив". При обучении программированию он появляется очень рано и знаком всем. Простейший, одномерный, массив состоит из однотипных элементов, которые перенумерованы подряд, так что каждый элемент информации (ячейка массива) имеет свой числовой *индекс*. Все ячейки массива имеют один и тот же размер, благодаря чему доступ к ячейке с нужным индексом j осуществляется очень просто: его адрес A_j находится по формуле:

$$A_j = A_0 + \Delta A \cdot j.$$

Здесь A_0 — *база массива*, адрес элемента с нулевым индексом. Базу удобно определять именно таким образом, даже если нулевого элемента в массиве нет. ΔA — *шаг массива*, расстояние между последовательными записями. Во многих программных системах этот шаг не равен длине записи из-за так называемого *выравнивания* — поля́ некоторых типов данных должны располагаться начиная с адреса, кратного 2, 4 или 8, и при соединении в записи полей различной "кратности" остается неиспользуемое место.

Массив как средство массового хранения однотипной информации обладает целым рядом существенных преимуществ: в нем очень проста операция обращения к элементу, достаточно вычислить адрес по

приведенной выше формуле — и все (хотя при использовании виртуальных массивов, имитирующих сплошную память, должна еще быть выполнена процедура, которая по адресу в воображаемой сплошной памяти находит адрес в физической памяти). Можно сказать, что когда удается использовать массивы, особенно одномерные — с одним индексом, это нужно делать.

Но у массивов есть и недостатки. Нужно, чтобы ключи записей (индексы в массиве) шли подряд. Если используются только некоторые из возможных значений индекса, то получается *редкое заполнение массива*, о котором речь уже шла в предыдущей главе. Правда, можно записывать элементы в массив подряд, считая ключ элемента частью информации. В таком массиве процедура обращения к элементу должна быть организована как-то иначе. Такое обращение выполняется достаточно просто, если записи расположены в массиве монотонно — по упорядочению или по убыванию. Мы вернемся к этому вопросу позднее.

Другая исходная конструкция даже древнее, чем массив. Это *последовательный файл*. Файлы появились еще при записи данных на перфокартах. Перфокарты собирались в стопку — *колоду*, и эта колода обрабатывалась машиной последовательно, карта за картой. При выводе перфокарт из компьютера карты перфорировались также последовательно, получался выходной файл. В качестве файла потом стали рассматривать и печатное устройство, которое последовательно печатало одну строчку текста за другой. Затем к этим *устройствам последовательного доступа* добавились магнитные ленты.

Во всех случаях основными операциями над последовательными файлами были установка на начало файла и чтение или запись следующего элемента. Преимущества этой конструкции — в ее простоте и отсутствии необходимости задавать число элементов в файле. Недостатки, по-видимому, понятны: любое действие с таким файлом требует его полного просмотра.

Наконец, мы должны упомянуть *цепной список* как основную конструкцию, реализующую в оперативной памяти компьютера свободно собираемые и перестраиваемые последовательности из однотипных элементов. Цепной список составляется из элементов, каждый из которых состоит из двух частей — *информационной*, в которой хранится вся относящаяся к данному элементу "полезная информация", и *заголовочной*, в которой указывается адрес следующего элемента.

Главное преимущество цепных списков — в возможности простого добавления и удаления элемента из списка, когда он уже найден

Недостатки цепного списка — трудность поиска в нем нужного элемента при большой длине списка. Однако конструкции, основанные на цепных списках, распространены очень широко.

6.3. Простейшее действие организации — сортировка

Для дискретного анализа характерно, что самые простые, казалось бы, ничем не примечательные задачи могут стать предметом серьезного многолетнего научного исследования. Здесь мы рассмотрим одну из таких простых задач, которая часто встречается в приложениях и до сих пор осталась интересной.

Задано конечное множество A, состоящее из n элементов, а на нем задано отношение линейного порядка P. Требуется перенумеровать элементы A числами от 1 до n таким образом, чтобы из неравенства $i < j$ следовало $(a_i, a_j) \in P$. Выполнение этой задачи называется *сортировкой множества.*

Задача становится интересной, когда количество упорядочиваемых элементов становится большим. Потребность в сортировке больших объемов данных ощущалась очень давно. В знаменитом некогда комплекте счетно-аналитических машин Холлерита *), работавших с перфокартами, была специальная сортирующая машина, которая сортировала последовательные файлы — колоды перфокарт. Машина раскладывала карты из одной колоды, скажем для простоты, по 10 карманам в зависимости от цифры, пробитой в данном столбце перфокарты.

Вот интересное упражнение для читателя: на каждой перфокарте в колонках с 76 по 80 пробит ее пятизначный номер. Как, используя сортировочную машину, разложить колоду перфокарт по возрастанию номеров?

Использованный в этой машине принцип сортировки оказывается выгодным и для современных компьютеров в некоторых специфических ситуациях. Мы вернемся к нему позднее, после рассмотрения более традиционных методов.

Сами способы сравнения записей, используемые в сортировке, могут быть очень разнообразными, но в большинстве случаев они

*) Hermann Hollerith (1860–1929), американский изобретатель. Созданная им фирма была в 1924 г. переименована в IBM — International Business Machines и существует, как вы знаете, до сих пор.

исходят из двух базовых элементарных упорядочений: упорядочение чисел по значению и упорядочение букв в алфавите *).

Упражнение 6.1. В одном ремесленно выполненном телефонном справочнике числа, входящие в названия, воспринимаются как части строк; например, почтовые отделения идут в таком порядке: 198, 2, 20, 202, 21 и т. п. Разработайте процедуру упорядочения строк, в которой входящие в строку числа упорядочивались бы по значению: 2, 20, 21, 198, 202.

Мы уже говорили на с. 93 о лексикографическом упорядочении строк, порождаемом упорядочением букв алфавита. Некоторым развитием этого способа упорядочения является упорядочение записей по нескольким ключам.

Во многих системах информация составлена из наборов однотипных структур — *записей*, причем каждая запись состоит из нескольких полей. Обычно одно или несколько полей образуют *ключ записи* — уникальное сочетание значений, однозначно идентифицирующее запись. При обсуждении вопросов упорядочения слово "ключ" используется и в другом смысле: ключ сортировки — это поле или совокупность полей с линейным порядком, определенным на множестве их возможных значений.

Итак, считается, что задан набор однотипных записей, которые мы хотим упорядочить. Пусть в записи R значение ключа k обозначается через $R.k$. Если записи упорядочиваются по s ключам $k_1, \ldots, k_s$, то запись A предшествует записи B в том и только том случае, если найдется такой индекс $r \leq s$, что $A.k_i = B.k_i$ для $i < r$ и $A.k_r$ предшествует $B.k_r$ в смысле упорядочения ключа k_r. В стандартных средствах сортировки обычно предусматривается сортировка по сложному ключу.

При рассмотрении различных методов сортировки для нас несущественно, каким правилом задается сравнение двух элементов множества, — существенно лишь, насколько трудоемко это сравнение. Принципиальные схемы сортировки можно рассматривать и на обычных наборах чисел; именно с этого случая мы и начнем.

Итак, пусть задан набор чисел $a_1, a_2, \ldots, a_n$. Требуется расположить их в возрастающем (неубывающем) порядке, т. е. найти перестановку $i_1, i_2, \ldots, i_n$, чтобы

$$a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}.$$

Методов сортировки и их вариантов сейчас накопилось очень много, мы ограничимся лишь несколькими принципиально различными. Подробности см. в фундаментальной книге Д. Кнута [25].

*) Еще одно базовое элементарное упорядочение — упорядочение вершин дерева в порядке их появления при просмотре дерева. Такое упорядочение встречается, например, в правилах наследования в монархических домах. Интересны также способы упорядочения иероглифов в китайских словарях.

Методы типа вставки. Сразу приходит в голову такой метод: наращивать отсортированную часть последовательности. Именно, сначала взять последовательность из одного первого элемента, она уже "отсортирована". Затем брать очередной элемент и, сравнивая его с предыдущими, переставлять вперед до тех пор, пока он не займет свое место. Вот так (это вставка только одного элемента):

11, 16, 18, 29, 56, **17**
11, 16, 18, 29, **17**, 56
11, 16, 18, **17**, 29, 56
11, 16, **17**, 18, 29, 56

Такой метод имеет трудоемкость порядка n^2 (эта оценка естественна, и легко построить пример, в котором она реализуется: возьмите последовательность, упорядоченную по убыванию, и попробуйте ее упорядочить по возрастанию).

Есть еще много разновидностей методов сортировки с такой же трудоемкостью. Мы их здесь рассматривать не будем.

Метод фон Неймана. Первым предложил алгоритм трудоемкости $O(n\log_2 n)$ американский математик Дж. фон Нейман *). Предложенная им сортировка основывалась на уже встречавшейся нам операции *слияния* двух упорядоченных массивов.

Первоначально сортируемый массив представляется в виде n "отсортированных" массивов, каждый длины 1. Сольем их попарно, получив примерно $n/2$ массивов длины 2. Потом опять. За $\lceil\log_2 n\rceil$ шагов (напомним: так обозначено округление числа до целого в бо́льшую сторону) получим один массив длины n. Так как трудоемкость каждого шага равна n, суммарная трудоемкость сортировки по этому методу равна $n\log_2 n$.

Пример. Отсортируем этим методом небольшой массив. При 11 элементах он отсортируется за 4 итерации

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

*) John von Neumann (1903–1957), выдающийся математик, родившийся в Венгрии, работал в Германии, Австрии и США. Его имя встретится нам и в дальнейшем. Фон Нейман был также одним из главных творцов современной вычислительной техники. Кстати, с его именем связано и возникновение статистического моделирования, и создание *теории игр* — математической теории, изучающей конфликты интересов, и многие современные вопросы математики.

Этот метод очень прост и удобен в реализации, если не считать того, что операция слияния требует дополнительной памяти — еще одной копии массива для хранения результата слияния. В качестве другого недостатка этого метода можно назвать его "универсальную трудоемкость" — для почти отсортированных массивов тратится столько же времени, как и в самом плохом случае. Но есть интересные варианты метода фон Неймана, избавленные от универсальной трудоемкости. В них сортируемый массив рассматривается как последовательность упорядоченных частей; их длина не фиксирована, а определяется только сохранением порядка.

В частности, на основе метода фон Неймана удобно строить алгоритмы для сортировки файлов и цепных списков. Исходный файл автоматически делится на монотонные участки, а на каждом шаге пишется несколько новых файлов (например, два), в каждом из которых этих монотонных участков становится вдвое меньше. В случае цепных списков все становится еще проще: ничего физически переписывать не нужно, просто изменяются связи элементов.

Метод Шелла. Замечательная модификация метода вставки была предложена Д. Шеллом *): проводить сортировку за несколько итераций, во время каждой из которых сортируются по отдельности (например, методом вставки) отдельные части массива. На каждой итерации задается некоторый *шаг сортировки*, который убывает от итерации к итерации, и массив разбит на "решетки" — наборы элементов, отстоящих друг от друга на целое число шагов. Таким образом, при шаге s массив разбивается на s решеток. Процесс заканчивается после выполнения итерации с $s = 1$, поэтому метод Шелла включает в себя обычный метод вставки как один из шагов. Тем не менее, он гораздо эффективнее обычного метода вставки, так как большая часть перестановок делается на начальных итерациях "крупными шагами".

Пример. Пусть требуется сортировать массив

$$a[1:14] = (12, 6, 27, 2, 5, 29, 11, 14, 7, 8, 16, 13, 28, 17).$$

Выбираем первоначально шаг 9 и упорядочиваем решетки 12–8, 6–16, 27–13, 2–28, 5–17. Получаем

$$(8, 6, 13, 2, 5, 29, 11, 14, 7, 12, 16, 27, 28, 17).$$

Уменьшаем шаг примерно вдвое, получаем шаг 5. Упорядочиваем решетки 8–29–16, 6–11–27, 13–14–28, 2–7–17, 5–12. Фактически будет переставлен только один элемент:

$$(8, 6, 13, 2, 5, 16, 11, 14, 7, 12, 29, 27, 28, 17).$$

*) Donald L. Shell, 1959.

Следующий шаг 3. Имеем

$$(2, 5, 7, 8, 6, 13, 11, 14, 16, 12, 17, 27, 28, 29).$$

Обратите внимание: массив "почти" упорядочен. Наконец после упорядочения с шагами 2 и 1 получаем полностью отсортированный массив

$$(2, 5, 6, 8, 7, 12, 11, 13, 16, 14, 17, 27, 28, 29).$$

Конечно, сортировка идет не по решеткам: каждый элемент продвигается назад с шагом s, пока не займет нужное место.

Убывающие шаги в методе можно выбирать произвольно, но они не должны быть кратны; при кратных шагах, например по степеням 2, происходит просто слияние упорядоченных решеток (как в методе Неймана, но без гарантии логарифмической трудоемкости). Выбранная в примере последовательность шагов вида $2^k + 1$ обеспечивает взаимную простоту последовательных шагов (докажите это сами!) и удобна для пересчета. Специалисты рекомендуют увеличить отношение между последовательными шагами до 3.

Метод Quicksort. Английский математик Хоар*) предложил эффективный метод сортировки, основанный на, казалось бы, наивной идее. Основная операция его метода — взять один из элементов массива и поставить на его законное место, при этом так, чтобы элементы, предшествующие ему в упорядочении, размещались до него, а следующие за ним — после. После этого получится два массива, *до* и *после*, каждый из которых можно упорядочивать тем же способом. Когда массив становится очень маленьким (2–3–4 элемента), его проще упорядочить методом вставки или чем-нибудь аналогичным. Оказывается, этот метод не так уж наивен, он имеет прекрасную реализацию и совершенно справедливо называется Quicksort.

Нам важно описать только элементарный шаг этой сортировки. На каждом шаге сортируемый массив просматривается в двух направлениях — от начала и от конца. С одной стороны стоит тот самый *разделяющий элемент*, место которого мы ищем, с другой стороны — *проверяемый элемент*, который сравнивается с разделяющим. Первоначально разделяющий — первый элемент массива, а проверяемый — последний. На каждом шаге, если эти два элемента упорядочены неправильно, они меняются местами и вместе с этим изменяются роли сторон. Таким образом, в любом случае в качестве проверяемого

*) Sir Charles Antony Richard Hoare, 1962.

элемента берется следующий по направлению к разделяющему элементу. Когда проверяемый и разделяющий элементы совпадут, итерация сортировки закончится.

Пример. Пусть требуется сортировать массив

$$a[1:14] = (12, 6, 27, 2, 5, 29, 11, 14, 7, 8, 16, 13, 28, 17).$$

Первоначально элемент 12 выберем в качестве разделяющего, а на противоположном конце массива стоит проверяемый элемент 17. Так как расположение 12—17 правильное, переходим к проверке следующего элемента, 28, а затем 13, 16, наконец, 8. Расположение 12—8 неправильное. Переставим эти элементы местами:

$$(8, 6, 27, 2, 5, 29, 11, 14, 7, 12, 16, 13, 28, 17).$$

Теперь они расположены правильно, но роли концов массива переменились. Проверяем расположение 6—12, затем 27—12, это нарушение заставляет нас снова сделать перестановку и сменить роли концов массива:

$$(8, 6, 12, 2, 5, 29, 11, 14, 7, 27, 16, 13, 28, 17).$$

Обратите внимание на то, что зона просмотра все время сужается. Продолжая, получаем

$$(8, 6, 7, 2, 5, 11, \mathit{12}, 14, 29, 27, 16, 13, 28, 17).$$

Элемент 12 стоит на своем правильном месте, и нужно сортировать отдельно два других массива, *до* и *после* этого элемента.

Упражнение 6.2. Существует вариант метода Quicksort, в котором последовательно выполняются такие действия: сначала находятся неправильно стоящие элементы в левой и в правой частях массива, а затем они меняются местами. Разработайте детали этого метода. Что в нем нужно делать при завершении итерации?

Сейчас популярен вариант Quicksort, в котором граничный элемент выбирается как средний по значению из трех элементов, например, из крайних и среднего по положению. Выбранный элемент ставится в начало массива, затем наименьший из трех образует зону меньших значений ZL, а наибольший — зону больших значений ZG. Остальные элементы последовательно просматриваются, и если текущий элемент больше барьерного, он просто присоединяется к зоне ZG — она расширяется. Если же текущий элемент меньше барьерного, то он меняется местами с первым элементом зоны ZG, которая затем сдвигается на одну позицию вправо, расширяя при этом зону ZL.

Пример. Посмотрим, как работает этот алгоритм.

Три выбранных элемента 12, 11, 17 уже расположены как надо, 11 и 17 встали на свои места, поменявшись соответственно с 6 и 27. Дальше зоны *ZL* и *ZG* мы выделяем скобками

12, (11), (17), 2, 5, 29, 6, 14, 7, 8, 16, 13, 28, 27

Элемент 2 меняется местами с 17, а затем зона *ZG* сдвигается (будьте осторожны: неправильно говорить, что элемент 2 просто приписывается к зоне *ZL*).

12, (11, 2), (17), 5, 29, 6, 14, 7, 8, 16, 13, 28, 27

Затем то же произойдет с элементом 5.

12, (11, 2, 5), (17), 29, 6, 14, 7, 8, 16, 13, 28, 27

Элемент 29 приписывается к зоне *ZG*

12, (11, 2, 5), (17, 29), 6, 14, 7, 8, 16, 13, 28, 27

Элемент 6 попадает в зону *ZL*; обратите внимание, как изменился порядок элементов в *ZG*.

12, (11, 2, 5, 6), (29, 17), 14, 7, 8, 16, 13, 28, 27

Элемент 14 приписывается к зоне *ZG*

12, (11, 2, 5, 6), (29, 17, 14), 7, 8, 16, 13, 28, 27

Элементы 7 и 8 попадают в зону *ZL*; каждый из них изменяет порядок элементов в *ZG*.

12, (11, 2, 5, 6, 7), (17, 14, 29), 8, 16, 13, 28, 27

12, (11, 2, 5, 6, 7, 8), (14, 29, 17), 16, 13, 28, 27

Остаток присоединяется к зоне *ZG*

12, (11, 2, 5, 6, 7, 8), (14, 29, 17, 16, 13, 28, 27)

Разделяющий элемент ставится на свое место: он меняется с последним элементом зоны *ZL*, которая после этого сдвигается на один элемент влево.

(8, 11, 2, 5, 6, 7), 12, (14, 29, 17, 16, 13, 28, 27)

Преимущество этого метода в том, что на каждом шаге делается не больше одной элементарной перестановки элементов массива. Расширение и сдвиг зон легко выполняются пересчетом границ этих зон.

Так разделяются все диапазоны, размер которых больше 3. После того как дробление интервалов завершено, один полный прогон сортировки вставкой заканчивает упорядочение.

Причиной резкого снижения эффективности этого метода сортировки может стать наличие большого числа равных элементов. Это затруднение легко преодолеть, если в операции сравнения элементов в случае их равенства поочередно вырабатывать значения "больше" и "меньше". Тогда равные элементы будут распределяться по интервалам примерно поровну.

Heapsort. Этот метод разработал Дж. Уильямс *). В основе метода — удобная нумерация вершин двоичного дерева и связь дерева с массивом. Слово heap в названии сортировки переводится *куча* — так принято называть информационные деревья со специальным упорядочением находящихся в вершинах элементов.

Рассмотрим полное двоичное дерево с ветвями высоты k. Такое дерево имеет $2^{k+1} - 1$ вершин (так что зависимость высоты дерева от числа вершин, грубо говоря, логарифмическая). Действительно, корневая вершина образует как бы нулевой слой этого дерева, а далее в каждом горизонтальном слое число вершин вдвое больше, чем в предыдущем, и слой k содержит 2^k вершин. Если перенумеровать вершины сверху вниз и слева направо, как показано на рис. 6.1, то легко убедиться (докажите сами!), что вершина с номером s соединяется с вершинами $2s$ и $2s + 1$ следующего слоя (*дети* этой вершины).

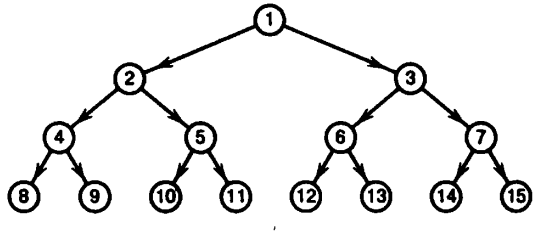


Рис. 6.1. Нумерация вершин двоичного дерева

Иерархическая сортировка основана на этом способе нумерации. Каждому индексу $i \in 1 : n$ сопоставляется вершина, и элементы массива $a[1 : n]$ могут рассматриваться как вершины двоичного дерева.

*) John William Joseph Williams. Рассматриваемая форма алгоритма согласно Д. Кнуту [25] содержит модификацию Р. Флойда (R. Floyd).

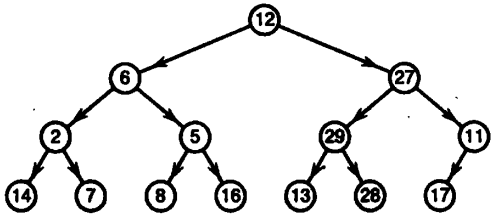


Рис. 6.2. Массив как дерево

На рис. 6.2 показано дерево, соответствующее массиву

$$a[1:14] = (12, 6, 27, 2, 5, 29, 11, 14, 7, 8, 16, 13, 28, 17).$$

В терминах этого дерева и происходит сортировка массива; простая арифметическая связь номеров соединенных вершин позволяет перевести эти термины в индексы.

Сортировка состоит из двух фаз. Первая фаза обеспечивает *иерархическую упорядоченность массива*: перестановкой элементов мы добиваемся того, чтобы значение в каждой вершине было *хуже*, чем у детей. Для этого используется операция "утапливания" элемента: текущий элемент сравнивается с худшим из элементов-детей и, если тот хуже, они меняются местами; на новом месте операция для текущего элемента повторяется. Так утапливается каждый элемент начиная от номера $\lfloor \frac{n}{2} \rfloor$ (почему не от номера n и не от $\lceil \frac{n}{2} \rceil$?) и кончая 1. В нашем примере сначала утопится элемент 11, поменявшись с 17, затем 5, поменявшись с 16, а не с 8, затем 2, поменявшись с 14. На рис. 6.3 слева показан путь, который пройдет элемент 27, а справа — состояние дерева по завершении первой фазы.

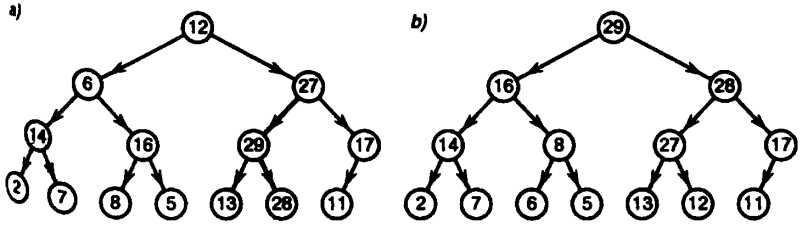


Рис. 6.3. Состояние массива:
a — после первой итерации; *b* — после первой фазы

По завершении первой фазы массив выглядит следующим образом:

$$a[1:14] = (29, 16, 28, 14, 8, 27, 17, 2, 7, 6, 5, 13, 12, 11).$$

Вторая фаза сортировки состоит из последовательного выполнения следующих действий: взять самый худший элемент (он сейчас на первом месте, а должен быть на последнем), поменять его местами с последним, сократить на единицу число вершин в дереве, "отцепив" последний элемент, утопить верхний элемент дерева, чтобы восстановить иерархическую упорядоченность. Так, после установки на место последнего элемента 29 на его место ставится элемент 11, и после утапливания этого элемента получаем

$$a[1:14] = (28, 16, 27, 14, 8, 13, 17, 2, 7, 6, 5, 11, 12; \mathit{29}).$$

Отсортированная часть массива, уже не включаемая в дерево, набрана курсивом и отделена точкой с запятой, чтобы было виднее. После второго шага получаем

$$a[1:14] = (27, 16, 17, 14, 8, 13, 12, 2, 7, 6, 5, 11; \mathit{28}, \mathit{29}).$$

Упражнение 6.3. Проделайте остальные шаги второй фазы, используя для наглядности представление массива в виде дерева.

Упражнение 6.4. Докажите, что каждая из фаз иерархической сортировки требует порядка $O(n \log_2 n)$ действий.

Сравнение этих методов показывает, что каждый из них имеет свою область применения. Метод вставки очень прост, и при малых размерах сортируемого массива, а также при ожидаемых небольших нарушениях порядка можно не беспокоиться о более эффективном методе. К тому же метод *устойчив* — сохраняет существующий в массиве порядок при упорядочении элементов с равными ключами сортировки. Из перечисленных методов кроме вставки устойчив только метод Неймана.

Наиболее практичны сортировки Шелла и Quicksort. Иерархическая сортировка при гарантированном порядке числа операций $O(n \log_2 n)$ оказывается гораздо более трудоемкой.

При сравнении методов основным критерием должно быть количество выполняемых операций. Главных операций две: сравнение двух записей и их перестановка. Единая оценка здесь невозможна в принципе, так как в разных ситуациях трудоемкости этих операций сильно различаются между собой.

При сортировке набора, состоящего из сложных информационных объектов (больших, или имеющих неравную длину, или вообще не представляющих собой одного блока памяти, переносимого с места на место, так что мы не рискуем даже использовать слово "запись"), целесообразно сортировать и переставлять не сами объекты, а указатели на них. Из таких указателей формируется обычный одномерный массив — *индекс*, и он сортируется обычным образом, с той только особенностью, что используемое в сортировке сравнение двух указателей выполняется сравнением объектов, на которые они указывают. Использование индексов позволяет работать с несколькими упорядочениями одних и тех же данных одновременно.

При сортировке данных, которые размещены во внешней памяти, наиболее существенно количество обращений к внешней памяти для чтения или записи информации — каждое такое обращение расходует значительно больше времени, чем упорядочение фрагмента данных, находящегося в оперативной памяти.

Специальные методы сортировки. В некоторых случаях удается выполнить сортировку гораздо экономнее. Так, если сортируемые значения являются целыми числами из ограниченного диапазона, их можно упорядочить за линейное время, используя идеи "сортировочной машины" Холлерита. Откроем теперь секрет этой сортировки: нужно начинать с младшего разряда и, разделив данные по его значению, соединить все колоды в одну, не перемешивая. Затем повторить то же со следующим разрядом и т. д. Сортировка k-значных десятичных чисел потребует k итераций.

Пример. Если мы сортируем числа от 0 до 999, нам потребуется три прогона, как это видно из рис. 6.4.

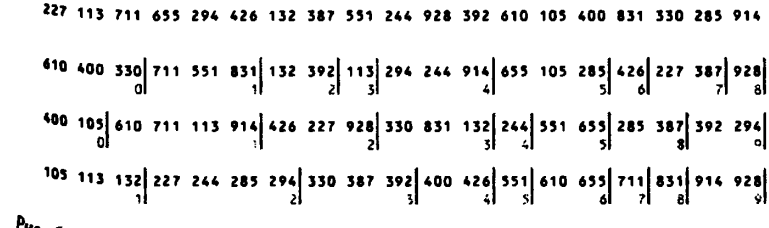


Рис. 6.4. Сортировка чисел от 0 до 999 за три прогона. В первой строчке — исходный массив, во второй — результат раскладывания по карманам в зависимости от последней цифры числа, в третьей — результат раскладывания второго массива по второй цифре и т. д.

Такая сортировка (ее часто называют сортировкой по остаткам — radix sorting) легко программируется, причем "карманы" могут быть организованы с помощью цепных списков; их число можно существенно увеличить, что уменьшает число прогонов. В некоторых случаях удается обойтись одним прогоном.

Упражнение 6.5. Как за время, линейное по k, упорядочить набор из k элементов множества D_n, введенного в упражнении 2.23 на с. 53?

Теперь настало время сказать об эффективном методе построения суффиксного массива. Мы говорили об этом массиве при рассмотрении метода сжатия Барроуза–Уилера (см. п. 5.3.3). Несколько замечательных методов было разработано в последние годы.

Построение суффиксного массива — это ведь тоже сортировка, мы сортируем в лексикографическом порядке суффиксы заданной строки. Рассмотрим предлагаемый метод на том же примере, с которым работали, излагая метод Барроуза–Уилера.

```
          1         2         3         4         5
12345678901234567890123456789012345678901234567890123456789012345678
карл_у_клары_украл_кораллы_а_клара_у_карла_украла_кларнет$
```

В строке 58 символов, считая флажковый, и столько же суффиксов. Каждый суффикс определяется номером начальной позиции. Один суффикс имеет длину 1, и он предшествует всем остальным по особому свойству флажкового символа. Остальные суффиксы можно предварительно отсортировать в зависимости от их двусимвольного начала. Для этого подсчитаем, сколько каких двусимвольных начал имеется в строке

```
$  1  1    _а 1  2    _к 5  3    _у 4  8    а_ 4  12
ал 3  16   ар 5  19   ет 1  24   ка 2  25   кл 3  27
ко 1  30   кр 2  31   л_ 2  33   ла 5  35   лл 1  40
лы 1  41   не 1  42   ор 1  43   ра 4  44   рл 2  48
ри 1  50   ры 1  51   т$ 1  52   у_ 2  53   ук 2  55
ы_ 2  57
```

Теперь ясно, что, например, суффиксы, начинающиеся с _к, располагаются при упорядочении в позициях от 3 до 8, — это определяется их сравнением с другими суффиксами по двусимвольным началам. А для сравнения этих суффиксов между собой нужно привлечь их продолжения. Значит, суффиксы 7, 19, 29, 37 и 50 должны быть расположены так же, как суффиксы 9, 21, 31, 39 и 52 (соответственно, ла..., ор..., ла..., ар..., ла...). Позиция суффикса 39 определилась точно — это 3, начало группы. Суффикс 21 занимает позицию 7 — конец

группы. Три остальных находятся в позициях от 4 до 6. Их взамное расположение подлежит дальнейшему уточнению.

Так можно уточнить упорядочение во всех группах, размер которых больше 1. После уточнения группы будут состоять из суффиксов, в начале которых совпадают по 4 буквы.

```
$      1  1   _а     1  2   _кар   1  3   _кла   3  4   _кор   1  7
_у_к   2  8   _укр   2 10   а_к_   1 12   а_кл   1 13   а_у_   1 14
а_ук   1 15   ал_к   1 16   ала_   1 17   алы    1 18   ара_   1 19
арл_   1 20   арла   1 21   арне   1 22   ары_   1 23   ет     1 24
карл   2 25   клар   3 27   ко     1 30   крал   2 31   л_у_   1 33
л_ко   1 34   ла_к   1 35   ла_у   1 36   лара   1 37   лазн   1 38
лары   1 39   лл     1 40   лы     1 41   не     1 42   ор     1 43
ра_у   1 44   рал_   1 45   рала   1 46   ралл   1 47   рл_у   1 48
рла_   1 49   рн     1 50   ры     1 51   т$     1 52   у_ка   1 53
у_кл   1 54   укра   2 55   ы_а_   1 57   ы_ук   1 58
```

Осталось уточнить совсем немного, причем здесь уже в каждой группе совпадает по четыре начальных символа. Ограничимся уточнением тех групп, в которых больше одного суффикса:

```
_клара_у   1  4   _кларнет   1  5   _клары_у   1  6   _у_карла   1  8
_у_клары   1  9   _украл_к   1 10   _украла_   1 11   карл_у_к   1 25
карла_ук   1 26   клара_у_   1 27   кларнет$   1 28   клары_ук   1 29
крал_кор   1 31   крала_кл   1 32   украл_ко   1 55   украла_к   1 56
```

У этого метода есть разные варианты реализации, здесь выбран тот, который легче излагать и понимать. Эксперименты показали поразительно высокую скорость сортировки суффиксов огромных массивов.

6.4. Простейшее ускорение поиска — дихотомия

Упорядоченные массивы хороши тем, что в них искать информацию проще. Самый простой способ, с которого легко начать, — деление поисковой области пополам, или *дихотомия*. В случае, когда ключи — это целые числа, в качестве области значений ключа принимается отрезок; обозначим его $a : b$. Чтобы разделить эту область на две, достаточно взять промежуточное значение d и выбирать один из двух отрезков, например $a : (d - 1)$ и $d : b$. С этим разделением связывается один стандартный шаг поиска: ключ искомой записи k сравнивается с граничным значением d и в зависимости от результата сравнения один из двух отрезков выбирается для дальнейшего поиска. Дальше то же самое происходит до тех пор, пока область поиска не станет пустой

(записи с данным ключом нет) или не сократится до обозримого размера (одна запись или небольшой список).

Систему дроблений поисковой области можно изобразить графически с помощью *двоичного дерева*, в котором каждая из промежуточных областей поиска будет изображаться вершиной, подчиненной предыдущей области. Исходная область является корнем этого дерева, т.е. самой верхней вершиной, никому не подчиненной. Каждый поиск проходит *путь* в этом дереве от корня до некоторой *терминальной* (т.е. оконечной) вершины — листа дерева.

Пример. Я взял русско-английский словарь и ищу в нем перевод слова "поиск". Начальная область поиска, которой отвечает корень дерева, — это слова от "а" до "ящур". Открыв словарь где-то в середине, я увидел страницу со словами от "выхаживать" до "вьючить". На этой странице поиск не кончился, и оставшаяся область разбилась на две части: от "а" до "выучиться" и от "вьюшка" до "ящур". Помня русский алфавит, выбираем вторую. Теперь проще собрать данные в таблицу:

Область поиска		Л – левая	С – середина		П – правая	Выбор
от	до	до	от	до	от	
а	ящур	выучи-	выхаж-	вьючи-	вьюшк-	П
вьюшк-	ящур	кучно-	куш	лауре-	лафа	П
лафа	ящур	пересравно-	пересравно-	перешразби-	перешприби-	П
перешперешперешперешподсуподсу-	ящурравно-препо-помин-помин-пойти	препопоминподпопойти	препопоминподпокапокапоминпойти	приби-поощр-подсупол	попподсупола	Л Л Л П Л С

Итого девять шагов. Первые два деления были заведомо неудачными, зона поиска делилась далеко не поровну, и для дальнейшего поиска оставалась слишком большая область.

Последовательное дробление и уменьшение области поиска — это стандартный прием информационного поиска. Поиск нужной страницы в словаре, при N страницах текста и делении области каждый раз ровно пополам, требует примерно $\log_2 N$ шагов.

Можно оценивать правило поиска по разным *критериям качества* — часто критерием считают либо затраты на поиск *в самом худшем случае*, либо *математическое ожидание* затрат на поиск.

В отличие от поиска в книге, которую мы раскрываем в случайно выбранном месте, компьютерное дерево поиска уже подготовле-

но, хотя и здесь могут быть разные варианты организации. Так, мы можем иметь дело с уже зафиксированной информационной базой, а можем — с информацией, развивающейся во времени, т. е. динамически обновляемой. Например, при трансляции программы компилятор создает справочник идентификаторов и по мере разбора программы добавляет в него встретившиеся идентификаторы.

6.5. Информационные деревья

В этом параграфе мы рассмотрим некоторые интересные схемы хранения информации, в которых участвуют деревья. Деревья нам уже встречались, и еще подробнее мы будем их рассматривать, когда пойдет речь о графах: ведь дерево — это частный случай более общей конструкции, графа. Но сейчас наступил подходящий момент для рассмотрения некоторых замечательных механизмов хранения информации, связанных с деревьями, и придется забежать немного вперед.

6.5.1. АВЛ-дерево

Если выбрать в качестве механизма поиска двоичное дерево, то каждый новый идентификатор можно "прицепить к дереву" в том месте, где он должен быть. Однако такое естественное развитие дерева может сильно вытянуть его в направлении какой-либо ветви или нескольких ветвей, и время поиска информации в дереве станет слишком большим. Чтобы время росло не очень быстро, дерево нужно регулярно перестраивать, *балансировать*, и желательно, чтобы перестройки проводились достаточно экономно.

Московские математики Г. М. Адельсон-Вельский и Е. М. Ландис [2] предложили в 1962 г. схему самобалансирующегося поискового дерева (см. также [30], где дерево названо "подравненным"); эта схема известна сейчас как АВЛ-дерево. Их конструкция позволяет поддерживать (подравнивать) приблизительное равенство левой и правой ветвей двоичного дерева во всех его узлах с помощью небольших преобразований, затрагивающих каждый раз не более трех узлов.

Назовем *высотой дерева* длину наибольшего пути в этом дереве. Назовем вершину дерева *сбалансированной*, если высота ее левого и правого поддерева различается не больше чем на 1. Назовем дерево *подравненным*, если все его вершины сбалансированы.

С добавлением в дерево новых вершин некоторые вершины могут стать несбалансированными. Понятно, что сбалансированность может

пропасть только у вершин, лежащих на пути от корня дерева к новой вершине, и разность высот достигает значения 2. Возьмем самую нижнюю вершину с нарушенным балансом, назовем ее a. Пусть для определенности левый путь длиннее правого на 2. Нас будут интересовать две следующие вершины на этом пути, назовем их b и c. На рис. 6.5 показаны две возможные схемы взаимного расположения вершин (изображена только часть дерева вниз от a).

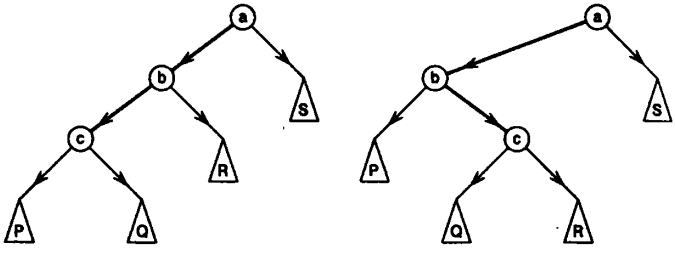


Рис. 6.5. Два типа пути исправления в поисковом дереве

Все остальные части дерева изображены условными блоками. Оказывается, достаточно небольшого изменения взаимного расположения вершин a, b и c и прицепленных к ним деревьев, чтобы баланс восстановился во всех вершинах. На рис. 6.6 показано для обоих случаев, представленных на рис. 6.5, какой становится эта часть дерева после изменений.

Обратите внимание на то, что фактически нужна только одна операция — поднятие вершины на один уровень вверх. Для первого случая нужно поднять вверх вершину b один раз, а во втором — вершину c два раза.

Докажем, что полученное дерево окажется подравненным. Введем некоторые обозначения. Пусть $H(i)$ — максимальная длина пути, начинающегося из вершины i, а h_k — максимальная длина пути в дереве k. По предположению $H(a) = H(b) + 1 = H(c) + 2$, $h_S = H(c) - 1$.

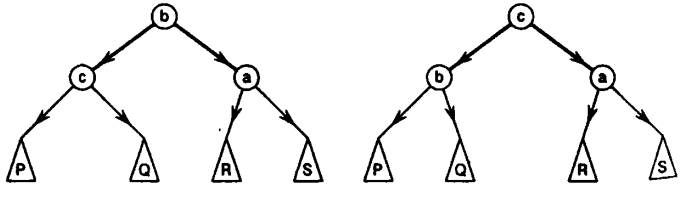


Рис. 6.6. Новое расположение вершин

Рассмотрим первый и второй случаи отдельно. В первом случае $h_R < H(c)$, так что после перестройки дерева $H(a) = h_S + 1 = H(c)$ и высота верхней вершины стала меньше. Во втором случае высота также уменьшилась на единицу, так как определяющие максимальный путь поддеревья Q и R поднялись на одну ступень выше.

Случаи, когда самым длинным оказался путь вправо, симметричны.

Пример. На рис. 6.7 показаны последовательные изменения дерева при добавлении в него записей с ключами 9, 11, 13, 17, 19, 23, 29, 31, 37, 18, 44, 75. Но еще лучше вы сможете разобраться в алгоритме, поработав с демонстрационной программой.

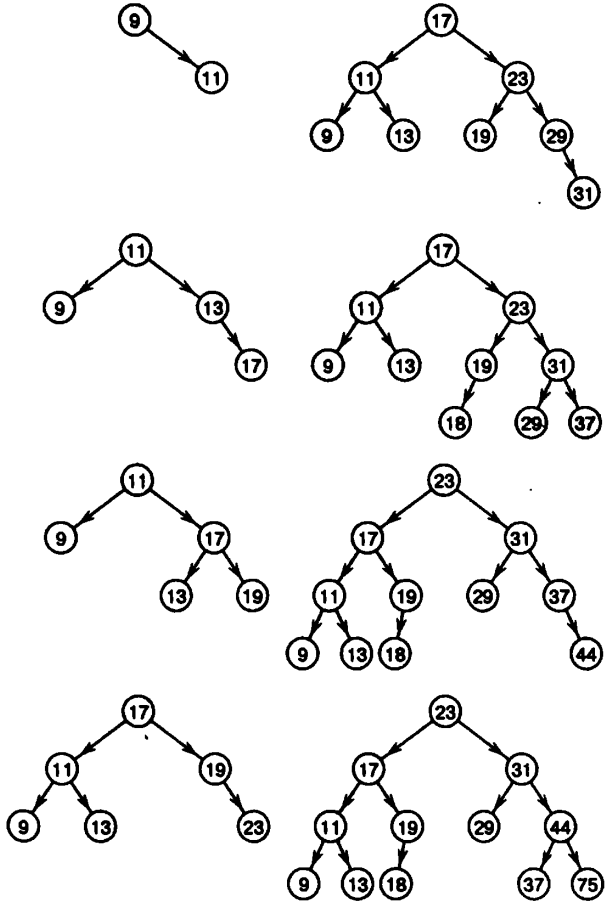


Рис. 6.7. Пример последовательных изменений АВЛ-дерева

АВЛ-деревья сейчас редко используются непосредственно, хотя в некоторых схемах, где требуется быстрый доступ к часто меняющейся информации, применяется конструкция *адаптирующегося*, т.е. приспосабливающегося дерева, основанная на АВЛ-преобразованиях.

6.5.2. *B*-дерево

Что же касается информационных систем, то здесь АВЛ-дерево уступило место более прогрессивной конструкции *B*-дерева*) и различным его обобщениям. Дело в том, что в современных информационных системах данные хранятся преимущественно во внешней памяти, а время, расходуемое на одно обращение к диску, существенно больше времени, требуемого на сопутствующие вычисления. Поэтому целесообразно в каждом узле дерева ветвить поиск не на два варианта, а на много.

Такое дерево при добавлении к нему новых записей в случае нехватки места расщепляет вершины. При расщеплении корневой вершины дерево "растет вверх", добавляя еще один уровень.

Будем рассматривать вариант, который называется *B*+-деревом. В этом дереве каждый лист содержит упорядоченный по ключу набор записей, а узел — граничные значения ключа у потомков данного узла. В простейшем случае в качестве границ можно брать наибольшие значения ключа в данном листе или у потомков данного узла.

Ограничимся демонстрацией на примере, как работает операция включения нового элемента в самом простейшем ее виде.

Пример. Начнем составлять в форме *B*+-дерева словарь сказки П. П. Ершова "Конек-горбунок". Возьмем начальный текст:

```
за_горами_за_лесами_за_широкими_морями_не_на_небе_на_земле_жил_
старик_в_одном_селе_у_крестьянина_три_сына_старший_умный_был_
детина_средний_сын_и_так_и_сяк_младший_вовсе_был_дурак_братья_
сеяли_пшеницу_да_возили_в_град_столицу_знать_столица_та_была_
недалече_от_села_там_пшеницу_продавали_деньги_счетом_принимали_
и_с_набитою_сумой_возвращалися_домой_
```

Для узлов нижнего уровня — *листьев* — выберем размер в 60 символов. Создадим первый лист L1 и заполним его, читая текст слово за словом и включая

*) По-видимому, буква *B* происходит от bounded, хотя предложившие эту конструкцию в 1972 г. Байер (R. Bayer) и Мак-Грейт (E. M. McGreight) выбор названия не объяснили.

в буфер по алфавиту отсутствующие в нем слова. Получаем (повторяющиеся слова пропускаются):

```
L1 горами_жил_за_земле_лесами_морями_на_не_небе_широкими_------
```

Неиспользованная часть буфера заполнена знаками тире. Мы остановились на слове "старик", для него места уже мало. Добавим еще один лист и поделим между листьями текст, а также создадим справочный узел, который хранит имена подчиненных ему узлов и их последние ключи, роль которых играют слова.

Нам для иллюстрации интересно, чтобы переполнения происходили быстрее, поэтому условимся, что справочный узел содержит ссылки не более чем на три вершины. Получаем:

```
L1 горами_жил_за_земле_лесами_--------------------------------
L2 морями_на_не_небе_старик_широкими_-------------------------
B1 L1:лесами_+L2:широкими_
```

Теперь без помех мы вставляем в соответствующие листья текст до слова "старший", это слово переполняет буфер L2, от которого отпочковывается L3:

```
L1 в_горами_жил_за_земле_крестьянина_лесами_------------------
L2 морями_на_не_небе_одном_селе_старик_-----------------------
L3 старший_сына_три_у_широкими_-------------------------------
B1 L1:лесами_+L2:старик_+L3:широкими_
```

Фраза "повторяющиеся слова пропускаются" означает, что при поступлении каждого следующего слова мы сначала проводим поиск — ищем это слово в словаре. Сейчас наш словарь немного вырос, и смотреть, как этот поиск проводится, стало интереснее. Чтобы проверить, есть ли в словаре очередное слово "сын", мы по справочнику корневого листа находим нужный узел следующего уровня (значит, смотрим узел B1 и видим, что первое из ограничивающих слов, которое следует за "сын", — это "широкий" в листе L3) и поиск продолжаем в нем. Если это лист, то просматриваем включенные в него слова, а если это снова узел, поступаем, как в корне.

Легкая вставка новых слов останавливается появлением слова "дурак" (это не нарочно!), переполняющего лист L1. Его расщепление и появление L4 расщепляет узел B1 с появлением B2 и соединяющего их узла B3:

```
L1 был_в_вовсе_горами_детина_дурак_---------------------------
L4 жил_за_земле_и_крестьянина_лесами_-------------------------
L2 младший_морями_на_не_небе_одном_селе_средний_старик_-------
L3 старший_сын_сына_сяк_так_три_у_умный_широкими_-------------
B1 L1:дурак_+L4:лесами_
B2 L2:старик_+L3:широкими_
B3 B1:лесами_+B2:широкими_
```

Доведя пример до конца, имеем:

```
L1 братья_был_была_в_вовсе_возвращалися_возили_----------------
L7 горами_град_да_деньги_детина_домой_дурак_--------------------
L4 жил_за_земле_знать_и_крестьянина_лесами_----------------------
L2 младший_морями_на_набитою_не_небе_недалече_одном_------------
L5 от_принимали_продавали_пшеницу_-------------------------------
L8 с_села_селе_сеяли_средний_старик_-----------------------------
L3 старший_столица_столицу_сумой_счетом_сын_сына_----------------
L6 сяк_та_так_там_три_у_умный_мирокими_--------------------------
B1 L1:возили_+L7:дурак_+L4:лесами_
B2 L2:одном_+L5:пшеницу_+L8:старик_
B4 L3:сына_+L6:мирокими_
B3 B1:лесами_+B2:старик_+B4:мирокими_
```

Сказка длинная, вы можете продолжить.

"Перекосы" при развитии в случае *B*-деревьев не так страшны, но для защиты от них предусмотрены специальные средства. К сожалению, заниматься этим вопросом, так же как операцией удаления элементов из *B*-дерева, здесь у нас нет возможности.

6.5.3. Дерево ключей и суффиксное дерево

Раз уж речь идет об информационных деревьях, полезно вернуться и еще к двум важным конструкциям, используемым при хранении большого количества различных слов.

Дерево ключей — это естественная конструкция, которая приходит в голову каждому, кто начинает такими вопросами заниматься.

Возьмем, к примеру, метод сжатия информации LZ. На каждом шаге мы имеем какое-то множество строк, которые получились при обработке уже просмотренной части текста. В примере на с. 127, дойдя до 22-й строки, мы получаем набор строчек

```
1  2  3  4  5  6  7  8  9  10  11 12 13 14 15 16 17  18 19 20  21 22
е  х  в  л  и  _  м  ед в  еди _и в_ ве ло с  ип еде _в _з в_и им и_
0е 0х 0в 0л 0и 0_ 0м 1д 0в 8и  6и 3_ 9е 4о 0с 5п 8е  6в 6з 12и 5м 5_
```

Построим дерево, в котором каждая вершина соответствует одной из этих строк (начиная с пустой нулевой строки), а подчиненными ей оказываются вершины, строки которых продолжают строку данной. На переходах из вершины в вершину напишем символ соответствующего продолжения. Получающееся дерево изображено на рис. 6.8.

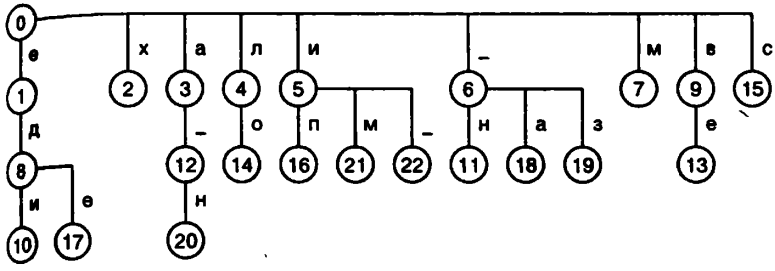


Рис. 6.8. Дерево накопленных строк в LZ-методе сжатия

На каждом пути от корня в любую другую вершину "прочитывается" соответствующая этой вершине строка. Например, строка еде читается на пути 0–1–8–17. Поиск наибольшей совпадающей строки в алгоритме LZ легко вести, передвигаясь по дереву. Начинаем с нулевой вершины и выбираем в ней переход с правильной буквой. Напомним, подошла очередь текста кот_задом_наперед, так что нужно выбрать переход с символом к. Алгоритм не находит продолжения и формирует следующую строку и соответствующую ему новую вершину дерева.

Хранить такое дерево можно, составляя для каждой вершины либо цепной список переходов с их метками, либо массив, в котором появляющийся новый переход заполняет нужную клетку. На рис. 6.9 показано такое представление для дерева из нашего примера, причем показаны только массивы, в которых хотя бы один элемент уже определен.

Рис. 6.9. Представление дерева накопленных строк массивами в вершинах. Слева от каждого массива — номер вершины, которой он соответствует

Использование таких массивов требует больше места в оперативной памяти, но зато затраты времени на поиск нужной строки пропорциональны длине строки *).

Похожее дерево получается и в коде Хаффмена. Отличие *кодового дерева* от дерева ключей в том, что из каждой вершины допускается переход не больше чем в две другие вершины (алфавит состоит из цифр 0 и 1), а кроме того, в том, что только терминальные вершины — листья — соответствуют кодовым строкам.

Вернемся к примеру, который мы рассматривали при описании алгоритма Хаффмена (см. с. 122). Были построены кодовые последовательности

```
_ 111    а 010    и 000    е 1100   о 1000   м 0110   н 0010
к 0011   в 11010  л 10100  д 10101  т 10010  р 10011  э 110110
с 101100 б 101101 п 011110 й 011100 у 011101 х 1101110 ш 1101111
ы 1011110 ч 1011111 ж 1011100 ю 1011101 я 0111110 ь 0111111
```

Им соответствует кодовое дерево, изображенное на рис. 6.10. Из каждой белой вершины выходят два перехода. Если считать, что переход вниз помечен символом 0, а переход вправо — символом 1, то на каждом пути от корня до листа читается соответствующая кодовая последовательность, например символу й соответствует строка 011100.

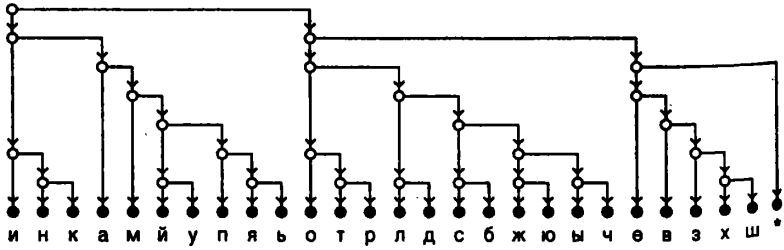


Рис. 6.10. Кодовое дерево для префиксных кодов Хаффмена. Знак пробела на рисунке изображен звездочкой

Дальнейшим развитием конструкции кодового дерева является уже рассматривавшееся на с. 103 суффиксное дерево. Можно считать, что суффиксное дерево получается из кодового удалением лишних

*) Приблизительно такое сочетание деревьев с массивами принято называть trie. Это слово создано искусственно как центральная часть слова retrieval. Переводчики книги [25] предложили использовать аналогичную часть слова "получение" и выбрали слово *луч*. В лучах нет меток переходов, а метка листа содержит дерево полностью

вершин — тех, из которых можно перейти только в одну вершину следующего уровня, — и конкатенацией меток соединяемых дуг. На рис. 6.8 так можно укрупнить переходы с получающимися метками ед, а_н, по и ве. Кроме того, нужно принять специальные меры, чтобы никакое слово, включаемое в дерево, не было началом другого слова.

Упомянем еще интересный вариант суффиксного дерева для двоичного алфавита, который называется PATRICIA *). Здесь текст представляется записанным в двоичном коде, и все ветвления в дереве делаются по очередному биту. Но контролируются не все биты, а только избранные, информативные. Лучше всего посмотреть эту конструкцию на небольшом примере.

Пример. Создадим "информационную систему" PATRICIA для поиска слов из фразы

```
и_падали_два_башмачка_со_стуком_на_пол
```

Переведем текст в шестнадцатеричное представление (в кодировке cp866, см. с. 25), а затем в двоичное:

```
и          A820                    1010 1000 0010 0000
падали     AFA0 A4A0 ABA8 20       1010 1111 1010 0000 1010 0100 1010 0000
                                   1010 1011 1010 1000 0010 0000
два        A4A2 A020               1010 0100 1010 0010 1010 0000 0010 0000
башмачка   A1A0 E8AC A0E7 AAA0 20   1010 0001 1010 0000 1110 1000 1010 1100
                                   1010 0000 1110 0111 1010 1010 1010 0000
                                   0010 0000
со         E1AE 20                 1110 0001 1010 1110 0010 0000
стуком     E1E2 E3AA AEAC 20       1110 0001 1110 0010 1110 0011 1010 1010
                                   1010 1110 1010 1100 0010 0000
на         ADA0 20                 1010 1101 1010 0000 0010 0000
пол        AFAE AB                 1010 1111 1010 1110 1010 1011
```

Взяв по два первых байта и упорядочив битовые строки лексикографически, получаем

```
башмачка   1010 0001 1010 0000     .0.. 00.. .... ....
два        1010 0100 1010 0010     .0.. 01.. .... ....
и          1010 1000 0010 0000     .0.. 10.. .... ....
на         1010 1101 1010 0000     .0.. 110. .... ....
падали     1010 1111 1010 0000     .0.. 111. .... 0...
пол        1010 1111 1010 1110     .0.. 111. .... 1...
со         1110 0001 1010 1110     .1.. .... .0.. ....
стуком     1110 0001 1110 0010     .1.. .... .1.. ....
```

Справа выписаны только информативные биты. Первый бит не информативен — он во всех строках равен 1. Значение второго бита разделяет слова на

*) Это название составлено из Practical Algorithm To Retrieve Information Coded In Alphanumeric.

два подмножества. В первом подмножестве следующий разделяющий бит — пятый, а во втором — десятый и т. д.

Если мы будем искать в этой системе, например, слово "лампа", двоичное представление которого начинается с 1010 1011 1010 0000, то, использовав для сравнения второй бит (0), а затем пятый (1) и шестой (0), мы придем к такому, типичному для этой системы, выводу: в системе слово, самое похожее на "лампа", — это "и"; если они не совпадают, то слово "лампа" в систему не входит. Осталось проверить совпадение!

Читателю полезно самому нарисовать получающееся двоичное дерево поиска. В каждой вершине этого дерева нужно помнить номер проверяемого бита, а в листе — номер или адрес "рекомендуемого слова".

6.5.4. Биномиальные деревья

Скоро нам понадобится еще один класс деревьев, введенный Вуйлеменом [84]. Определим последовательность деревьев BT_0, BT_1, BT_2, BT_3, ... следующим образом.

Дерево BT_0 состоит из одной вершины, являющейся в этом дереве корнем. Дерево BT_k при $k > 0$ составлено из двух деревьев BT_{k-1} добавлением перехода из одного корня в другой. Такие деревья называются *биномиальными*. На рис. 6.11 изображены кроме общей схемы (слева) биномиальные деревья для нескольких k.

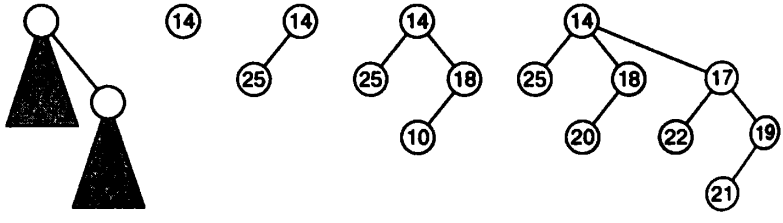


Рис. 6.11. Биномиальное дерево. Слева — общая схема соединения двух деревьев в одно, справа — деревья BT_0, BT_1, BT_2 и BT_3

Упражнение 6.6. Назовем *уровнем* вершины в дереве BT_n количество переходов в пути от корня дерева до этой вершины. Докажите, что при любом $k \in 0:n$ число вершин уровня k равно C_n^k. Этим фактом и объясняется название деревьев BT_n — *биномиальные деревья*.

Легко видеть, что количество вершин в дереве BT_n равно 2^n.

Пусть каждой вершине дерева сопоставлен ее *ключ* — некоторое число. Ключом дерева будем называть ключ его корня. Скажем, что дерево BT_n, $n > 0$, соединено монотонно, если его ключ не больше

ключей составляющих его деревьев BT_{n-1}. Скажем, что дерево BT_n образует кучу*) (heap), если $n = 0$ или оно составлено монотонно из куч.

Деревья BT_0, BT_1 и BT_3 на рис. 6.11 являются кучами, а BT_2 — нет, из-за того что дерево из вершин с ключами 10 и 18 составлено не монотонно.

При составлении дерева BT_n из 2^n отдельных вершин очень просто добиться того, чтобы оно было кучей, — нужно каждый раз составлять деревья монотонно, т.е. правильно выбирать, какой из двух корней к какому присоединять.

Биномиальные деревья скоро нам понадобятся. Отметим, что есть разновидность таких деревьев, называемая *фибоначчиевыми деревьями*. Может оказаться естественным, что в фибоначчиевом дереве n-го порядка число вершин равно соответствующему числу Фибоначчи F_n.

6.5.5. Квадродеревья

Интересное использование деревьев встречается в задачах экономного хранения графических образов. Начнем с примера. Вспомним "битовый цветочек" со с. 28. Рисунок кодировался битовой таблицей размера 16 × 16. Для более экономного кодирования можно использовать схему выделения "полностью закрашенных" блоков.

Разобьем исходный квадратный рисунок на четвертинки, полностью закрашенные части оставим без изменений, а остальные "четвертуем" так же, как исходный рисунок. Так будем поступать, пока все элементы разбиения части не станут полностью закрашенными. На рис. 6.12 мы нарисовали только два уровня разбиения, но и из них видно, что большая часть изображения уже попала в полностью закрашенные блоки.

Такое разбиение изображения называется *квадродеревом***). Квадродеревья выгодно использовать, когда в изображении имеется много монохромных участков.

Квадродеревья используются и в задачах поиска точки на плоскости: область поиска четвертуется, а затем по мере надобности каждая из частей может снова четвертоваться. Имеются варианты квадродеревьев для пространств большей размерности и для изображений с бо́льшим числом цветов.

*) Напомним, что этот способ упорядочения нам уже встречался в иерархической сортировке (Heapsort) на с. 158.

**) Английский термин — quadtree. Встречаются другие переводы этого термина — *четрево* (в переводе книги [25]) и *квадрантное дерево*.

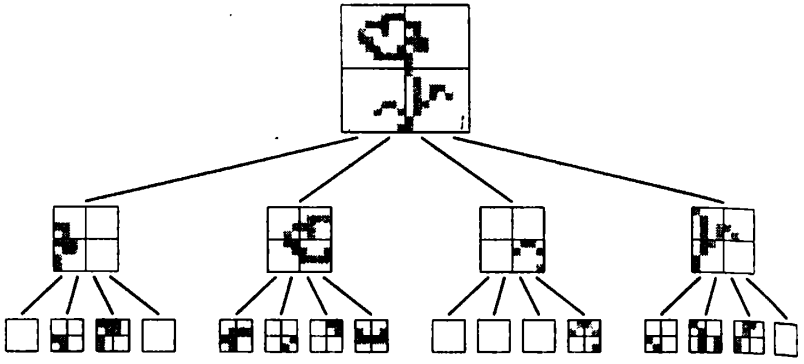


Рис. 6.12. Битовый цветочек в виде квадродерева. Показаны только два уровня разбиения, но хорошо видно, какими будут остальные два уровня

6.6. Хеширование

Наряду с поисковыми деревьями часто используется конструкция, основанная на так называемом *хешировании*. Это слово произведено от английского глагола to hash (перемалывать). Чтобы объяснить, что это такое, нужно начать издалека. Мы уже говорили о вычислительных удобствах работы с обычным одномерным массивом.

Говорилось также и о том, что если диапазон возможных значений ключей-индексов слишком велик, то даже при задании ключей числами думать о непосредственном использовании массивов не приходится. Можно усовершенствовать конструкцию массива, выбрав для индексов приемлемый диапазон I и задав отображение $\Phi : K \to I$ множества ключей K в множество I. Никто, конечно, не ожидает однозначности этого отображения, и в одно значение индекса может отобразиться несколько значений ключа. Это не так страшно, как кажется, просто мы не должны рассчитывать на возможность поместить информацию в один элемент массива; этот элемент должен быть головой цепного списка записей или блоков записей. При поиске элемента по ключу мы находим соответствующий ключу элемент массива и в его цепном списке ищем запись с нужным ключом. Поиск существенно упрощается, если записи распределены по элементам массива достаточно равномерно.

Вот мы и дошли до хеширования. Выгодно подбирать функцию Φ, которая ровнее распределяла бы ключи по индексам, размешивала бы

их. Обычно никаких сведений о совокупности ключей нет, и лучше всего сделать преобразование, которое "перемалывает" ключ.

Пример 1. Ключом записи о студенте в университетской информационной системе "Студент" естественно считать номер зачетной книжки. В нашем университете принято, чтобы первые цифры номера определяли год поступления, дальше следовал диапазон, выделенный данному факультету, а уже потом — подряд, как получится. Таким образом, номер зачетной книжки — это большое целое число.

Преобразовать номер зачетки k в индекс массива можно так. Вычислим k^2, возьмем несколько средних цифр результата, получим какое-то число t и вычислим остаток от деления t на размер диапазона индексов nl. Пусть $nl = 291$. Имеем:

Номер	Квадрат номера	Вырезка	Индекс
98010315	9606021846399225	602184639	6
98010316	9606022042419856	602204241	111
98010317	9606022238440489	602223844	217
98011315	9606217868029225	621786802	118
98011316	9606218064051856	621806405	224
98011317	9606218260074489	621826007	38

Вы видите, как разбрасываются значения индекса при небольших изменениях исходного номера. Эта хэш-функция взята в качестве варианта, простого для объяснений. Распределение, которое она дает, оказывается хуже, чем у других функций.

Пример 2. Какую можно предложить хеш-функцию для превращения строки символов в индекс? Сопоставим каждому символу его ASCII-код, получим последовательность чисел $d = \{d_i\}$. Что с ними делать дальше? Складывать? Нет, близкие строки получат близкие индексы. Умножать? Тоже плохо. Хорошо вычислять "взвешенную сумму" этих чисел, когда каждое слагаемое берется со своим множителем. Из таких формул удобнее всего полиномиальная $h(d) = \sum_i r^i d_i$, в которой i-е число d_i берется с множителем r^i, где r — некоторое целое число, лучше всего взаимно простое с длиной диапазона. Так, при $nl = 291$ и $r = 73$ получаем (здесь мы взяли не коды цифр, а их числовые значения):

Номер	Накопленные суммы								Индекс
98010315	75	134	263	144	9	165	247	144	144
98010316	75	134	263	144	9	165	247	217	217
98011315	75	134	263	144	82	115	62	187	187
9801131	75	134	263	144	82	115	62		62
980113	75	134	263	144	82	115			115

Такая формула позволяет вычислять индексы для строк переменной длины и не требует длинных вычислений, так как все расчеты ведутся по модулю nl, т.е. в остатках от деления на размер диапазона.

Пример 3. Хеш-функции используются для организации *ассоциативных массивов*, в которых индексами служат строки. Имеются языки программирования, в которых есть только ассоциативные массивы. Мы не должны тратить место на языки программирования, но не можем умолчать об AWK [56]. Посмотрите, как на этом языке выглядит программа для подсчета встречаемости слов в тексте:

```
{ for (i=1; i<NF; i++) a[$i]++ }
END{ for (w in a)  print "a[" w "]=" a[w] }
```

Первая строка определяет цикл по всем строкам входного текста. Каждая прочтенная строка разбивается на слова `$1`, ..., `$k`. Число слов в строке `k` записывается в системную переменную `NF`. Для каждого слова `$i` соответствующий элемент массива увеличивается на единицу. (Откуда он взялся, этот элемент? Первое обращение к нему создало его и записало в него нулевое значение.) Вторая строка, начинающаяся с END, содержит действия, выполняемые по завершении чтения, — в данном случае это цикл по всем созданным элементам. В цикле для каждого элемента печатается его индекс и значение.

Пример 4. *Поиск близких точек в пространстве.* Пусть на плоскости задано некоторое множество точек $S = \{s_i\}$, $i \in 1:m$, и требуется найти те пары точек $i, j \in 1:m$, расстояние между которыми меньше заданного порогового значения d. Под расстоянием понимается обычное евклидово расстояние $\rho(s_i, s_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Попарное сравнение точек имеет, очевидно, трудоемкость $O(m^2)$ и при больших значениях m недостаточно эффективно.

Для ускорения сравнений очень полезным оказывается *алгоритм П. Элайеса* [55]. Найдем прямоугольник, содержащий все точки, и разобьем его на достаточно мелкие одинаковые квадраты. Каждый такой квадрат будет адресоваться двумя целыми числами i_X и i_Y — порядковыми номерами по оси X и по оси Y. Каждая из точек попадет в один из этих квадратов, его индексы легко вычисляются по координатам. Теперь для поиска близких точек достаточно сравнивать между собой точки одного квадрата и точки близких квадратов.

При чем же здесь хеширование? Очень просто. Таких квадратов может быть слишком много, но многие из них могут оказаться пустыми. Чтобы избежать создания громадного массива, который по большей части не нужен, мы введем уже упомянутый *ассоциативный массив* — специальную информационную конструкцию, которая внешне будет выглядеть "почти как массив". При ее создании нам и понадобится хеширование.

Каждому квадрату, т. е. определяющей его паре чисел (i_X, i_Y), сопоставим индекс $\Psi(i_X, i_Y) = (a \cdot i_X + b \cdot i_Y) \bmod D$, где коэффициенты a и b выбраны взаимно простыми между собой и с D — размером хеш-массива. Например, можно взять $a = 17$, $b = 29$. Дальнейшее читатель может "домыслить" сам.

6.7. Приоритетные очереди

Завершим эту главу одним специальным классом информационных механизмов. В специальных ситуациях можно иногда обеспечить очень простые и эффективные конструкции. Например, вспомним реализацию алгоритма Хаффмена на с. 122: мы имели набор символов, настоящих или условных, каждому из которых сопоставлялась вероятность (вероятности могли быть одинаковыми, но тогда это не было для нас существенно). Исходные символы были упорядочены по возрастанию вероятностей, а из новых символов составлялась новая очередь, элементы которой благодаря особенностям задачи также были упорядочены. Поэтому простым было и добавление новых элементов — в конец очереди, — и извлечение очередного минимального элемента — ... как здесь сказать? — из двух упорядоченных списков, подчиняясь логике их слияния. Сейчас мы рассмотрим похожую задачу для более общей схемы формирования новых элементов.

Представьте себе, что нам нужно оперативное хранилище данных, где главную роль играют характеризующие их числовые значения, которые, как обычно, мы будем называть *ключами*. Элементы с ключами добавляются в хранилище и по запросу удаляются из него. При этом добавление происходит в произвольном порядке, а при удалении каждый раз нужно выбирать элемент со значением ключа, наименьшим из имеющихся в хранилище.

Такое хранилище называется *приоритетной очередью* (priority queue). Приоритетные очереди встречаются в различных ситуациях. В моделировании случайных процессов обслуживания возникает совокупность будущих событий, изменяющих ход процесса.

Пример. Пусть мы моделируем работу буфета. В модели есть объекты "Буфетчица", "Очередь посетителей", "Столики", "Уборщица". Объект "Буфетчица" обслуживает очередного посетителя, и определено время t_1 этого обслуживания, разыгрываемое случайно. Объект "Очередь" может увеличиться или уменьшиться (кто-то уйдет), определено время t_2 этого изменения. Объект "Столики" может занять или освободить один столик, убрав или оставив грязную посуду, — в момент t_3. Объект "Уборщица" может пойти убирать со столиков посуду в момент t_4. Ближайшее изменение в системе наступит в момент $\tau = \min\{t_1, t_2, t_3, t_4\}$. Объект, которому пришло время изменить свое состояние, получает новое описание ближайшего будущего с новым моментом его изменения.

Часто таких объектов в системе много, и само множество объектов в ходе моделирования изменяется. Искать минимальное t_i становится

обременительно, и хочется не повторять много раз поиск минимума среди одних и тех же чисел. Немного позднее нам встретятся и другие примеры такого же типа.

Рассмотрим сейчас несколько вариантов организации информационной системы, у которой кроме очевидных операций инициализации и закрытия есть две основные операции:

`insert` — включение в систему элемента с данным значением ключа;

`extract` — извлечение элемента с минимальным значением ключа.

6.7.1. Простейшие приоритетные очереди

В качестве приоритетной очереди может использоваться обычный неупорядоченный цепной список. В такой конструкции операция добавления нового элемента выглядит просто: прицепляем его к началу или к концу списка. А вот операция поиска минимального элемента требует просмотра всего списка.

Если упорядочить элементы цепного списка по возрастанию ключей, то, напротив, поиск минимального элемента будет прост — надо лишь взять голову списка, а запись усложнится — для каждого нового элемента нужно будет разыскивать его законное место в списке.

В первом варианте операция `insert` проста, а `extract` — сложна, ее трудоемкость линейно зависит от числа элементов в списке, который нужно просмотреть при поиске минимума.

Во втором варианте, напротив, проста операция `extract` — нужно просто взять первый элемент списка, а операция `insert` более трудоемка из-за необходимости поиска правильного места для нового элемента.

Ниже будут рассмотрены две достаточно общие конструкции, позволяющие удачно сбалансировать трудности обеих операций.

6.7.2. "Корзинная" приоритетная очередь

Начнем с сильно упрощенной схемы. Пусть значения ключа целочисленны, неотрицательны и поступающие на хранение значения не бывают меньше, чем уже взятые из хранилища.

Заведем на каждое возможное значение ключа специальное хранилище, а сами эти хранилища организуем, например, в массив. Каждое такое хранилище (его принято называть bucket — *корзина* или *ведро*)

легко и вполне эффективно представляется цепным списком. Что же касается массива корзин, то мы отложим пока вопросы его рациональной организации.

Операция `insert` в корзинной системе очень проста — нужно добавить новую запись в корзину, соответствующую данному значению ключа. По причинам, излагаемым ниже, простейшее "стековое" добавление нового элемента в начало списка оказывается самым удобным, им мы и ограничимся.

Операцию `extract` удобно описывать как элементарный шаг простого и естественного процесса: просматриваются все корзины, начиная с нулевой, и при нахождении ближайшей непустой корзины из нее извлекаются одна за другой все записи. Шагом этого процесса является извлечение одного элемента.

Таким образом, в корзинной системе трудоемкость каждого включения константна, а трудоемкость всего процесса извлечения состоит из трудоемкости, пропорциональной суммарному числу элементов в корзинах (это затраты на извлечение элементов), и трудоемкости просмотра, линейно зависящей от числа просмотренных корзин.

Если предположить, что каждое новое поступающее в систему значение отличается от уже имеющихся в ней значений не очень сильно, организация массива корзин может значительно упроститься.

В обстоятельной работе [63] описана многоуровневая корзинная система, в которой записи, далекие от использования, помещаются в "крупные" корзины, с большими диапазонами значений; при приближении уровня используемых значений к этому диапазону содержимое корзины раскладывается по "более мелким корзинам".

6.7.3. Биномиальная куча

Биномиальная приоритетная очередь составляется из нескольких биномиальных деревьев. Сначала мы рассмотрим простую организацию такой очереди, а затем несколько ее улучшим. Ввиду предстоящего улучшения некоторые вычислительные детали первого варианта мы изложим по возможности сжато.

Итак, простейший случай. Пусть очередь содержит n элементов. Рассмотрим двоичное разложение числа n:

В терминах этого разбиения можно сказать, что хранилище будет состоять из деревьев $BT_{i_1}, BT_{i_2}, \ldots, BT_{i_k}$, каждое из которых содержит соответствующее количество элементов, составляющих кучу. Следовательно, в каждом дереве элемент с наименьшим ключом будет находиться в корне дерева. Деревья хранилища соединяются в закольцованный двусторонний цепной список, а внешний указатель указывает на дерево с минимальным ключом.

На рис. 6.13 показана такая система с 11 элементами.

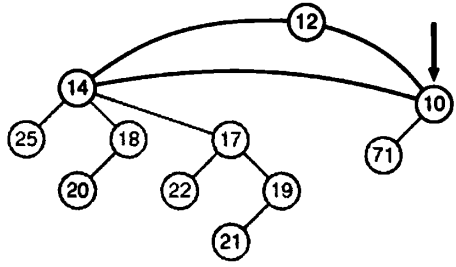


Рис. 6.13. Биномиальная приоритетная очередь из трех деревьев

Операция `insert` управляется ритмом прибавления единицы к числу в двоичной записи. Я имею в виду, что, хотя действия будут другими, общее управление будет таким же.

Что происходит при добавлении 1 к числу 11? Или в двоичном представлении при добавлении 1 к 1011_2? Поскольку в последнем разряде 1, то 1 + 1 = 10, ноль пишем, один в уме. Во втором разряде есть единица, складываем старые 10 и новые 10, получаем 100, записываем 1 в третий разряд и получаем 1100_2.

И с деревьями так же. Из нового элемента образуем новое дерево BT_0. Если такого дерева в системе нет, то добавляем его в систему, а если было — соединяем два дерева BT_0 в одно дерево BT_1, с которым поступаем точно так же. И точно так же мы могли бы поступать начиная с дерева любого порядка. Назовем описанную операцию *прибавлением дерева к набору деревьев*.

Описание изменений цепного списка при прибавлении дерева мы опустим. Важно только, что мы изменим внешние указатели не более одного раза.

Операция `extract` легко находит дерево с минимальным ключом и удаляет его корень. В результате дерево "рассыпается" на несколько меньших. Из этих деревьев и деревьев, входивших в цепной список,

формируется новый цепной список, примерно так же, как это делается в операции `insert`.

Рассмотрим теперь вариант этой схемы, называемый "ленивым". Это название относится к исполнению операции `insert`: очередное дерево BT_0 просто включается в цепной список с возможным изменением указателя на минимальный ключ. Таким образом, трудоемкость этой операции не превосходит некоторого постоянного значения (в том смысле, что не зависит от размеров уже имеющихся деревьев) и будет, как говорят, *константной*.

На долю операции `extract` выпадает больше работы. После исключения корня дерева с минимальным ключом это дерево, как было сказано выше, "рассыпается", и набор деревьев перестраивается. Для этого мы пользуемся введенной выше операцией прибавления дерева к набору: берется пустой набор, к нему прибавляются все обломки рассыпанного дерева и все деревья, входившие в цепной список наравне с рассыпанным. Затем из полученного набора составляется заново цепной список.

Трудоемкость операции `extract` можно разбить на часть, соответствующую самому извлечению минимального элемента из очереди, и часть, соответствующую перестройке очереди. Поиск минимума имеет константную трудоемкость, затраты на рассыпание дерева пропорциональны порядку дерева и, следовательно, логарифму числа вершин в очереди, точнее, $k = \lceil \log_2 n \rceil$. Рассмотрим затраты на сборку очереди. Она собирается из "обломков рассыпанного дерева", других деревьев, оставшихся от прошлого раза, и из вновь добавленных деревьев нулевого порядка. Соединение в единую систему обломков и "других деревьев" имеет трудоемкость, пропорциональную k, — это затраты на сложение двух k-разрядных двоичных чисел. Присоединение "новых деревьев" пропорционально произведению их числа на k.

Таким образом, общая трудоемкость всех операций `extract` не превосходит по порядку $(n_e + n_i) \cdot \log_2 n_i$, где n_i — число включений, а n_e — число исключений.

Глава 7

Предикаты и отношения

Вначале мы рассмотрим математический объект, который может трактоваться как подмножество прямого произведения двух или большего числа множеств. Для математической логики — это очень важное понятие **отношения, или предиката,** для теории множеств — это частичное упорядочение, для теории графов — это ориентированный граф.

Для нас наиболее удобной будет терминология теории графов, и мы быстро перейдем к ней. Значительное место в оставшейся части курса займет теория графов, ее комбинаторные аспекты, связь теории графов с линейной алгеброй, экстремальными задачами на графах. Теория графов заслуживает повышенного внимания тем, что дает вам очень удобный и красивый язык, позволяющий описывать связи между объектами в наглядной и легко воспринимаемой форме.

Что же касается этой главы, то после общих сведений об отношениях и порядках мы посмотрим, как многомерные отношения используются в так называемых реляционных базах данных.

7.1. Определения

Пусть заданы два непустых конечных множества X и Y и рассматривается некоторое подмножество U их прямого произведения $U \subset X \times Y$. Это подмножество называется на языке математической логики (двуместным) *отношением*, а если оно задано отображением $X \times Y \to 0:1$, то (двуместным) *предикатом*. Сейчас для нас это различие в математической форме представления объекта не будет существенно, и мы не будем различать предикатов и отношений.

По аналогии с двуместными отношениями можно ввести и многоместные, но мы займемся ими только в самом конце главы.

Понятие отношения следует рассматривать как естественное обобщение знакомых нам отношений из разных разделов математики, каждое из них может рассматриваться как частный случай этого общего понятия. Так, отношение $>$ (больше) для целых чисел может быть описано как множество пар целых чисел, удовлетворяющих нужному неравенству. Отношение $\in$ (принадлежность элемента множеству) может быть описано как множество пар (элемент, множество), таких что элемент принадлежит множеству.

Рассмотрим еще несколько примеров.

Пример 1. Отношение отец задается множеством пар индивидуумов (x, y), таких что x — отец y.

Пример 2. Отношение непосредственного родства задается множеством пар индивидуумов, имеющих прямую родственную связь (обычно такими считаются отношения родителей с детьми и братьев/сестер друг с другом).

Отношение V на $Y \times X$ называется *обратным к отношению* U, если

$$V = \{(y, x) | (x, y) \in U\}$$

(естественно использовать обозначение $V = U^{-1}$). Отношение W на $X \times Z$ называется *композицией*, или *произведением*, отношений U на $X \times Y$ и V на $Y \times Z$, если

$$W = \{(x, z) \mid \text{существует такое } y \in Y, \text{ что } (x, y) \in U, (y, z) \in V\}.$$

Как правило, пишут $W = UV$ или $W = U \cdot V$.

Упражнение 7.1. Построить отношения на множестве клеток шахматной доски, определяющие их взаимную достижимость за один ход той или иной фигуры (ладья, конь, пешка).

Упражнение 7.2. Доказать, что $(U^{-1})^{-1} = U$.

Упражнение 7.3. Доказать, что $((V_1V_2)V_3) = (V_1(V_2V_3))$.

Наиболее важен случай, когда сомножители прямого произведения совпадают (обозначим это единственное множество через A). В этом случае мы будем говорить об отношениях на множестве A.

Отношение U называется *рефлексивным*, если $(x, x) \in U$ для любого x. Так, для чисел отношение $\geq$ рефлексивно, а отношение $>$ — нет.

Отношение U называется *симметричным*, если $U = U^{-1}$, т.е. если из $(x, y) \in U$ следует $(y, x) \in U$. Симметричными являются отношения равенства, а из рассмотренных нами примеров — отношение родства.

Отношение U называется *антисимметричным*, если из $(x, y) \in U$ и $(y, x) \in U$ следует $y = x$. Все отношения арифметического типа (равенство и неравенства) антисимметричны.

Отношение U называется *транзитивным*, если $U \cdot U = U$, т.е. если из $(x, y) \in U$ и $(y, z) \in U$ следует $(x, z) \in U$. Отношения типа $>$ и $<$, строгие и нестрогие, конечно же, транзитивны. Нетранзитивно отношение родства, поскольку индивидуумы, имеющие общих родственников, не обязательно должны быть родственниками.

Отношение U называется *эквивалентностью*, если оно рефлексивно, симметрично и транзитивно. Это определение заслуживает дополнительных комментариев.

Обозначим через E_x множество элементов, находящихся с элементом x в отношении эквивалентности (или, попросту, элементов, эквивалентных x). Легко показать, что $E_y = E_x$ для $y \in E_x$ и $E_x \cap E_y = \varnothing$ для $y \notin E_x$. (Покажите это сами!) Множества E_x называются *классами эквивалентности* данного отношения. Совокупность всех классов эквивалентности E_x образует разбиение множества A.

Упражнение 7.4 [13]. Пусть S — строка длины n. Рассмотрим ее суффиксы; каждый суффикс задается позицией его начального символа. Отношение E_k определяется над парами суффиксов S: $i\, E_k\, j$ в том и только том случае, если суффикс i и суффикс j строки S совпадают не меньше чем на k начальных символах. Очевидно, что E_k — отношение эквивалентности, так что оно разбивает элементы на классы эквивалентности. Далее, так как S имеет n символов, то каждый класс в E_n состоит из одного элемента. Проверьте следующие два факта:

1) Если $i \neq j$, то $i\, E_{k+1}\, j$ в том и только том случае, когда $i\, E_k\, j$ и $(i+1)\, E_k\, (j+1)$.

2) Любой класс E_{k+1} является подмножеством класса E_k, и таким образом, разбиение E_{k+1} является измельчением разбиения E_k.

7.2. Отношения порядка

Отношение U называется *частичным порядком*, если оно рефлексивно, транзитивно и антисимметрично. Говорят, что элемент i предшествует элементу j, если $(i, j) \in U$. Часто для задания предшествований используют специальный знак предшествования: $i \prec j$ или даже $i \prec_U j$, если требуется уточнить, о каком отношении порядка

идет речь. Говорят, что элементы i и j сравнимы в порядке U, если $i \prec j \vee j \prec i$. Элемент называется *максимальным* в частичном порядке U, если ему не предшествует никакой другой элемент.

В множестве, на котором задан частичный порядок, может быть несколько максимальных элементов. Очевидно, что максимальные элементы не сравнимы. Совокупность всех максимальных элементов множества называется его *границей Парето* *).

Пример. На множестве пар целых чисел легко задать частичный порядок, используя покоординатное предшествование

$$(x_1, y_1) \prec (x_2, y_2) \leftrightarrow x_1 \leq x_2 \wedge y_1 \leq y_2$$

(куда отнести случай равенства пар — не существенно).

Частичный порядок U называется *линейным* или *полным* на множестве A, если в этом порядке сравнимы все элементы из A.

Частичный порядок $\prec$ на множестве A называется *иерархическим* **), если (1) для любых различных $a, b, c \in A$ из $b \prec a$ и $c \prec a$ следует, что элементы b и c сравнимы, т.е. либо $b \prec c$, либо $c \prec b$, и (2) существует только один максимальный элемент.

Легко показать, что в иерархическом порядке каждому элементу a, кроме максимального, можно сопоставить *непосредственного предшественника*, такой элемент $p(a)$, что из $b \prec a$ и $b \neq p(a)$ следует $b \prec p(a)$.

Упражнение 7.5. Докажите, что у каждого элемента иерархически упорядоченного множества, кроме максимального элемента, есть непосредственный предшественник.

Упражнение 7.6. Рассмотрим некоторое множество строк A, в которое вместе с каждой строкой a входят все ее префиксы, в том числе пустая строка. Пусть $a \prec b$ в том и только том случае, если a является префиксом b. Докажите, что такой частичный порядок является иерархическим, а нулевая строка — максимальный элемент в нем.

*) Vilfredo Pareto (1848–1923), выдающийся итальянский экономист, один из пионеров математических методов в экономике. Набор доходов участников экономического процесса оптимален по Парето, если не существует другого набора, который для одного из участников лучше данного, а для других не хуже. Здесь идея Парето перенесена на более общий случай.

**) Слово *иерархия* буквально означает "святая власть". Эта схема подчинения нижестоящих вышестоящим наблюдалась в наиболее чистом виде у жрецов в Древнем Египте. Она принята и в христианских церквах (и даже в небесном воинстве). Нам это слово уже встречалось, когда мы рассматривали сортировку Heapsort (с. 158). Там же говорилось об иерархической упорядоченности массива a: из $i \prec j$ следовало, что $a[i] \geq a[j]$.

Упражнение 7.7. Пусть $\mathscr{P}_1, \ldots, \mathscr{P}_k$ — k разбиений множества A, последовательно измельчающие друг друга, т.е. для любого $i \in 1 : k-1$ разбиение $\mathscr{P}_{i+1}$ мельче, чем разбиение $\mathscr{P}_i$. Рассмотрим совокупность всех элементов всех этих разбиений, удалив совпадения и дополнив ее множеством A. Покажите, что отношение включения для множеств этой совокупности является иерархическим порядком, а множество A — максимальным элементом.

7.3. Отношения в базах данных

Трудно обойти вопрос о том, как понятие отношения используется в теории *реляционных баз данных*. Когда говорят о базах данных, то основные темы обсуждения — это поддержание баз в рабочем состоянии, защита информации при внесении изменений, обеспечение целостности информации, правила одновременного доступа нескольких пользователей. Важных вопросов очень много, и почти ни об одном из них нет возможности говорить. Здесь мы коснемся только тех базовых понятий, которые связаны с представлением данных в виде многоместных отношений.

В реляционной базе данных вся информация хранится в виде набора таблиц, а каждую таблицу можно трактовать как отношение. Начнем с формальных определений.

Пусть задано некоторое конечное множество K, элементы которого будут называться *атрибутами* отношения. Количество атрибутов называется *арностью* или *степенью* отношения. Каждому атрибуту $k \in K$ сопоставляется множество его возможных значений D_k, именуемое *доменом* (domain). Разным атрибутам могут соответствовать одинаковые домены, это нас не должно удивлять. Введем прямое произведение доменов $D(K) = \mathrm{X}_{k \in K} D_k$ и будем называть его подмножества *отношениями*, а само K — *схемой этих отношений*. Отдельные элементы отношения называются *кортежами*. Кортеж состоит из *компонент* — текущих значений атрибутов *).

Итак, отношение — это таблица, в которой каждая строка есть кортеж, а каждый столбец соответствует одному атрибуту. Такую конкретную таблицу называют *текущим значением* отношения K. Предполагается, что все строки таблицы различны (от совпадения строк защищает система формирования таблицы).

Пример. Экзаменационная ведомость — это таблица со столбцами ФИО (фамилия, имя, отчество), НОМ-ЗАЧ (номер зачетки), ОЦЕНКА. Набор из этих трех атрибутов является схемой отношения, а конкретная заполненная ведомость — текущим значением отношения. Список студентов факультета —

*) В математическом тексте используется слово "компонента" — женского рода, а в остальных случаях "компонент" — мужского. Базы данных — это пограничная территория, где возможны споры. Слово arity (*арность* или *местность*) было придумано математиками в XX столетии по аналогии со словами unarity, binarity, ternarity.

это тоже отношение, но обычно с гораздо бо́льшим набором атрибутов: кроме уже встречавшихся ФИО (для минимального по информативности списка) и НОМ-ЗАЧ могут появиться ДАТ-РОЖД (дата рождения), ДАТ-ЗАЧ (дата зачисления), АДРЕС, КУРС, ГРУППА, СТИП (стипендия) и т.д. Можно считать, что атрибуты ДАТ-РОЖД и ДАТ-ЗАЧ имеют один и тот же домен ДАТА (хотя, конечно, эти даты лежат в разных диапазонах значений).

При описании какой-либо совокупности данных (предметной области) мы разрабатываем совокупность схем отношений, которые будем использовать для представления информации; эта совокупность называется *схемой (реляционной) базы данных*. Схема базы данных — это скелет базы, она должна быть очень устойчивой и хорошо продуманной на уровне проекта.

Саму же *реляционную базу данных* составляет набор описанных в схеме таблиц — значений отношений. При развитии и эксплуатации базы данных эти текущие значения будут часто изменяться.

Для получения из базы данных информации выполняются преобразования таблиц, при этом большой диапазон функций выполняется сравнительно небольшим набором операций (они составляют *реляционную алгебру* [17]).

Объединение — эта операция, как и две следующих, применима к таблицам, имеющим одну и ту же схему; она выполняет обычное теоретико-множественное объединение нескольких наборов кортежей. Например, объединение нескольких зачетных ведомостей в одну. В случае если какой-либо кортеж содержится в нескольких таблицах, в результирующей таблице он появится только один раз.

Пересечение — теоретико-множественное пересечение двух наборов кортежей.

Вычитание — теоретико-множественная разность двух наборов кортежей.

Произведение — операция, редко используемая в чистом виде. Из двух таблиц составляется таблица, в которой каждый кортеж первой таблицы сцепляется с каждым кортежем второй. Схема результирующей таблицы получается объединением схем таблиц-аргументов, так что ее арность равна сумме арностей аргументов, а количество кортежей равно произведению количеств кортежей у аргументов.

В терминах произведения удобно описывать некоторые другие, более практичные операции. Впрочем, произведение удобно для того, чтобы добавить к таблице один или несколько атрибутов. Например, экзаменационную ведомость полезно расширить, добавив атрибуты НАЗВ-ДИСЦ (название дисциплины) и СЕМ (семестр). После этого расширенные ведомости можно объединить в одну сводную ведомость (а исходные ведомости, как вы понимаете, объединять было бы неправильно).

Выборка — получение из таблицы другой таблицы, в которой оставлены только кортежи, удовлетворяющие данному условию. Например, выборка из экзаменационной ведомости записей с оценкой "удовлетворительно".

Проекция — удаление из таблицы некоторых атрибутов (столбцов). При таком сокращении набора атрибутов некоторые кортежи могут совпасть по оставшимся. Возникшие "дублирования" исключаются.

Расширение — добавление к таблице некоторых атрибутов (столбцов) с постоянными значениями (как в случае расширенной экзаменационной ведомости) или вычисляемыми по значениям других атрибутов (например, по цене и количеству данного товара можно вычислить общую стоимость). Эта удобная операция была добавлена позднее.

Соединение — очень важная операция, которую удобно описать в терминах уже имеющихся операций. Представьте себе, что составлено произведение двух таблиц, а затем у результата оставлены лишь кортежи, в которых одинаковы значения на некоторых атрибутах, появившихся из разных аргументов. С помощью этой операции можно список студентов соединить с расширенными экзаменационными ведомостями и таким образом приписать каждому студенту его оценки.

Существует и более общая схема соединения, в которой приемлемость кортежа из произведения определяется более сложным условием согласованности его частей.

Деление — пусть даны две таблицы, A и B, и схема B входит в схему A (т.е. каждый атрибут B входит в A с таким же доменом). Разность этих двух схем можно представить себе как схему некоторого третьего отношения C. Рассмотрим наибольшее (по множеству кортежей) отношение C, произведение которого на B содержится в A. Вот это отношение и называется результатом деления A на B.

Например, деление сводной экзаменационной ведомости (точнее, ее проекции, в которой отсутствует атрибут ОЦЕНКА) на список экзаменов дает список студентов, сдавших все экзамены.

При использовании базы данных перечисленные операции применяются для формирования новых таблиц. Результатом работы может быть сама таблица или какие-то итоговые суммы, для нахождения которых имеется специальная операция *подведения итогов*. Имеется специальный язык запросов SQL*), на котором можно проектировать создание требуемой таблицы из имеющихся с использованием перечисленных операций.

*) Structured Query Language — язык структурированных запросов; этот язык является мировым стандартом.

Здесь появляются очень интересные задачи *оптимизации исполнения* запроса. Дело в том, что один и тот же запрос может исполняться разными путями, и трудоемкость исполнения сильно зависит от пути. Мы отсылаем читателя к специальным книгам по базам данных.

А нам следует коснуться еще вопросов выбора *схемы базы данных* — совокупности схем таблиц, по которым распределена информация. Здесь специалисты по базам данных накопили богатый опыт, который воплотился в отчетливых общих принципах.

Ключи. Прежде всего, каждый кортеж в каждой таблице должен иметь уникальное значение *ключа*. Мы уже встречались с этим термином в главе 6 (не говоря уже о криптографическом ключе в главе 5), считая, что ключ записи уже определен. В теории реляционных баз данных о ключах говорят больше, речь идет о том, как этот ключ выбирать *).

Пусть K — схема рассматриваемого отношения. Множество $K' \subset K$ называется *потенциальным ключом*, если в таблице не может быть кортежей, совпадающих на K', и никакое собственное подмножество K' этим свойством не обладает. Потенциальный ключ может быть *простым*, состоящим из одного атрибута, или *составным* — из нескольких. Один из потенциальных ключей выбирается в качестве *первичного ключа*, а остальные называются *альтернативными*.

Например, в экзаменационной ведомости и в списке студентов факультета атрибут НОМ-ЗАЧ может быть (простым) ключом. А в сводной ведомости этот атрибут уже не будет уникальным, так как в ней допустимо (даже ожидается) несколько записей об экзаменах, сданных одним и тем же студентом. Здесь естественным ключом будет составной — пара атрибутов (НОМ-ЗАЧ, НАЗВ-ДИСЦ).

Было бы неестественно держать в сводной ведомости слишком много информации о студенте, даже если бы все наши данные состояли из одной этой ведомости. Почему?

Прежде всего, это упрощение зависимости данных. Зависимость выражается в совместных ограничениях на возможные текущие значения, появляющиеся в кортежах. Например, если таблица с информацией о студентах содержит атрибуты НОМ-ЗАЧ, ФИО, то можно сказать, что существует "функциональная зависимость" атрибута ФИО от атрибута НОМ-ЗАЧ: номер зачетки в любом конкретном состоянии данных однозначно определяет фамилию студента. Наличие такой функциональной зависимости может вызывать следующие проблемы:

1) *Избыточность.* Фамилия студента повторяется для каждого студента, записанного в таблицу, хотя номера зачетки достаточно для его идентификации.

*) Дейт [17] считает, что термин *ключ* слишком важен в теории баз данных, чтобы его можно было использовать без уточняющих слов. Но мы иногда будем это делать.

2) *Потенциальная противоречивость.* Вследствие избыточности мы можем сделать ошибку, обновив фамилию студента в одном кортеже и оставив ее неизменной или обновив иначе в других кортежах и отношениях.

3) *Аномалии включения.* Если бы база данных о студентах состояла только из экзаменационных ведомостей, то в базе не было бы данных о студенте, который не сдал еще ни одного экзамена.

Функциональные зависимости. Определим их точнее. Пусть K — схема отношения. Скажем, что два кортежа совпадают по подмножеству $X \subset K$, если значения всех атрибутов из X у них одинаковы.

Возьмем два подмножества атрибутов X и Y. Подмножество Y функционально зависит от X, и это обозначается как $X \to Y$, если в любом текущем значении схемы K любые кортежи, совпадающие по X, совпадают и по Y (подмножество атрибутов ФИО функционально зависит от одноэлементного подмножества НОМ-ЗАЧ). Можно рассмотреть *отношение функциональной зависимости* на множестве всех возможных подмножеств атрибутов данной схемы. Окажется (докажите сами!), что это отношение транзитивно.

Нормальные формы схем отношений. В теории реляционных баз данных определен ряд свойств, которыми могут обладать (и желательно, чтобы обладали) схемы отношений с зависимостями. Принято говорить о "нормальных формах" схем, обладающих теми или иными свойствами. Наиболее важные из них называются "третьей нормальной формой" и "нормальной формой Бойса–Кодда". Эти нормальные формы (НФ) гарантируют, что многих проблем избыточности и аномалий, упомянутых выше, уже не будет. См. об этом подробно, например, в [17, 48]; мы лишь упомянем первые три формы.

Первая нормальная форма. Домен каждого атрибута состоит из неделимых значений, а не из кортежей.

Будем считать (для упрощения определений), что в рассматриваемом отношении нет альтернативных ключей.

Вторая нормальная форма. Отношение находится в первой НФ, и каждый неключевой атрибут неприводимо зависит от первичного ключа. (Это требование означает, что никакая совокупность атрибутов не может функционально зависеть от части первичного ключа. Нормализация схемы заключается в том, что подобная зависимость выделяется в отдельное отношение.)

Третья нормальная форма. Отношение находится во второй НФ, и каждый неключевой атрибут нетранзитивно зависит от первичного ключа. (Все случаи транзитивной зависимости рекомендуется также выносить в отдельные отношения.)

Глава 8

Теория графов

8.1. Определения

Пусть M и N — два конечных множества. Тройка $\langle M, N, T\rangle$, где T — отображение множества N в прямое произведение $M \times M$, называется (конечным) *ориентированным графом*. Элементы множества M называются *вершинами* графа, элементы множества N — *дугами графа*, отображение T сопоставляет каждой дуге $j \in N$ упорядоченную пару вершин, первая из которых называется *началом дуги*, а вторая — *концом дуги*. Будем обозначать начало и конец дуги u через beg u и end u соответственно *). Говорят, что эти вершины и дуга *инцидентны* (присущи) друг другу.

Дуга, у которой начало и конец совпадают, называется *петлей*.

Граф можно изобразить рисунком (само название имеет графическое происхождение), на котором каждая вершина представлена точкой или ее более крупным заменителем, а каждая дуга — линией, соединяющей начало и конец дуги. Стрелка на дуге или рядом показывает направление от начала к концу. Форма линии несущественна, важен только самый факт соединения вершин.

Например, граф с множеством вершин $M = \{A, B, C, D\}$, множеством дуг $N = \{1, 2, 3, 4, 5, 6\}$ и отображением T, задаваемым таблицей, может быть изображен чертежом (рис. 8.1).

*) Существуют и неориентированные графы, у которых направление соединения вершин не определяется. В них соединение называется *ребром*, а вершина — *узлом*. Мы не рассматриваем неориентированных графов только для экономии времени; в тех случаях, когда ориентация дуги несущественна, ее можно игнорировать.

Дуга	*u*	1	2	3	4	5	6
Начало	beg *u*	A	A	A	C	B	D
Конец	end *u*	B	C	D	D	D	B

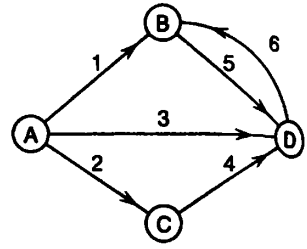


Рис. 8.1. Пример графа

Терминологию теории графов, ее понятия и изобразительные средства удобно использовать для изучения отношений.

Любое двуместное отношение U на множестве A может быть изображено графом $\langle A, U, T\rangle$, для которого отображение T определяется тривиально (каждая дуга графа — это элемент отношения, т. е. упорядоченная пара вершин, и отображение элемента в пару определяет начало и конец дуги). Множество графов, соответствующих отношениям, уже множества всех графов в смысле общего определения, так как в первых не может быть двух различных дуг, у которых совпадают и начало и конец.

Пример. Грамматический разбор предложений, которому нас учили в школе, предполагает построение графа отношений между словами, слова — это вершины графа, а связи между ними — дуги (ср. рис. 8.2). Правда, чувствуются какие-то затруднения (какие?) в рисовании графа для более современной фразы "Мама мыла мышку шампунем и кондиционером в одном флаконе". Похоже, что при грамматическом разборе некоторые словосочетания играют роль атомов и дальше делиться не хотят.

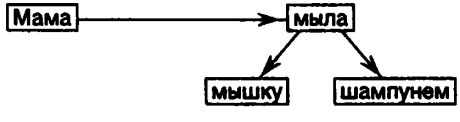


Рис. 8.2. Граф структуры предложения

Посмотрим, как выглядят в терминах графов те свойства отношений, которые мы рассматривали. Рефлексивность означает, что из каждой вершины выходит петля. Симметричность означает, что каждой дуге в графе соответствует встречная дуга, начало которой — это конец данной, а конец — начало данной (как дуги 5 и 6 на рис. 8.1).

Антисимметричность — каждые две различные вершины соединены дугой не более чем в одном направлении.

Описание транзитивности и эквивалентности требует еще нескольких определений. Пусть задан граф $\langle M, N, T\rangle$.

Путем звенности k в этом графе называется пара отображений $A: 1{:}k \to N$ и $V: 0{:}k \to M$, таких что для любого $i \in 1:k$ выполняется $\mathsf{end}\, A(i) = V(i)$ и $\mathsf{beg}\, A(i) = V(i-1)$.

Возможно, что это определение покажется вам странным. На самом деле оно очень удобно. Представьте себе последовательность дуг, у которых начало каждой следующей совпадает с концом предыдущей. Допустите, что в этой последовательности дуги и (или) вершины могут встречаться любое число раз ($A \to C \to D \to B \to D$ на рис. 8.1). Как задать такой объект? Это можно сделать, только связав появления вершин и дуг в последовательности с номерами их появления. Эти номера и дают наши отображения.

Теперь пойдет целая серия определений.

Тройка $\langle M, N', T\rangle$, где N' — подмножество N, называется *частичным графом* графа $\langle M, N, T\rangle$. Здесь, строго говоря, вместо T следовало бы писать T' — ограничение отображения T на множество N'. Замечание относится и к другим определениям, следующим ниже. Мы не станем указывать на изменение отображений, а в дальнейшем вообще перестанем писать в определении графа отображение T, когда будет понятно, о чем идет речь.

Пусть M' — какое-либо подмножество M. Обозначим через $N(M')$ множество всех дуг, у которых и начала и концы принадлежат M'.

Граф $\langle M', N(M'), T\rangle$ называется *подграфом* графа $\langle M, N, T\rangle$. Граф $\langle M', N', T\rangle$, где $N' \subset N(M')$, называется *частичным подграфом* графа $\langle M, N, T\rangle$.

Сейчас мы определим некоторые специальные частичные подграфы. Описываемые объекты на самом деле очень просты, и наши определения могут показаться слишком сложными. Но, к сожалению, более простых определений нам найти не удалось.

Частичный подграф $\langle M', N', T\rangle$ называется *простым путем*, если:

1) число его дуг k на единицу меньше числа вершин;

2) можно так перенумеровать M' числами от 0 до k и N' числами от 1 до k, что для любой дуги $u \in N'$

$$\mathsf{num}(u) = \mathsf{num}(\mathsf{end}\ u) = \mathsf{num}(\mathsf{beg}\ u) + 1.$$

Простой путь (рис. 8.3) — это последовательность дуг, в которой каждая следующая дуга имеет началом конец предыдущей дуги и каждая дуга встречается не более одного раза (если отказаться от этого второго условия и допустить повторы вершин или дуг в пути, то граф называется *путем* — без слова "простой").

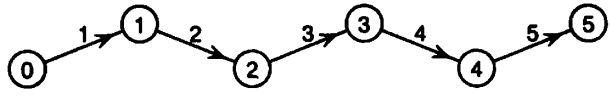


Рис. 8.3. Простой путь

В терминах путей легко объясняется понятие транзитивности: *отношение транзитивно*, если для любого пути в графе, соответствующем этому отношению, найдется дуга, идущая из начала пути в его конец (т. е. если любой путь "дублируется" существующей в графе дугой).

Несколько позднее мы вернемся еще раз к транзитивности, а также посмотрим, как в графах отражается свойство эквивалентности. А сейчас продолжим вводить определения.

Частичный подграф $\langle M', N', T\rangle$ называется *цепью*, если:

1) число его дуг k на единицу меньше числа вершин;

2) можно так перенумеровать M' числами от 0 до k и N' числами от 1 до k, что для любой дуги $u \in N'$

$$\mathsf{num}(u) = \mathsf{num}(\mathsf{end}\ u) = \mathsf{num}(\mathsf{beg}\ u) + 1$$

либо

$$\mathsf{num}(u) = \mathsf{num}(\mathsf{end}\ u) + 1 = \mathsf{num}(\mathsf{beg}\ u).$$

Цепь (рис. 8.4) — это последовательность дуг, в которой каждая дуга имеет общую вершину с предыдущей дугой и со следующей дугой и эти общие вершины различны. Естественно назвать дуги, для которых выполняется первое из соотношений, *положительно ориентированными*, а те, для которых выполняется второе соотношение, — *отрицательно ориентированными* (или просто *положительными*

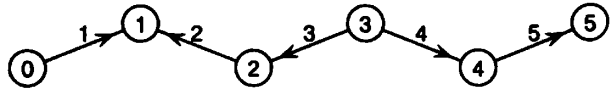


Рис. 8.4. Цепь

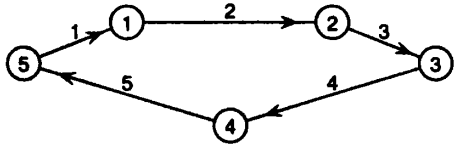


Рис. 8.5. Контур

и *отрицательными*). Легко видеть, что имеется две различные нумерации цепи, и дуги, положительные в одной нумерации, становятся отрицательными в другой.

Частичный подграф $\langle M', N', T\rangle$ называется *контуром*, если:

1) число дуг k равно числу вершин;

2) можно так перенумеровать M' и N' числами от 1 до k, что для любой дуги $u \in N'$

$$\mathsf{num}(u) \underset{\mathsf{mod}\ k}{=} \mathsf{num}(\mathsf{end}\ u) \underset{\mathsf{mod}\ k}{=} \mathsf{num}(\mathsf{beg}\ u) + 1.$$

Контур (рис. 8.5) — это простой путь, в котором начало и конец совпадают (и не зафиксировано начало этой нумерации; всего таких нумераций столько, сколько возможных начал отсчета, т. е. k).

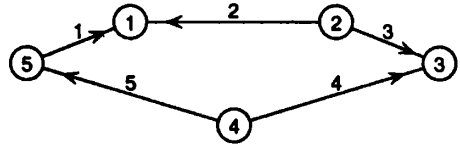


Рис. 8.6. Цикл

Частичный подграф $\langle M', N', T\rangle$ называется *циклом*, если:

1) число дуг k равно числу вершин;

2) можно так перенумеровать M' и N' числами от 1 до k, что для любой дуги $u \in N'$

$$\mathsf{num}(u) \underset{\mathsf{mod}\ k}{=} \mathsf{num}(\mathsf{end}\ u) \underset{\mathsf{mod}\ k}{=} \mathsf{num}(\mathsf{beg}\ u) + 1$$

либо

$$\mathsf{num}(u) \underset{\mathsf{mod}\ k}{=} \mathsf{num}(\mathsf{end}\ u) + 1 \underset{\mathsf{mod}\ k}{=} \mathsf{num}(\mathsf{beg}\ u).$$

Цикл (рис. 8.6) — это цепь, в которой начало и конец совпадают (здесь число нумераций равно $2k$, так как можно выбирать и направление и начало нумерации). Аналогично случаю цепи можно говорить о положительной и отрицательной ориентации дуг при данном направлении обхода цикла.

8.2. Построение транзитивного замыкания графа (отношения)

Уже говорилось, что отношение можно рассматривать как задание упорядочения на множестве M. Это упорядочение может быть частичным, заданным не для всех пар элементов множества. Однако даже частичное упорядочение должно обладать свойством транзитивности. Часто возникает ситуация, когда произвольно заданный набор предшествований требуется доопределить до упорядочения. Это не вполне *частичный порядок*, который мы определили в прошлой главе: у нас могут быть "эквивалентные" элементы множества, а рефлексивность может отсутствовать, — речь идет о произвольном отношении, которое нужно дополнить для придания ему свойства транзитивности.

В терминах графов можно сформулировать задачу так: задан граф, соответствующий отношению P. Требуется найти все пары $(x, y) \in M$, соединенные путем.

Итак, пусть имеется отношение P, описывающее первичные соотношения порядка между элементами множества M. Требуется построить отношение R, называемое *транзитивным замыканием* отношения P и состоящее из всех пар (x, y), для которых найдется последовательность $x = x_0, x_1, \ldots, x_k = y$, где $(x_{i-1}, x_i) \in P$, $i \in 1 : k$.

Алгоритм для решения этой задачи оказывается неожиданно простым; первоначально нужно положить $R = P$, а затем расширять R, анализируя дополнительные возможности. Оказывается, что это решение может быть проведено одним тройным циклом такого вида:

```
R := P;     //  чтобы была виднее подготовка
for (x in M) do
   for (y,z in M) do
     if ( (y,x) in R) & ( (x,z) in R) then
       INCLUDE( (y,z), R)
```

Этот алгоритм называется *алгоритмом Уоршелла*. Вы видите, что внешним циклом перебираются все элементы множества M и для каждого элемента x, если он может использоваться как промежуточный для соединения двух элементов y и z (что определяется просмотром всех возможных пар (y, z)), отношение R расширяется добавлением пары (y, z). Здесь наиболее существенна такая деталь: цикл по промежуточным элементам x должен быть внешним.

Покажем, что этот алгоритм дает решение задачи, т. е. если в исходном отношении существовал путь $x = x_0, x_1, \ldots, x_k = y$ из точки x в точку y, то получающееся после работы алгоритма Уоршелла отношение R будет включать пару (x, y). Действительно, как только параметр внешнего цикла дойдет до вершины, лежащей внутри пути, назовем ее y_1, то появится новая дуга, минующая y_1, и благодаря этому появится путь из x в y, который на одну дугу короче имевшегося. Полный просмотр множества M исключит из пути все такие промежуточные точки. Вместе с тем появление в отношении R пары (x, y), которой не соответствует никакой путь, как легко видеть, невозможно.

Выполнение этого тройного цикла требует $O(|M|^3)$ операций.

Пример. Пусть $M = 1 : 20$ и существует путь

$$7 \to 13 \to 4 \to 19 \to 5 \to 2 \to 16.$$

При $x = 2$ добавится дуга $5 \to 16$, и в графе будет существовать путь

$$7 \to 13 \to 4 \to 19 \to 5 \to 16.$$

При $x = 4$ добавится дуга $13 \to 19$, а при $x = 5$ дуга $19 \to 16$ (так же, как и дуга $19 \to 2$), и мы сократим путь до $7 \to 13 \to 19 \to 16$ (никаких соединений мы не удаляли, и гарантированно появился новый, более короткий путь). После итераций 13 и 19 появится путь $7 \to 16$.

Упражнение 8.1. Постройте пример, показывающий, что приведенный ниже цикл не строит транзитивного замыкания:

```
R := P;
for (x in M) do
  for (y in M) do
    for (z in M) do
      if ( (x,y) in R) &  ( (y,z) in R) then
        INCLUDE( (x,z), R)
      fi
od od od
```

Алгоритм с улучшенной оценкой трудоемкости в $O(|M|^3/\log_2|M|)$ был предложен в статье четырех московских авторов *) [4]. В нескольких словах идея сокращения состоит в том, что сначала метод делится на несколько этапов, почти все из них имеют трудоемкость $O(|M|^2)$, а в единственном более трудоемком этапе удается добиться сокращения трудоемкости за счет того, что множество вершин разбивается на сравнительно небольшие подмножества (не больше $\log_2|M|$ вершин в каждом) и для них формируются вспомогательные справочники, сокращающие трудоемкость.

8.3. Связность. Компоненты связности и сильной связности

Граф $\langle M, N, T\rangle$ называется *связным*, если любые две различные его вершины можно соединить цепью. Любой граф может быть однозначно разделен на максимальные связные подграфы, которые называются его *компонентами связности*. Определение компоненты связности особенно удачно выглядит, если воспользоваться понятием эквивалентности.

Пусть задан граф $\langle M, N, T\rangle$. Определим на множестве M отношение C, относя к нему те пары вершин $(x, y) \in M \times M$, которые могут быть соединены цепью, и все пары (x, x). Это отношение, очевидно, рефлексивно и симметрично.

Лемма. Отношение C транзитивно, т.е. из того, что цепями могут быть соединены вершины пары (x, y) и вершины пары (y, z), следует, что цепью можно соединить и вершину x с вершиной z.

Доказательство. Рассмотрим цепь $x = x_0, \ldots, x_m = y$, соединяющую x и y и перенумерованную от x к y, и цепь $y = y_0, \ldots, y_n = z$, соединяющую y и z и перенумерованную от y к z. Пусть k — наименьший номер вершины из первой цепи, содержащейся во второй цепи (такая, очевидно, существует, так как $x_m = y$ является вершиной из первой цепи, содержащейся во второй цепи). Пусть l — номер этой вершины во второй цепи. Тогда последовательность вершин

$$v_0 = x_0, \ \ldots, \ v_k = x_k = y_l, \ v_{k+1} = y_{l+1}, \ \ldots, \ v_{k+n-l} = y_n = z,$$

"снабженная" соответствующей последовательностью дуг, и образует искомую цепь. □

Следовательно, отображение C — эквивалентность и, значит, задает разбиение множества M на классы эквивалентности. Легко видеть, что

*) В западной литературе он известен как "алгоритм четырех русских".

каждый такой класс M_α задает подграф и этот подграф сам является связным графом. Получающиеся подграфы и называются *компонентами связности* исходного графа.

Пример. Граф, изображенный на рис. 8.7, состоит из трех компонент связности.

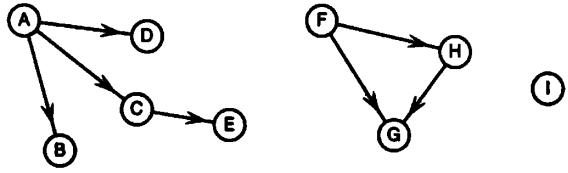


Рис. 8.7. Пример несвязного графа

Реже используется не обладающее транзитивностью свойство "вершины могут быть соединены путем". Однако есть более полезный его вариант "вершины могут быть соединены путем в любом направлении".

Граф $\langle M, N, T\rangle$ называется *сильно связным*, если любые две различные его вершины i_1 и i_2 можно соединить путем с началом i_1 и концом i_2. В любом графе можно однозначно выделить максимальные сильно связные подграфы, которые называются его *компонентами сильной связности.*

Аналогично предыдущему введем отношение

$$R(i_1, i_2) = \text{"существует путь из } i_1 \text{ в } i_2\text{"}.$$

Это отношение очевидно рефлексивно, так как можно считать, что каждая вершина соединена сама с собой. Оно также и транзитивно, так как, если взять путь из i_1 в i_2 и путь из i_2 в i_3, их последовательное соединение образует путь из i_1 в i_3 (цепи так просто не сцеплялись). Обратное отношение R^{-1} также транзитивно, а пересечение $S = R \cap R^{-1}$ и транзитивно и симметрично. Таким образом, мы снова имеем отношение эквивалентности S. Классы эквивалентности в этом отношении и являются компонентами сильной связности.

Построим граф $\langle M_s, N_s\rangle$ *), где элементами M_s являются компоненты сильной связности графа $\langle M, N, T\rangle$, а дугами — те дуги графа $\langle M, N, T\rangle$, концы которых принадлежат разным компонентам. Кроме

*) Соответствующее отображение T строится попутно, его точное описание не принесет нам пользы.

того, отождествим дуги с одинаковыми началом и концом. Такой граф назовем *диаграммой порядка* (в литературе по теории графов он называется также графом Герца или концентрацией) графа $\langle M, N, T \rangle$.

Пример. Для графа, изображенного на рис. 8.8, диаграмма порядка показана на рис. 8.9. Для обозначения ее вершин в каждом классе эквивалентности произвольно выбрано по одному элементу.

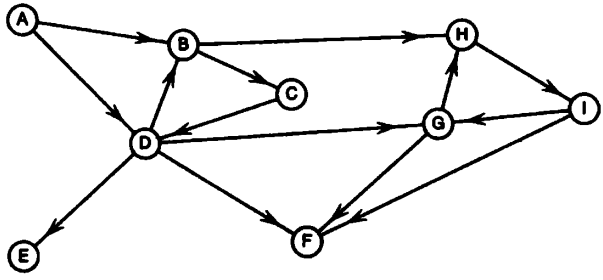


Рис. 8.8. Граф с контурами

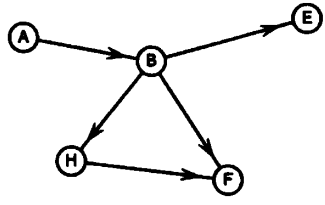


Рис. 8.9. Диаграмма порядка графа, изображенного на рис. 8.8

Теорема. Граф компонент сильной связности (диаграмма порядка) не имеет контуров.

Доказательство. Будем доказывать теорему от противного. Контур в графе $\langle M_s, N_s \rangle$ легко преобразуется в контур в исходном графе $\langle M, N \rangle$. Действительно, каждая дуга j_k этого контура имеет свой прообраз в множестве N. Сами по себе эти дуги не образуют контура в исходном графе, так как у них не обязательно совпадают начала и концы. Но две вершины, лежащие в одной компоненте сильной связности, могут быть соединены путем, и с помощью этих путей все разрывы восполняются. □

Контуры разыскиваются с помощью простой схемы перебора, в которой строится постоянно наращиваемый путь. Он либо замыкается на вершину, уже содержащуюся в нем (и это означает построение контура), либо входит в тупиковую вершину, из которой нет выхода. В этом случае дуга, ведущая в такую вершину, может быть удалена из рассмотрения и из пути *). Построение пути следует вести начиная с каждой из еще не удаленных вершин.

Формальное описание этого алгоритма.

Алгоритм поиска контура

Состояние вычислительного процесса. Граф $\langle M_0, N_0 \rangle$ и простой путь $\langle M', N' \rangle$.

Начальное состояние. $M_0 = M$, $N_0 = N$, путь состоит из любой вершины $i_0 \in M$.

Стандартный шаг. Пусть k — длина пути, i_k — конец пути. Рассмотрим множество дуг из N_0, для которых вершина i_k является началом:

$$N^-(i_k) = \{j \in N_0 \,|\, \mathbf{beg}\ j = i_k\}.$$

Если множество $N^-(i_k)$ не пусто, выберем в нем произвольно дугу, обозначим ее через j_{k+1}, а ее конец через i_{k+1}. Если i_{k+1} содержится в M', то построен контур. Если нет, то увеличим k на единицу, нарастив путь $\langle M', N' \rangle$.

Если же множество $N^-(i_k)$ пусто, объявим вершину i_k тупиковой и удалим ее из M_0, а из N_0 удалим все дуги, для которых i_k является концом. Путь $\langle M', N' \rangle$ при этом следует уменьшить на одну дугу, если $k > 0$, а если $k = 0$, нужно искать очередную вершину, еще остающуюся в множестве M_0.

Алгоритм поиска контура легко модифицируется в алгоритм построения диаграммы порядка.

Модификация относится к работе с вершинами графа. Они заменяются *укрупненными вершинами*, состав которых изменяется в ходе вычислений. Для каждой укрупненной вершины перебираются дуги, выходящие из вершин, составляющих эту укрупненную вершину. Дуги, превратившиеся из-за укрупнения в петли, конечно, игнорируются.

*) Таким образом, над путем осуществляются типичные *стековые* операции — добавление в конец и удаление из конца.

Пример. Рассмотрим граф, изображенный на рис. 8.8. Построение его диаграммы порядка описывается следующей таблицей:

V	v	Дуга	Путь	Событие
A	A	AB	$A \to B$	
B	B	BC	$A \to B \to C$	
C	C	CD	$A \to B \to C \to D$	
D	D	DB	$A \to B \to C \to D \to B$	Контур
BCD	D	DE	$A \to (BCD) \to E$	Тупик
BCD	D	DF	$A \to (BCD) \to F$	Тупик
BCD	D	DG	$A \to (BCD) \to G$	
G	G	GF	$A \to (BCD) \to G \to F$	Здесь были
G	G	GH	$A \to (BCD) \to G \to H$	
H	H	HI	$A \to (BCD) \to G \to H \to I$	
I	I	IF	$A \to (BCD) \to G \to H \to I \to F$	Здесь были
I	I	IG	$A \to (BCD) \to G \to H \to I \to G$	Контур
GHI	I		$A \to (BCD) \to (GHI)$	Тупик
BCD	B	BH	$A \to (BCD) \to (GHI)$	Здесь были
A	D	AD	$A \to (BCD)$	Здесь были

Отметим, что если в приведенном алгоритме последовательно присваивать удаляемым из M_0 вершинам убывающие номера — первой удаляемой вершине номер $m = |M|$, следующей $m - 1$ и т. д., то в случае, когда граф не содержит контуров, все вершины окажутся перенумерованными от 1 до m, и у любой дуги номер ее начала будет меньше номера ее конца. Эта нумерация задает упорядочение вершин, которое будет *усилением* частичного порядка, заданного графом. Такое действие продолжения частичного порядка до полного называется *топологической сортировкой*.

Теперь мы можем вернуться к одной задаче, в решении которой нам поможет алгоритм отыскания контуров.

Пример. Вспомним задачу о цепном коде из упражнения 2.12 на с. 35. Задано число n и требуется построить циклическую строку из 2^n нулей и единиц, в которой все 2^n фрагментов длины n различны. Для иллюстрации будем использовать граф, приведенный на рис. 2.6, где $n = 4$. Построим граф, вершины которого соответствуют всевозможным строкам из нулей и единиц длины $n - 1$. Проведем дугу из вершины i в вершину j, если строка j получается из строки i отбрасыванием символа в начале и приписыванием символа в конце, например таким способом 100 получается из 010. Приписываемый символ назовем *меткой* дуги. Граф изображен на рис. 8.10. Из каждой вершины этого графа выходит ровно две дуги, и число дуг равно 2^n. Будем говорить, что при

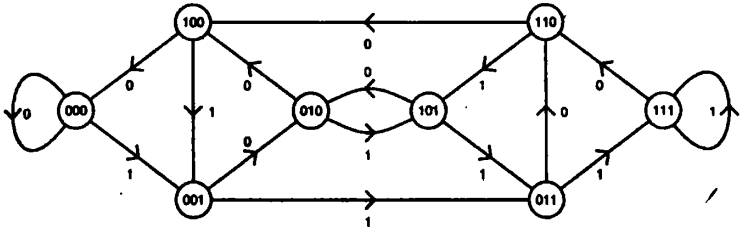


Рис. 8.10. Граф для построения цепного кода

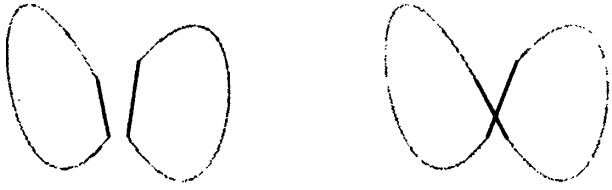


Рис. 8.11. Присоединение контура к замкнутому пути

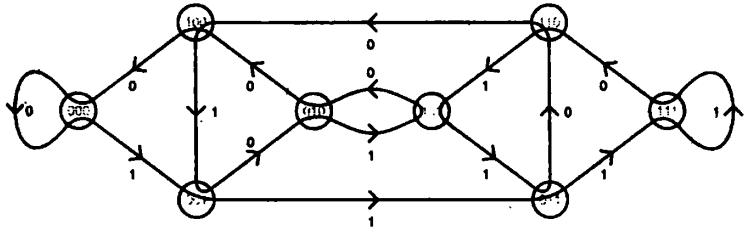


Рис. 8.12. Окончательный замкнутый путь

движении по любому пути в графе *прочитывается* строка, составленная из меток дуг этого пути в порядке их прохождения. Легко видеть, что цепному коду соответствует путь в нашем графе, содержащий каждую дугу ровно по одному разу и замкнутый, т. е. начинающийся и заканчивающийся в одной и той же вершине *). Такой путь можно построить, находя в графе контуры и склеивая их в замкнутый путь так, как показано на рис. 8.11. Окончательный путь показан на рис. 8.12. Этот метод легко программируется [43], он позволяет находить цепные коды для достаточно больших n.

*) Такой путь называется эйлеровским путем. Эйлер рассматривал такие пути в связи с *задачей о кёнигсбергских мостах*: нужно было определить, возможна ли прогулка по Кёнигсбергу, в которой каждый из имевшихся в городе мостов проходится по одному разу.

8.4. Деревья

Термин "дерево" нам уже встречался, когда мы занимались алгоритмами поиска и кодирования и представлением информации. Теперь деревья в несколько более общем определении появятся как особая разновидность графов. Здесь они служат своего рода носителями свойства связности.

Теорема. В связном графе $\langle M, N, T\rangle$ найдется частичный связный граф $\langle M, N', T\rangle$, в котором $|M| - 1 = |N'| = k$ и можно перенумеровать вершины из M числами от 0 до k, а дуги из N' числами от 1 до k таким образом, что для любой дуги $u \in N'$ выполняется соотношение

$$\mathsf{num}(u) = \max\{\mathsf{num}(\mathsf{beg}\ u), \mathsf{num}(\mathsf{end}\ u)\}.$$

Доказательство. Мы используем индукцию и будем строить частичный подграф графа $\langle M', N', T\rangle$, расширяя его множество вершин и множество дуг до тех пор, пока множество M' не совпадет с самим M. При этом на каждом шаге построения число вершин будет на 1 больше числа дуг, частичный подграф будет связным, и на его вершинах и дугах будет определена требуемая нумерация.

Выберем произвольную вершину $i_0 \in M$ и примем первоначально $M' = \{i_0\}$, $N' = \varnothing$, $\mathsf{num}(i_0) = 0$. Первоначально k равно, очевидно, 0.

Предположим (мы имеем базу индукции и делаем индукционный переход), что уже построен частичный подграф $\langle M', N', T\rangle$, обладающий перечисленными свойствами и такой, что $M \setminus M' \neq \varnothing$.

Выберем какую-либо вершину $\bar{i}$, не входящую еще в M'. Так как граф $\langle M, N, T\rangle$ связен, существует цепь, соединяющая i_0 и $\bar{i}$. В этой цепи начальная вершина i_0 принадлежит M', а конечная $\bar{i}$ не принадлежит, и значит, в цепи найдутся две последовательно нумерованные вершины i' и i'', соединенные дугой u', такие что $i' \in M'$, $i'' \notin M'$. Добавим теперь вершину i'' в множество M', а дугу u' в множество N', увеличив число k на 1 и приписав вершине и дуге это новое значение k в качестве их номера в нумерации. Новые M' и N' обладают всеми перечисленными свойствами: дуг на 1 меньше, чем вершин, так как была добавлена одна вершина и одна дуга; граф связен, так как все старые вершины были соединены между собой цепью по предположению, i'' имеет соединение с i', а через нее со всеми остальными вершинами; нумерация определена, и на ней выполнено необходимое соотношение, в том числе и для новой дуги, номер которой совпадает с номером i'' и больше номера i', который был введен на более ранних шагах построения частичного подграфа.

Этот процесс можно продолжать до тех пор, пока множество M' не совпадет с M. В результате получается тот частичный связный граф, существование которого утверждается в теореме. □

Следствие 1. Если $|N| < |M| - 1$, граф не может быть связным.

Следствие совершенно очевидно. Жертвуя точностью ради выразительности, передать его смысл можно просто: в связном графе должно быть достаточно много дуг.

Следствие 2. Если $|N| > |M| - 1$, граф содержит циклы.

Это следствие легко доказывается для связного графа: после построения N' останется хотя бы одна дуга u^*, которая замкнет цепь, соединяющую ее начало и конец в графе $\langle M, N' \rangle$.

Для несвязного графа должна найтись компонента связности, в которой достаточно много дуг. Действительно, если бы в каждой компоненте дуг было бы меньше, чем вершин, то во всем графе число дуг было бы не больше, чем число вершин минус число компонент связности.

Связный граф, в котором число дуг на 1 меньше числа вершин, называется *деревом*. В отличие от тех деревьев, которые мы рассматривали раньше в связи с задачами информационного поиска, в так определенном дереве может не быть *корня* — вершины, из которой идут пути во все остальные вершины. Деревья, обладающие корнем, будут в дальнейшем называться *ориентированными* деревьями.

Упражнение 8.2. Доказать, что в сильно связном графе существует ориентированное дерево с любой наперед выбранной вершиной в качестве корня.

Дерево, являющееся частичным графом связного графа, называется его *остовным деревом* (spanning tree), так что теорему этого параграфа можно назвать *теоремой об остовном дереве*.

Хорошо видна свобода выбора следующей дуги в процессе построения графа $\langle M, N', T \rangle$. Этой свободой можно воспользоваться для каких-либо дополнительных целей, в частности для выбора остовного дерева минимальной длины, о котором говорится в следующей экстремальной задаче.

Задача о кратчайшем остовном дереве. Пусть каждой дуге j графа $\langle M, N, T \rangle$ сопоставлено неотрицательное число $l[j]$, именуемое длиной этой дуги*). Требуется построить такое остовное дерево $\langle M, N', T \rangle$, у которого сумма длин дуг $\sum_{j \in N'} l[j]$ была бы минимальна. ⊠

*) Говорят также о цене дуги, о затратах или стоимости проезда по дуге, а еще о весе дуги. Для графов с заданными весами дуг иногда используется термин *взвешенные графы*.

Мы рассмотрим два способа решения этой задачи. Первый из них точно следует процессу построения графа $\langle M', N', T\rangle$, изложенному в теореме, второй несколько отличается и благодаря этому имеет некоторые вычислительные преимущества. Будет интересно проследить их сходство и различие как в организации вычислений, так и в доказательстве оптимальности получающегося решения.

Первый алгоритм (алгоритм Прима)*). Назовем *границей* множества вершин M' и обозначим через $\partial N(M')$ множество дуг, соединяющих вершину из M' с вершиной из $M \setminus M'$. Как в доказательстве теоремы, выберем произвольно начальную вершину i_0, образуем дерево с множеством вершин $M' = i_0$ и множеством дуг $N' = \varnothing$. В описанном в доказательстве процессе для увеличения множества N' мы (для определенности, на итерации k) выбирали произвольно дугу из $\partial N(M'_{k-1})$. Будем теперь выбирать ее таким образом, чтобы на ней достигался минимум длины дуги:

$$l[j_k] = \min\{l[j] \mid j \in \partial N(M'_{k-1})\}.$$

Этот алгоритм так прост, что можно не выписывать его формально, ограничившись небольшим примером. Чтобы изображение было проще, в качестве графа возьмем фрагмент прямоугольной решетки. Длины дуг написаны около дуг, и длина будет также названием дуги. Элементы M' и N' выделяются жирным шрифтом. Дуги, принадлежащие границе, помечены стрелками, ведущими из M'. Итак, рассмотрим граф, изображенный на рис. 8.13, *а*.

Здесь уже выделены начальная вершина A и граничные дуги 17 и 11. Из этих двух дуг меньшую длину имеет дуга 11, и именно она включается с номером 1 в множество N'. Одновременно в M' включается вершина F. Теперь граница состоит из дуг 17, 19 и 10 (рис. 8.13, *b*).

Включаем в N' дугу 10 (рис. 8.13, *c*).

Теперь пришла очередь дуги 17 (рис. 8.13, *d*).

Дальше следует дуга 19 (при этом дуга 28 выбывает из границы) и т. д. Окончательно получаем граф, изображенный на рис. 8.13, *e*, где номера дуг приведены в соответствии с таблицей:

Номер	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Вершина	A	F	K	B	G	C	H	I	D	E	L	M	N	P	J
Дуга		11	10	17	19	23	21	15	11	14	26	20	18	21	41

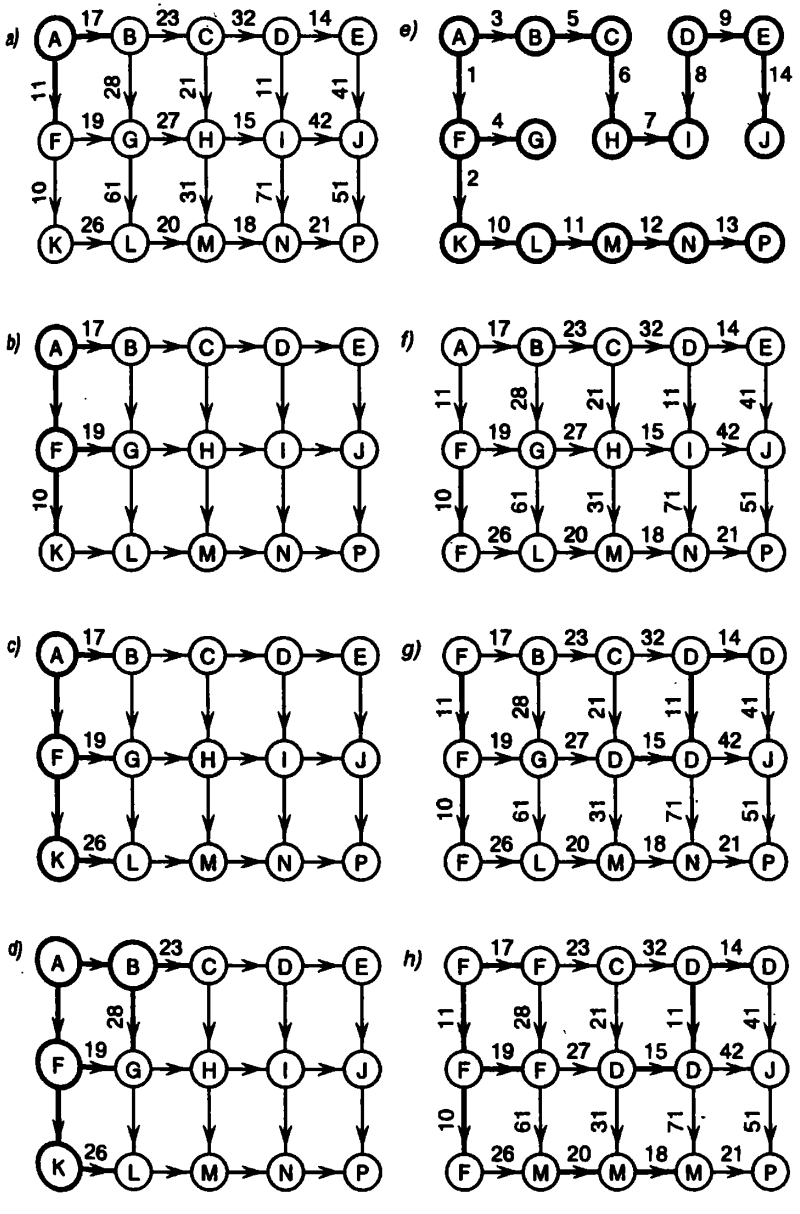


Рис. 8.13. Методы Прима и Краскала

Теорема. Остовное дерево, получаемое в результате работы описанного алгоритма, имеет наименьшую суммарную длину дуг.

Доказательство. Пусть $\langle M, N^* \rangle$ — дерево, полученное по алгоритму Прима. Покажем, что, каково бы ни было остовное дерево $\langle M, N'' \rangle$, его можно так изменить, что новое дерево будет больше похоже на $\langle M, N^* \rangle$, а сумма длин дуг разве лишь уменьшится.

Что значит "больше похоже"? Здесь нужно ввести количественное сравнение. Обозначим через $\langle M_k, N_k \rangle$ частичный подграф, полученный в алгоритме на k-м шаге, $k = 0, \ldots, |M| - 1$. Введем теперь числовую характеристику множества дуг $d(N'') = \max\{k \mid N_k \subset N''\}$. Эта характеристика дает номер последнего шага, на котором множество N^* развивалось как подмножество N'' (а дальше стало отличаться). Чем больше $d(N'')$, тем больше множество N'' похоже на N^*.

Итак, возьмем граф $\langle M, N'' \rangle$, вычислим для него $s = d(N'') + 1$, тогда $j_k \in N''$ для $k < s$ и $j_s \notin N''$. Добавим дугу j_s к множеству N''; вместе с цепью, соединяющей beg j_s и end j_s, она образует некий цикл. Отметим, что одна из этих двух вершин имеет номер s, а другая — номер, меньший s. В цепи, соединяющей эти две вершины, найдется пара соседних вершин, одна из которых имеет номер меньше s, а другая — больший или равный s. Назовем их i' и i''. Соединяющая их дуга j' принадлежит множеству $\partial N(M_s)$ — границе s-го шага. Следовательно, по выбору j_s как дуги минимальной длины мы должны иметь $l[j_s] \le l[j']$. Замена дуги j' дугой j_s в множестве N'' разве лишь уменьшит суммарную длину дуг. Покажем, что такая замена оставит граф $\langle M, N'' \rangle$ деревом и увеличит значение $d(N'')$.

Действительно, увеличение $d(N'')$ следует из того, что теперь множество N_s полностью входит в N''. Что же касается дерева, то при замене дуги количество дуг сохранилось и осталось доказать связность получившегося графа. Достаточно показать, что каждая вершина соединена цепью, например с beg j_s.

Цепи, не содержащие j', сохраняются неизменными.

Цепь, идущая через j', проходит через вершины i' и i'' (для определенности: доходит до i', проходит через j' и далее доходит до beg j_s).

Цепь от i' до beg j_s можно теперь заменить цепью, идущей от i' до end j_s, а затем дугой j_s.

Значит, пока множество N'' не совпадает с N^*, его можно сделать более похожим на N^* с уменьшением (невозрастанием) суммарной длины дуг. В результате конечного числа таких улучшений мы придем к $N'' = N^*$. □

Этот простой алгоритм неудобен тем, что на каждом шаге процесса приходится пересчитывать граничное множество дуг. Оказывается, этой неприятной операции можно просто избежать.

Второй алгоритм (алгоритм Краскала*)). В этом алгоритме строится последовательность частичных графов $\langle M, N_k \rangle$. Начальное множество N_0 принимается пустым, так что граф $\langle M, N_0 \rangle$ состоит из $|M|$ изолированных вершин. На каждом шаге k мы имеем граф, состоящий из нескольких компонент связности, каждая компонента — это дерево. Возьмем дугу j_k, начало и конец которой принадлежат разным компонентам связности, и составим N_k из N_{k-1} и этой дуги. Процесс продолжается до тех пор, пока число компонент связности не будет равно 1.

Осталось уточнить порядок выбора дуг j_k. Упорядочим множество дуг по возрастанию длин и будем просматривать все дуги, отбрасывая те из них, которые соединяют вершины из одной компоненты связности.

Рассмотрим действие этого алгоритма на том же примере. Первой будет выбрана дуга 2 (нумерацию дуг мы берем с рис. 8.13,*e*, а названия вершин будем менять так, чтобы были видны вершины одной компоненты) (рис. 8.13,*f*).

Затем последовательно выбираем дуги 1, 8, 9, 7 (рис. 8.13,*g*).

Далее — дуги 3, 12, 4, 11 (рис. 8.13,*h*).

И наконец — дуги 13, 5, 10, 14.

Опишем теперь этот процесс более формально, используя, насколько возможно, конструкции алгоритмических языков:

```
N' := EmptySet;  nComponents := nVert;  nextArc := 1;
while (nComponents > 1) do
  if comp( BEG(nextArc) ) <> comp( END(nextArc) ) then
    N' := N' JOIN (nextArc); Dec(nComponents);
    JoinComponents( BEG(nextArc), END(nextArc) )
  fi;
  nextArc +:= 1
od;
```

Для определения компоненты, которой принадлежит данная вершина, удобно иметь специальный массив, а для просмотра компонент и их соединения — хранить компоненты в виде цепных списков. Процесс соединения компонент легко организовать так, чтобы меньшая по размеру компонента присоединялась к большей: немного непривычно для

*) Joseph B. Kruskal [72].

тех, кто уже научился цепным спискам на обычных задачах, так как нужно присоединять меньший список к большему сразу после головы большего списка.

При такой организации данных трудоемкость присоединения пропорциональна длине меньшего списка, и справедлива следующая теорема.

Теорема. Трудоемкость описанной выше реализации алгоритма Краскала имеет порядок $|N| + |M| \log_2 |M|$.

Доказательство. Действительно, основная часть алгоритма состоит из просмотра дуг, и трудоемкость просмотра пропорциональна числу дуг и действий, необходимых для соединения компонент. Трудоемкость соединений считается "коллективно": участвуя в соединениях, каждая из вершин проходит через несколько множеств, размер которых увеличивается каждый раз не менее чем вдвое. Поэтому число таких изменений не может быть больше, чем $\log_2 |M|$. □

При оценке полной трудоемкости построения кратчайшего дерева нужно к найденной оценке добавить еще порядка $|N| \log_2 |N|$ операций на упорядочение дуг по возрастанию длин. Так как для связного графа имеет место неравенство $|N| \geq |M| - 1$, то окончательная оценка трудоемкости $O(|N| \log_2 |N|)$.

Теорема. Если дуги в описанном алгоритме просматриваются в порядке возрастания длин, дерево $\langle M, N^* \rangle$, получающееся в результате работы алгоритма, имеет минимальную длину.

Доказательство. Принцип доказательства такой же, как в первом алгоритме, — мы доказываем, что любое дерево может быть сделано больше похожим на $\langle M, N^* \rangle$ без увеличения суммарной длины входящих в него дуг. Отличие в небольших деталях: вместо нумерации из теоремы об остовном дереве должна использоваться нумерация, возникающая при упорядочении дуг.

Пусть $\langle M, N'' \rangle$ — некоторое дерево и

$$r(N'') = \min \{\mathrm{num}(j) \mid j \in N^* \setminus N''\},$$

— первая дуга из N^*, не включенная в N''. Положим $s = r(N'')$ и добавим дугу j_s к множеству N''. Это добавление порождает цикл, в котором должна быть дуга с номером, большим s (иначе начало и конец дуги j_s лежали бы в одной компоненте связности перед s-м шагом алгоритма). Замена этой дуги дугой j_s и дает необходимое изменение (длина не растет, $r(N'')$ растет, граф $\langle M, N'' \rangle$ остается деревом). □

Деревья используются в разнообразных ситуациях, поэтому полезно рассмотреть и другие их свойства. В книге К. Бержа [8], которой мы во многом следуем, эти свойства сформулированы в виде теоремы.

Теорема. Следующие шесть определений дерева эквивалентны:

1) связный граф без циклов;

2) связный граф, в котором дуг на 1 меньше, чем вершин;

3) граф без циклов, в котором дуг на 1 меньше, чем вершин;

4) минимальный связный граф, т. е. граф, который при удалении любой дуги перестает быть связным;

5) максимальный граф без циклов, т. е. такой, в котором добавление любой новой дуги порождает цикл;

6) граф, в котором любые две вершины соединены цепью и притом только одной.

Доказательство. Рассмотрим граф, в котором пункты теоремы 1)–6) будут вершинами. Доказав, например, что из 1) следует 5), мы проведем в графе дугу из 1) в 5). Очевидно, нам достаточно довести доказательство теоремы до момента, когда найдется путь из любой вершины графа в любую другую. Удается построить просто контур, на котором лежат все вершины.

1)→ 5). Докажем, что если граф без циклов связен, то это максимальный граф без циклов. Действительно, при добавлении к графу любой дуги замыкается цепь, соединяющая ее начало и конец, и получается цикл.

5)→ 6). Если при добавлении дуги появляется цикл, значит, эта новая дуга замкнула цепь, соединяющую ее начало и конец. Значит, любые две вершины соединены цепью. Если бы какие-либо вершины были соединены двумя цепями, из них было бы легко построить цикл.

6)→ 4). То, что граф связен, очевидно. Так как для любых двух вершин соединяющая их цепь единственна, для начала и конца любой дуги этот путь — именно сама эта дуга. Ее удаление разрывает цепь между ними.

4)→ 2). Нужно доказать, что, если граф минимально связный, в нем число дуг n на 1 меньше, чем число вершин m. Воспользуемся следствиями из теоремы об остовном дереве. Очевидно, для связности нужно требовать выполнения неравенства $n \geq m - 1$. Но при $n > m - 1$

в графе должна найтись дуга, не входящая в остовное дерево, удаление которой не нарушает связности графа.

2)→3). Доказываем, что если $n = m - 1$, то связный граф не имеет циклов. Воспользуемся нумерацией, о которой говорится в теореме об остовном дереве. Ведь все вершины и дуги имеют номера. Пусть в графе есть цикл. Возьмем в нем вершину с наибольшим номером. Она инцидентна двум дугам, для каждой из которых этот номер — максимум из номера начала и номера конца и, следовательно, равен номеру дуги. Но две дуги иметь один и тот же номер не могут.

3)→1). Если в графе без циклов $n = m - 1$, то он связен. Действительно, если он не связен, в нем найдется компонента связности k, в которой $n_k > m_k - 1$, и значит, есть цикл. □

8.5. Применения деревьев

Понятие дерева очень полезно. Попробуем перечислить те ситуации, в которых деревья нам уже встречались, и добавим некоторые новые.

8.5.1. Иерархические схемы

Очень часто деревья появляются в связи с *иерархическим порядком*. Рассматривая множество A, на котором задан иерархический порядок, мы можем сопоставить ему граф с множеством вершин A и дугами, ведущими в каждую вершину a, кроме максимального элемента, из вершины $p(a)$.

Вспомнив примеры из упражнений на с. 187, мы сразу увидим, что, скажем, кодовое дерево для префиксного кода и дерево строк в алгоритме Зива–Лемпеля являются ориентированными деревьями.

Можно упомянуть также известное *дерево каталогов в операционной системе MS-DOS*[*]. Каждый каталог, кроме корневого, имеет непосредственного предшественника. Все вершины дерева — это каталоги, каждый из них имеет метку — название каталога. Используемое понятие *пути* до файла вполне соответствует пути в дереве от корня

[*] Структура каталогов в операционных системах типа UNIX выглядит более сложно, в ней связи между каталогами не древовидны и допустим произвольный граф без контуров.

до вершины: путем до файла является конкатенация меток вершин на пути от корневой вершины до требуемой.

Ориентированными деревьями являются и биномиальное дерево, рассмотренное на с. 174, и фибоначчиево дерево, упомянутое там же.

В машиностроительном проектировании используются *деревья сборки*, описывающие строение сложных агрегатов. Мы можем представить это приложение таким образом. Задавая строение сложного агрегата, проектировщик перечисляет его составные части (и их кратности) и тем самым составляет *выходящую звезду* для корневой вершины графа (это перечисление оформляется чертежом, который называется *сборкой*). Для каждой из частей делается то же самое, пока не доходят до изготовляемых деталей.

Ориентированные деревья, изображающие последовательные измельчения разбиений множества, характерны для задач поиска, но встречаются и в систематике. В качестве примера можно назвать схемы библиотечных классификаций или гигантскую систему биологической систематизации. Дальше, когда мы будем рассматривать переборные методы решения дискретных экстремальных задач, нам встретится дерево вариантов или частичных решений, которое представляет собой развивающееся в ходе работы алгоритма дерево из подмножеств множества решений.

8.5.2. Представление дерева в компьютере

Компьютерные представления деревьев различны. Вспомним способ, используемый в иерархической сортировке, — соединения между вершинами благодаря удачной нумерации просто вычисляются. Но основных представлений два.

В том случае, когда число непосредственных потомков вершины ограничено (например, в двоичных деревьях и в B-деревьях), информационная структура, соответствующая вершине, резервирует место для всех нужных ссылок на потомков, а если требуется, то и для ссылки на предка.

Если же число непосредственных потомков может быть произвольным, то для перечисления потомков естественно создать и использовать цепной список. В этом случае для каждой вершины нужно хранить три ссылки: на *отца*, т. е. на ближайшего предка, на *старшего сына*, т. е. на голову цепного списка потомков, и на *младшего брата*, т. е. на вершину, следующую за данной в цепном списке потомков их общего предка.

Иногда к дереву добавляют дополнительные связи между вершинами. Такое дерево называется *прошитым* (threaded tree).

8.5.3. Обходы и нумерации деревьев

Вершины дерева можно нумеровать, и эта нумерация равнозначна порядку их *обхода*. Например, нумерация, используемая в сортировке Heapsort, соответствует обходу дерева по этажам: сначала вдоль самого верхнего этажа, затем по второму, по третьему и т. д.

Обходы, соблюдающие структуру дерева, различаются по тому, в каком порядке просматривается вершина и ее потомки. Будем сейчас рассматривать двоичные деревья. У каждой вершины возможны два потомка, которых можно считать корнями некоторых поддеревьев, возможно, пустых. Назовем эти поддеревья *левым* и *правым*.

Нумерация, при которой каждая вершина получает номер до своих потомков, а дальше нумеруются сначала потомки из левого поддерева, а затем из правого, называется *нисходящей* нумерацией. Если нумеруются сначала потомки (также из левого, а затем из правого поддерева), а только потом корень, то нумерация называется *восходящей*. Если нумеруются потомки из левого поддерева, затем корень, а затем потомки из правого дерева, нумерация называется *фронтальной* *).

Принципы восходящей и нисходящей нумерации очевидным образом переносятся на деревья с произвольным числом потомков.

8.5.4. Суффиксные деревья

Суффиксные деревья, которые были введены на с. 103, появились сравнительно недавно, но в книге Гасфилда [13] им посвящено несколько глав. Кроме описанных ранее можно рассматривать суффиксные деревья для двух и более строк и специальными терминальными символами определять, к какой строке относится данный суффикс. Затраты на построение такого общего дерева линейно зависят от суммарной длины строк.

Вот одно из приложений суффиксного дерева для двух строк.

Упражнение 8.3. Пусть даны две строки a и b. Предложите алгоритм, который, используя построенное суффиксное дерево для a и b и восходящий порядок обхода суффиксного дерева, находит их наибольшую общую подстро-

*) Английские термины, соответственно, preorder, postorder и inorder. В книге [23] использованы термины *прямой*, *обратный* и *центрированный* порядок обхода.

ку (например, для строк *фанфарон* и *сафари* это будет строка *фар*) за время, пропорциональное суммарной длине строк.

8.5.5. Неориентированные деревья

Неориентированные деревья используются, когда направление связей несущественно. Таковы, например, транспортные и многие коммуникационные сети. В приложении на с. 318 приведен пример, в котором рассматривается граф процессоров вычислительного комплекса и для организации связи этих процессов строится неориентированное дерево.

8.6. Матрица инциденций и линейные системы

Информацию о графе можно задавать различными способами (например, списком дуг, в котором каждой дуге сопоставляются ее начало и конец). Особое место в способах задания графа занимают *матричные* формы задания, и из нескольких возможных форм наиболее известны *матрица инциденций* и *матрица смежности*. Начнем со второй из них, чтобы в дальнейшем уже не отвлекаться от более важной матрицы инциденций. Здесь и в дальнейшем нам будет удобно, работая с матрицами, указывать в квадратных скобках множества индексов строк и столбцов.

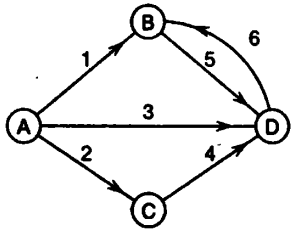
Матрицей смежности графа $\langle M, N\rangle$ называется квадратная матрица $r[M, M]$ (значит, индексами ее строк и столбцов являются вершины графа), в которой значением каждого элемента $r[i, k]$, $i, k \in M$; является число дуг с началом i и концом k:

$$r[i, k] = |\{j \mid j \in N, \mathbf{beg}\ j = i, \mathbf{end}\ j = k\}|.$$

Упражнение 8.4. Докажите, что элементы k-й степени матрицы $r[M, M]$ задают количества k-звенных путей между вершинами графа.

Обратимся теперь к матрице инциденций.

Пусть граф $\langle M, N\rangle$ не имеет петель. В этом случае ему можно сопоставить матрицу, в которой вершинам соответствуют строки, а дугам — столбцы. В каждом столбце j этой матрицы всего два элемента отличны от нуля: в строке $\mathbf{beg}\ j$, соответствующей началу дуги (какой дуги? соответствующей этому столбцу j), стоит 1, в строке $\mathbf{end}\ j$ стоит -1. Такая матрица и называется *матрицей инциденций* графа.



	1	2	3	4	5	6
A	1	1	1	0	0	0
B	−1	0	0	0	1	−1
C	0	−1	0	1	0	0
D	0	0	−1	−1	−1	1

Рис. 8.14. Граф и его матрица инциденций

Например, для встречавшегося нам уже графа матрица инциденций представлена на рис. 8.14.

Так как сумма элементов каждого столбца равна нулю, ранг матрицы инциденций всегда меньше, чем число ее строк.

Теорема. Ранг матрицы инциденций связного графа $\langle M, N\rangle$ равен $|M| - 1$.

Доказательство. Обозначим матрицу инциденций графа через $a[M, N]$. Так как граф связен, существует остовное дерево $\langle M, N'\rangle$, в котором $|N'| = |M| - 1$. Поскольку

$$|M| > \text{rank}\,(a[M, N]) \geq \text{rank}\,(a[M, N']),$$

достаточно показать, что $\text{rank}\,(a[M, N']) = |N'|$.

Воспользуемся нумерацией, о которой шла речь в теореме об остовном дереве, и расположим строки и столбцы матрицы $a[M, N']$ по номерам строк и столбцов. Мы увидим, что свойство нумерации преобразуется в следующее свойство матрицы: в любом i-м столбце один ненулевой элемент расположен в строке i, а другой — где-то выше. Таким образом, если отбросить нулевую строку этой матрицы, то останется верхнетреугольная матрица с ненулевой диагональю. Определитель этой матрицы равен по абсолютной величине 1, значит, матрица неособенная, верхняя и нижняя границы для ранга совпадают, и теорема доказана. □

Следствие. Ранг матрицы инциденций произвольного графа равен разности числа вершин и числа компонент связности.

Действительно, в случае несвязного графа матрица инциденций имеет блочную структуру, и ее ранг равен сумме рангов блоков. Это и вызывает для всей матрицы потерю ранга на число компонент связности.

Теорема. В матрице инциденций значение любого минора равно 0, 1 или -1.

Доказательство. Утверждение очевидно для миноров первого порядка — это значения элементов матрицы. Таким образом, мы имеем базу для доказательства по индукции. Предположим, что утверждение уже доказано для миноров $k-1$ порядка, и докажем для порядка k.

Итак, рассмотрим подматрицу $a[M', N']$ порядка k. Если в каждом столбце этой подматрицы число ненулевых элементов равно 2, ее определитель равен 0, так как в этом случае сумма по каждому столбцу равна 0 и строки подматрицы линейно зависимы. Предположим поэтому, что найдется столбец, в котором один ненулевой элемент или все элементы — нули (этот второй случай неинтересен). Разложим определитель матрицы по этому столбцу. С точностью до знака он будет равен тому минору, который дополняет единственный ненулевой элемент (1 или -1), и ссылка на индукционное предположение завершает доказательство. □

Упражнение 8.5. Пусть в матрице $d[K, L]$ все элементы равны 0, 1 или -1, в каждом столбце ровно два ненулевых элемента и существует такое разбиение множества K на два множества $K = K_1 \cup K_2$, что в столбце с ненулевыми элементами разных знаков оба элемента (строки, где они стоят) принадлежат одному элементу разбиения, а при одинаковых знаках — разным. Доказать, что у такой матрицы все миноры равны 0, 1 или -1.

Теперь рассмотрим систему линейных уравнений с матрицей инциденций и обсудим особенности решения таких систем. Эти системы часто встречаются в приложениях, там, где речь идет о потоках в сетях.

Потоком называется совокупность чисел, заданных на дугах графа (или, как часто говорят, *участках сети*), со специфическими условиями баланса в вершинах (*узлах сети*).

Условие баланса, известное как первое правило Кирхгофа [*)], словесно выражается так: поток, поступающий в каждую вершину i, должен равняться потоку, выходящему из этой вершины. Каждый из этих двух потоков в свою очередь мы должны разделить на поток по сети и внешний поток.

Что касается внешнего потока, то можно вычесть выходящий поток из входящего (экспорт из импорта) и получить, так сказать, *чистый*

[*)] Gustav-Robert Kirchhoff (1824–1887), немецкий математик и физик, член-корреспондент Петербургской академии наук.

импорт, который может быть любого знака, так как отрицательный импорт означает просто экспорт. Обозначим чистый импорт в вершине i через $b[i]$. Аналогично, можно найти чистый экспорт $\beta[i]$ из вершины i по дугам графа, который будет равен разности экспорта и импорта по дугам графа. Здесь под экспортом следует понимать сумму потоков по выходящим дугам, а под импортом — сумму потоков по входящим. Обозначая через $x[j]$ поток по дуге j, получаем

$$\beta[i] = \sum_{\text{beg}j = i} x[j] - \sum_{\text{end}j = i} x[j] = \sum_{j \in N} a[i, j] \times x[j] = a[i, N] \times x[N].$$

Приравнивая $\beta[i]$ и $b[i]$ и записывая эти условия для всех вершин графа, мы и получаем систему

$$a[M, N] \times x[N] = b[M],$$

где, как мы уже видели, $a[M, N]$ — матрица инциденций рассматриваемого графа $\langle M, N \rangle$.

При анализе этой системы и методов ее решения все время будут сочетаться сведения, которые известны из общей теории алгебраических систем, и специфические особенности матрицы, связанные с ее "графским происхождением".

Начнем наш анализ с простого вопроса. Отметим, что поскольку матрица инциденций несвязного графа распадается на несколько независимых блоков, соответствующих его компонентам связности, и система распадается на подсистемы для этих блоков, то можно сразу предполагать, что граф связен.

Обычное необходимое и достаточное условие разрешимости системы — добавление к матрице столбца правых частей не увеличивает ее ранга — в рассматриваемом случае означает, что столбец $b[M]$ подчиняется той же зависимости, что и столбцы матрицы, именно

$$\sum_{i \in M} b[i] = 0.$$

Займемся изучением вида общего решения нашей системы. Как известно, общее решение неоднородной линейной системы может рассматриваться как сумма частного решения неоднородной системы и общего решения однородной. И для того и для другого слагаемого мы сейчас воспользуемся известной нам спецификой системы.

При рассмотрении частных решений удобно часть переменных заранее приравнять нулю и рассматривать систему только для оставшихся переменных. Выберем множество оставшихся переменных N' таким образом, чтобы граф $\langle M, N'\rangle$ был деревом. Как уже отмечалось, при соответствующей нумерации вершин и дуг дерева (строк и столбцов матрицы) матрица $a[M, N']$ становится треугольной. Следовательно, в последнем уравнении системы $a[M, N'] \times x[N'] = b[M]$ участвует только одна переменная. Найдя ее и подставив в какое-то из предыдущих уравнений, мы вновь получим треугольную матрицу, с которой можно поступить точно так же, и т. д.

При рассмотрении однородной системы нас больше будет интересовать сама структура решения. Из предыдущего ясно, что если граф не имеет циклов (если он — дерево), то однородная система имеет только тривиальное решение. Введем новое понятие.

Пусть $\langle M_c, N_c\rangle$ — некоторый цикл в графе $\langle M, N\rangle$. Выберем какую-либо ориентацию этого цикла (вспомним, что ориентация определяется, например, выбором нумерации цикла); в соответствии с ней все дуги цикла разделятся на положительные и отрицательные. Вектор $z[N]$, компоненты которого, соответствующие положительным дугам цикла, равны 1, отрицательным дугам — -1, а остальные компоненты равны 0, называется *циклическим вектором*, соответствующим циклу $\langle M_c, N_c\rangle$.

Лемма. Циклический вектор $z[N]$, соответствующий любому циклу $\langle M_c, N_c\rangle$ графа $\langle M, N\rangle$, является решением системы уравнений

$$a[M, N] \times x[N] = \mathbf{0}[M].$$

Доказательство. В этой системе нам достаточно рассматривать только множество индексов N_c, так как все остальные переменные равны нулю. А поскольку столбцы, соответствующие N_c, имеют ненулевые элементы только в строках с индексами M_c, нам достаточно рассматривать систему

$$a[M_c, N_c] \times x[N_c] = \mathbf{0}[M_c].$$

Пусть $i \in M_c$. В строке $a[i, N_c]$ есть только два ненулевых элемента. Эти элементы могут быть разных знаков, в случае если одна дуга цикла входит в вершину i, а другая выходит, или одного знака, если они обе входят или обе выходят. В первом случае дуги имеют одинаковую ориентацию, во втором — разную; и в любом случае это соотношение знаков дает нулевое скалярное произведение строки на вектор. □

Теорема (Пуанкаре–Веблен–Александер*). Любое решение $x[N]$ системы

$$a[M, N] \times x[N] = \mathbf{0}[M]$$

является линейной комбинацией циклических векторов.

Доказательство. Пусть $x[N]$ — решение системы. Покажем, что его можно представить в виде суммы циклического вектора, умноженного на некоторый коэффициент, и другого решения с меньшим числом ненулевых компонент. Пусть N_a — множество дуг, соответствующих ненулевым компонентам вектора $x[N]$. Граф $\langle M, N_a \rangle$ содержит циклы, как это отмечалось выше. Выберем какой-либо цикл $\langle M_c, N_c \rangle$ и сопоставим ему циклический вектор $z[N]$. Пусть $j_c \in N_c$ — какая-либо дуга цикла; не умаляя общности, можем считать ее положительно ориентированной. Представим вектор $x[N]$ в виде

$$x[N] = x[j_c] \times z[N] + x'[N],$$

где вектор $x'[N]$, равный $x[N] - x[j_c] \times z[N]$, очевидно, также является решением системы, и в нем по крайней мере на одну ненулевую компоненту меньше. Далее этот процесс применяется к $x'[N]$ и всем последующим векторам до полного исчерпания множества N_a. □

Теорема (Кирхгоф). Пусть $\langle M, N \rangle$ — граф и $\langle M, N' \rangle$ — остовное дерево. Любое решение $x[N]$ системы

$$a[M, N] \times x[N] = \mathbf{0}[M]$$

однозначно представимо как линейная комбинация циклических векторов, соответствующих циклам, которые возникают при добавлении в множество N' дуг из $N \setminus N'$.

Доказательство. Получить такое представление легко. Каждой дуге $j \in N \setminus N'$ соответствует цикл. Определим соответствующий циклический вектор $z_j[N]$ таким образом, чтобы дуга j была положительной, и рассмотрим вектор

$$y[N] = x[N] - \sum_j x[j] \times z_j[N].$$

Он также является решением системы. Покажем, что это нулевой вектор. Действительно, компоненты $y[N \setminus N']$ все равны нулю, а $y[N']$

*) Jules-Henri Poincaré (1854–1912), французский математик, иностранный почетный член-корреспондент Петербургской академии наук; Oswald Veblen (1880–1960) и James Wandell Alexander (1888–1971), американские математики.

является решением однородной системы для дерева и поэтому тоже не может иметь ненулевых компонент. Доказательство единственности представления оставляется читателю. □

8.7. Задача о кратчайшем пути и ее варианты

Теперь рассмотрим другой подход к решению экстремальных задач на графах. Наиболее отчетливо этот подход был развит Ричардом Беллманом *), который предложил название *динамическое программирование*. Эта идея нам уже несколько раз встречалась, и мы будем ее обсуждать еще и в общей постановке, а сейчас нам встретится опять очень важный частный случай.

Рассмотрим следующую экстремальную задачу, которая встречается во многих прикладных вопросах. Это задача о поиске в заданном графе пути, соединяющего две заданные вершины и доставляющего минимум или максимум некоторой аддитивной функции, определенной на путях. Чаще всего эта функция трактуется как длина пути, и задача называется *задачей о кратчайшем пути*, хотя имеются и некоторые другие ее постановки.

Задача о кратчайшем пути. Итак, пусть задан граф $\langle M, N\rangle$ и каждой его дуге u сопоставлено положительное число $l[u]$. Для двух заданных вершин i^- и i^+ требуется найти путь минимальной длины с началом i^- и концом i^+. ⊠

Как и в предыдущих случаях, целесообразно рассматривать эту задачу в параметрической форме — решать ее сразу для всех возможных начал i^-, или для всех возможных концов i^+, или и то и другое сразу. Решение этой задачи для одних i^- и i^+ может пригодиться как "кирпичик" для других.

Это хорошо видно на том варианте задачи, когда ищутся *кратчайшие пути от каждой вершины до каждой*. С этого варианта задачи мы и начнем.

При построении матрицы кратчайших расстояний между всеми вершинами графа воспользуемся следующей идеей: сначала строится матрица $v[M, M]$ возможных одношаговых переходов из каждой вершины в каждую:

$$v[i, k] = \min\{l[u] \mid \mathsf{beg}\ u = i, \mathsf{end}\ u = k\}$$

*) Richard Ernest Bellman (1920–1984), американский математик, автор динамического программирования, многие его книги переведены на русский язык.

(минимум по пустому множеству по определению имеет бесконечное значение), а затем эта матрица пересчитывается тройным циклом по множеству вершин, причем внешний цикл — по промежуточной вершине:

```
for (i in M) do
   for (i1,i2 in M) do
     if (v[i1,i2] > v[i1,i] + v[i,i2]) then
       v[i1,i2] := v[i1,i]+v[i,i2]
     fi
od od
```

Смысл алгоритма*) в том, что когда при фиксированном i перебираются все возможные пары $i1$ и $i2$, то происходит сравнение лучшего из имеющихся к данному моменту путей от $i1$ до $i2$ с путем, составленным из лучшего пути от $i1$ до i и лучшего пути от i до $i2$, и в результате этого сравнения может быть улучшен путь от $i1$ до $i2$.

Пример. Поясним, что происходит, на небольшом примере. Пусть в исходном графе наилучший путь p^* от вершины 11 до вершины 17 был

$$11 \to 3 \to 7 \to 2 \to 12 \to 8 \to 17.$$

На второй итерации при $i1 = 7$ и $i2 = 12$ длина пути $7 \to 12$ станет равной длине пути $7 \to 2 \to 12$, и значит, теперь наилучшим будет и путь

$$11 \to 3 \to 7 \to 12 \to 8 \to 17.$$

На третьей итерации при $i1 = 11$ и $i2 = 7$ длина пути $11 \to 7$ станет равной длине пути $11 \to 3 \to 7$, и значит, теперь наилучшим будет и путь

$$11 \to 7 \to 12 \to 8 \to 17.$$

Несколько следующих итераций не внесут в этот путь изменений или внесут не окончательные изменения, но на седьмой итерации при $i1 = 11$ и $i2 = 12$ длина пути $11 \to 12$ станет равной длине пути $11 \to 7 \to 12$, и наилучшим будет и путь

$$11 \to 12 \to 8 \to 17.$$

Аналогичные изменения произойдут на 8 и 12 итерациях, и после них длина "прямого" пути $11 \to 17$ будет равна длине пути p^*, минимальность которого в исходном графе мы предположили.

Сам путь, на котором достигается минимум, легко строится, если одновременно с матрицей v мы будем пересчитывать матрицу $d[M, M]$, в которой $d[i, k]$ — выходящая из i дуга, с которой начинается путь, ведущий в k (для пересчета этой матрицы при каждом исполнении

*) Этот алгоритм носит имя Уоршолла–Флойда, так как он очень похож на алгоритм С. Уоршолла для транзитивного замыкания отношения. Роберт Флойд предложил его в том же 1962 г.

оператора присваивания внутри тройного цикла должен исполняться еще один оператор:

```
d[i1,i2] := d[i1,i]
```

— переход из $i1$ в $i2$ должен быть таким же, как в i). По завершении расчетов путь из любой вершины в любую легко создается по данным матрицы d.

Пример. Рассмотрим граф с четырьмя вершинами, A, B, C и D, и шестью дугами, данные о которых задаются таблицей:

Дуга	1	2	3	4	5	6
Начало	*A*	*A*	*B*	*C*	*B*	*D*
Конец	*D*	*C*	*D*	*B*	*A*	*C*
Длина	3	7	12	7	15	3

Строим матрицы v и d, заполняя их начальными значениями:

v	*A*	*B*	*C*	*D*
A			7	3
B	15			12
C		7		
D			3	

d	*A*	*B*	*C*	*D*
A			2	1
B	5			3
C		4		
D			6	

Начинаем расчеты: на первой итерации в качестве промежуточной вершины берем A. В столбце A матрицы v есть только один заполненный элемент, а в строке A — два. Переход $B \to A \to C$ длины $15 + 7 = 22$ нам следует принять, а переход $B \to A \to D$ длины $15 + 3 = 18$ отвергнуть, так как он хуже имеющегося. Получаем

v	*A*	*B*	*C*	*D*
A			7	3
B	15		22	12
C		7		
D			3	

d	*A*	*B*	*C*	*D*
A			2	1
B	5		5	3
C		4		
D			6	

Вершина B в качестве промежуточной интересна только для переходов из C, все получающиеся переходы новы и принимаются без всякого сравнения.

v	*A*	*B*	*C*	*D*
A			7	3
B	15		22	12
C	22	7	29	19
D			3	

d	*A*	*B*	*C*	*D*
A			2	1
B	5		5	3
C	4	4	4	4
D			6	

Вершина C в качестве промежуточной возможна для всех переходов, но интересна только для новых.

v	A	B	C	D
A	29	14	7	3
B	15	29	22	12
C	22	7	29	19
D	25	10	3	22

d	A	B	C	D
A	2	2	2	1
B	5	5	5	3
C	4	4	4	4
D	6	6	6	6

Осталось разобраться с вершиной D. Здесь изменений много, нужно пытаться исправить все переходы, но мы приведем прямо окончательный результат:

v	A	B	C	D
A	28	13	6	3
B	15	22	15	12
C	22	7	22	19
D	25	10	3	22

d	A	B	C	D
A	1	1	1	1
B	5	3	3	3
C	4	4	4	4
D	6	6	6	6

Теперь легко построить путь из любой вершины в любую. Например, для пути $B \to C$ получаем: $d[B, C] = 3$, путь начинается с дуги 3, следующей после B должна быть вершина D. Так как $d[D, C] = 6$ и end $6 = C$, то путь уже построен.

Строить всю матрицу кратчайших расстояний обычно слишком дорого, она занимает много места, поэтому чаще ищутся кратчайшие пути от какой-нибудь одной вершины i^- до всех вершин графа. Имеется несколько вариантов алгоритма; мы рассмотрим для начала метод, в котором напрямую используется принцип Беллмана.

Обозначим через $l(\mathrm{p})$ длину пути p, а через $v(i)$ длину кратчайшего пути от заданной вершины i^- до вершины i:

$$v(i) = \min\{l(\mathrm{p}) \mid \mathsf{end}\ p = i, \mathsf{beg}\ p = i^-\}.$$

Множество $\mathbf{P}(i)$ путей, на котором ищется минимум, можно разбить на непересекающиеся подмножества путей:

$$\mathbf{P}(i) = \bigcup_{\{u \mid \mathsf{end}\ u = i\}} \mathbf{P}_u,$$

где $\mathbf{P}_u$ — множество путей, кончающихся дугой u. По общим свойствам минимума (минимум функции на объединении нескольких множеств равен минимуму из минимумов, взятых по отдельным множествам) имеем

$$v(i) = \min\{\min\{l(\mathbf{p}) \mid \mathbf{p} \in \mathbf{P}_u\} \mid \mathsf{end}\ u = i\}.$$

Но каждый путь, кончающийся дугой u, может рассматриваться как соединение пути $\mathbf{p}'$ от i^- до вершины beg u с дугой u. Соответственно, $l(\mathbf{p}) = l(\mathbf{p}') + l(u)$. Пути $\mathbf{p}'$ в точности пробегают все множество $\mathbf{P}(\text{beg } u)$, поэтому

$$v(i) = \min\{l(u) + \min\{l(\mathbf{p}') \mid \mathbf{p}' \in \mathbf{P}(\text{beg } u)\} \mid \text{end } u = i\} =$$

$$= \min\{l(u) + v(\text{beg } u) \mid \text{end } u = i\}.$$

Уравнение

$$v(i) = \begin{cases} \min\{l(u) + v(\text{beg } u) \mid \text{end } u = i\}, & i \neq i^-, \\ 0, & i = i^- \end{cases}$$

называется *функциональным уравнением динамического программирования* или *уравнением Беллмана*. Решать это уравнение можно, например, методом последовательных приближений: для этого нужно задать некоторое *начальное приближение* $v^0(i)$ и затем находить последовательно функции $v^k(i)$, пересчитывая их из рекуррентного соотношения

$$v^k(i) = \begin{cases} \min\{l(u) + v^{k-1}(\text{beg } u) \mid \text{end } u = i\}, & i \neq i^-, \\ 0, & i = i^-. \end{cases}$$

При некоторых предположениях о графе легко показать, что, например, при выборе в качестве начального приближения

$$v^0(i) = \begin{cases} \infty, & i \neq i^-, \\ 0, & i = i^- \end{cases}$$

последовательность $v^k(i)$ сходится к $v(i)$ за конечное число шагов. Действительно, если, например, граф не содержит контуров, то правильные значения $v(i)$ будут установлены после k-го шага для тех вершин, к которым из i^- ведут пути, насчитывающие не больше k дуг.

Однако непосредственное использование уравнения Беллмана нерационально, и используются более изощренные методы.

Наиболее известен метод, предложенный Э. Дейкстрой [64] *).

*) Edsger Wybe Dijkstra (1930–2002), голландский математик и программист, долгое время живший в США. Он известен многими своими исследованиями и идеями в программировании, в частности успешной борьбой с безудержным использованием оператора `go to`. Описываемый здесь метод поиска кратчайших путей входит в число наиболее известных достижений Дейкстры.

Алгоритм Дейкстры для поиска кратчайших путей

Состояние вычислительного процесса. Все множество вершин M разбивается на три подмножества:

M_0 — вершины, расстояние до которых уже вычислено;

M_1 — вершины, расстояние до которых вычисляется;

M_2 — вершины, расстояние до которых еще не вычислялось.

Начальное состояние. Множество M_0 пусто, множество M_1 состоит из вершины i^-, M_2 — из всех остальных вершин. Расстояние от i^- до себя самой равно, разумеется, 0, и немного странно говорить, что это расстояние еще вычисляется, но так удобнее для алгоритма.

Стандартный шаг. В множестве M_1 выбирается вершина i_1, которой соответствует наименьшее текущее значение $v(i)$, и эта вершина переводится в M_0. При переводе просматриваются все дуги u, для которых beg $u = i_1$. Пусть end $u = i_2$ и $v' = l(u) + v(i_1)$ — предлагаемое расстояние до i_2 через дугу u. В зависимости от того, какому из множеств принадлежит вершина i_2, возможны три варианта действий:

если $i_2 \in M_0$, никаких действий не требуется;

если $i_2 \in M_2$, вершина i_2 переводится из M_2 в M_1, причем ей сопоставляется значение $v(i_2) = v'$;

если же $i_2 \in M_1$, то текущее значение $v(i_2)$ сравнивается с v' и, если новый путь выгоднее, полагается равным v'.

Эффективность метода Дейкстры существенно зависит от того, как организован поиск в множестве M_1 вершины с наименьшим текущим расстоянием. Можно сказать, что для представления множества M_1 используется конструкция приоритетной очереди, и сослаться на с. 180.

Долгое время мы успешно использовали в своих программах "ведерный" вариант приоритетной очереди, который в нашей научной литературе связывается с именем А. Б. Грибова [5]. Вершины множества M_1 разбиваются в этом методе на подмножества с одинаковым или близким, отличающимся меньше чем на минимальную длину дуги, значением $v(i)$ (эти подмножества и называются *ведрами*, и на каждой итерации берется вершина из непустого ведра, соответствующего минимальному значению расстояния).

Отметим, что при уменьшении текущего значения $v(i)$ вершину i не обязательно нужно переводить из одного ведра в другое, — достаточно сделать новую запись в нужном ведре, а на каждом шаге проверять, не устарела ли очередная запись.

Для нормальной работы алгоритма не требуется (кроме специально создаваемых задач) слишком большого массива ведер: достаточно размера $[\overline{c}/\underline{c}] + 1$, где $\overline{c}$ и $\underline{c}$ — соответственно, наибольшая и наименьшая длина дуги. Эти ведра используются в циклическом порядке — "освободившееся ведро приставляется к концу массива" (конечно, виртуально).

В недавнем обзоре [61] описано использование более сложных приоритетных очередей, которые справляются и со специально создаваемыми трудностями. Авторы считают наиболее эффективным использование "крупных корзин" для долгого хранения записей и биномиальных куч для оперативной работы.

Проводившееся нами конкурсное сравнение методов решения этой задачи показало, что, несмотря на очень высокую эффективность метода Дейкстры–Грибова, в практических задачах с "географической" системой расстояний еще более эффективен метод, принадлежащий Б. Ю. Левиту [32]. В методе Левита уравнение Беллмана используется непосредственно, с возможным пересчетом получающихся значений.

Алгоритм Левита

Состояние вычислительного процесса. Множество M также делится на подмножества M_0, M_1 и M_2, имеющие тот же самый смысл, с той только разницей, что попадание вершины в множество M_0 может быть не окончательным. Элементы, входящие в множество M_1, делятся на два упорядоченных подмножества M_1' и M_1''. Множество M_1' назовем основной очередью, а множество M_1'' — срочной очередью.

Начальное состояние. Множества M_0 и M_1'' пусты, множество M_1' состоит из элемента i^-, множество M_2 — из всех остальных вершин.

Стандартный шаг. Выбирается элемент i_1 для перевода из M_1 в M_0. Если множество M_1'' не пусто, берется очередной элемент из него, если пусто — очередной элемент из M_1'. Как и в методе Дейкстры, вершина i_1 переводится в M_0 и просматриваются дуги, выходящие из нее. Рассмотрим дугу u и ее конечную вершину i_2.

Если $i_2 \in M_2$, то вершина i_2, как и раньше, переводится из M_2 в M_1 (точнее, в конец очереди M_1'), причем ей сопоставляется значение $v(i_2) = l(u) + v(i_1)$.

Если $i_2 \in M_1$, то текущее значение $v(i_2)$ сравнивается с $l(u) + v(i_1)$ и полагается равным наименьшему из них. При этом вершина никак не передвигается в очереди.

Если же $i_2 \in M_0$ и $v(i_2) > l(u) + v(i_1)$, то вершина i_2 получает новое значение $v(i_2) := l(u) + v(i_1)$ и возвращается для повторной обработки в множество M_1, на этот раз в срочную очередь M_1''.

В сравнении с методом Дейкстры метод Левита проигрывает на том, что некоторые вершины приходится обрабатывать повторно, а выигрывает на более простых алгоритмах включения и исключения вершин из множества M_1. Эксперименты показывают, что для графов с "геометрическим" происхождением, т. е. для графов, построенных на основе транспортных сетей и реальных расстояний, метод Левита оказывается более быстрым. Он выигрывает и по размеру программы.

Метод Левита обладает еще и тем преимуществом перед методом Дейкстры, что он применим в случае отрицательных длин дуг (ведь "длина дуги" — это просто название, которое дает нам полезные ассоциации с реальностью). Если считать, что значения $l(u)$ не обязательно положительны, решение задачи о кратчайшем пути значительно усложняется.

Первая трудность в том, что теряется простое правило метода Дейкстры для определения окончательности вычисленного расстояния до конкретной дуги. Эта трудность, как мы увидим дальше, обходится, хотя и с некоторой потерей эффективности метода (приходится проверять все дуги, ведущие в данную вершину).

Вторая трудность серьезнее: при отрицательных длинах в графе могут найтись контуры с отрицательной суммой длин дуг (назовем такие контуры "отрицательными"). Прибавление к пути отрицательного контура уменьшает значение целевой функции, и чем больше обходов отрицательного контура мы прибавим, тем "лучше". Избавиться от бесконечного убывания минимума просто так невозможно, но есть два выхода из трудного положения (конечно же, выбор выхода зависит не от нас, а от решаемой практической задачи).

- Запретить включение в путь контуров, т. е. рассматривать только простые пути, но такой запрет делает задачу очень сложной.
- В случае отрицательных контуров считать, что задача решения не имеет, и ограничиться решением задачи в случаях, когда отрицательных контуров нет. В этом случае метод Левита даст требуемое оптимальное решение, а при некоторой модификации позволит "отлавливать" отрицательные контуры.

Однако преимущество метода Дейкстры в том, что он легко модифицируется для других задач на графах. В частности, очень важна

для приложений рассматриваемая дальше задача о максимальном пути в графе. Эта задача используется при управлении сложными комплексами работ, когда разыскивается цепочка работ, определяющая время выполнения всего комплекса.

Модифицированным методом Дейкстры могут решаться и задачи о выборе пути с учетом *пропускных способностей дуг*. Пропускная способность дуги задает максимальный размер транспортного потока, который дуга может пропустить в единицу времени. Пропускная способность пути определяется как минимум из пропускных способностей дуг, входящих в путь. (Есть интересная задача о максимальном потоке, который можно пропустить через транспортную сеть при ограничении пропускных способностей дуг.)

Можно рассматривать задачу поиска кратчайшего пути с пропускной способностью не ниже данной*). Этот вариант задачи решается очень просто: достаточно исключить из рассмотрения дуги графа, пропускная способность которых недостаточна. Более интересна следующая постановка задачи с другой целевой функцией.

Задача о пути с наибольшей пропускной способностью. Пусть каждой дуге u графа $\langle M, N\rangle$ сопоставлена ее (положительная) пропускная способность $d(u)$. Пусть пропускная способность $d(\mathbf{p})$ пути $\mathbf{p}$ определяется как $d(\mathbf{p}) = \min\{d(u) \mid u \in \mathbf{p}\}$. Требуется найти путь от заданной вершины i^- до другой заданной вершины i^+, обладающий наибольшей пропускной способностью. ⊠

Обозначим через $\delta(i)$ максимальную (*достижимую*) пропускную способность пути от i^- до вершины i. Легко получается следующий вариант уравнения Беллмана:

$$\delta(i) = \begin{cases} \max\{\min\{d(u), \delta(\mathbf{beg}\ u)\} \mid \mathbf{end}\ u = i\}, & i \neq i^-, \\ \infty, & i = i^-. \end{cases}$$

Это уравнение решается в стиле метода Дейкстры со следующими изменениями алгоритма:

1) первоначально в множество M_1 записывается вершина i^- с "бесконечным" значением достижимой пропускной способности (годится любое значение, превышающее максимальную пропускную способность дуг);

*) Такую задачу приходится решать реально, когда разрабатывается маршрут перевозки крупногабаритного груза, который не провезти по некоторым участкам транспортной сети.

2) при выборе вершины, переводимой из M_1 в M_0, выбирается вершина с наибольшим значением достижимой пропускной способности;

3) при переводе вершины из M_2 в M_1, а также при корректировке достижимой пропускной способности вершин из M_1 по дуге u в качестве новой предлагаемой пропускной способности следует взять $\min\{d(u), \delta(\text{beg } u)\}$.

Интересно соединить эти две характеристики пути — длину и пропускную способность — в единую целевую функцию. Начнем с простого замечания.

Пропускная способность пути может определять время доставки груза (или передачи информации, если речь идет о сетях связи). Время доставки зависит от объема груза R и пропускной способности $r(u) = R/d(u)$. В сети связи при непосредственной передаче информации без ее промежуточной буферизации передача должна идти со скоростью, определяемой самым медленным участком, так что для времени передачи по пути $\mathbf{p}$ мы получаем $r(\mathbf{p}) = \max\{r(u) \mid u \in \mathbf{p}\}$.

Задача о пути с минимальным временем доставки вполне аналогична предыдущей; для минимального времени доставки до вершины i получаем уравнение

$$\rho(i) = \begin{cases} \min\{\max\{r(u), \rho(\text{beg } u)\} \mid \text{end } u = i\}, & i \neq i^-, \\ 0, & i = i^-. \end{cases}$$

Недавно появилась еще одна постановка задачи об оптимальном пути, которую авторы назвали задачей о *наискорейшем пути* ([59], см. также [79]). В уже введенных обозначениях задача формулируется так.

Задача о наискорейшем пути. Найти путь $\mathbf{p}$ из вершины i^- до вершины i^+, на котором достигается минимум суммы стоимости пути $l(\mathbf{p})$ и $r(\mathbf{p})$. □

Можно предложить следующий подход к решению этой задачи.

Найдем прежде всего кратчайший путь (т. е. путь с минимальной длиной $l(\mathbf{p}_0)$). Зафиксируем время доставки на этом пути $r_0 = r(\mathbf{p}_0)$. Легко видеть, что в графе не существует пути с бо́льшим временем доставки и меньшим значением целевой функции. Удалим из графа все дуги со временем доставки, не превосходящим r_0. В получившемся графе опять найдем кратчайший путь и т. д.

В заключение заметим, что во многих задачах следует относиться к выбору графа, на котором разыскивается наилучший путь, достаточно свободно: если в задаче с самого начала имеется какой-то граф, ниоткуда не следует, что это тот граф, на котором нужно решать задачу динамического программирования.

8.8. Задачи о кратчайшем дереве путей

При постановке математических задач нужно внимательно следить за деталями — очень близкие по формулировке задачи могут поразительно различаться по методу решения и по структуре получаемого ответа. В качестве примера рассмотрим здесь две задачи о наилучшем дереве путей, в каждом случае — наилучшем по своему показателю. Первая из них очень похожа на задачу о дереве кратчайших путей (и одновременно на задачу о кратчайшем остовном дереве, от которой отличается тем, что искомое дерево должно быть ориентированным и задана корневая вершина, из которой дерево "растет") и решается достаточно просто, вторая задача очень трудна.

Итак, *первая задача.*

Задача о кратчайшем дереве путей. Пусть задан граф $G = \langle M, N \rangle$, каждой дуге которого $j \in N$ сопоставлено положительное число $l[j]$. Пусть задана также вершина i_0. Требуется построить ориентированное дерево $\langle M, N' \rangle$ с корнем i_0 (так что в нем должны существовать пути из i_0 во все остальные вершины), сумма длин дуг которого $\sum_{j \in N'} l[j]$ минимальна. ⊠

Пример. Различие между деревом кратчайших путей и кратчайшим деревом путей можно увидеть уже на очень простом примере (рис. 8.15). Для дерева кратчайших путей (левая картинка) нужно выбирать самый короткий путь до каждой вершины, и вершины 2 и 3 соединяются с вершиной 0 прямыми путями. Для кратчайшего дерева (правая картинка) нужно обеспечить

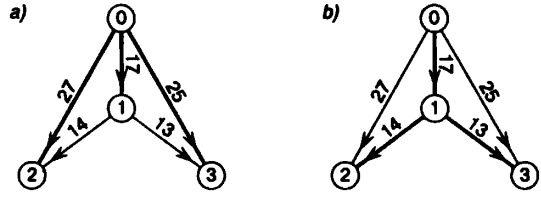


Рис. 8.15. Дерево кратчайших путей (*a*) и кратчайшее дерево путей (*b*)

пути до всех вершин с минимальными общими затратами, и здесь выгоднее достраивать пути до вершин 2 и 3 от пути до вершины 1.

Описываемый ниже алгоритм решения задачи о кратчайшем дереве путей (с точностью до деталей) был предложен китайскими математиками Чу Йонджином и Лю Цзенхонгом [62].

Пусть $M' = M \setminus \{i_0\}$. Прежде всего напомним, что для построения ориентированного дерева нужно для каждой вершины из M' выбрать одну входящую дугу, и если граф, составленный из этих дуг, будет связен, то получится ориентированное дерево. Действительно, это будет дерево, циклов и контуров в нем нет, и для каждой вершины $i \in M'$ есть входящая дуга. Строя из вершины i путь в обратном направлении, мы должны где-то остановиться, и можем это сделать только в тот момент, когда путь попадет в корень i_0.

Здесь и в дальнейшем нам понадобится следующая операция *спуска до нуля*: пусть на конечном множестве A задана функция $f : A \to R$. Вычислим $\varphi = \min\{f(a) \mid a \in A\}$ и положим $F(a) = f(a) - \varphi$. Получившаяся функция неотрицательна на множестве A, и ее минимум равен 0. Переход от f к F и называется *спуском до нуля* на множестве A.

Чтобы различать объекты разных итераций, обозначим исходный граф через $G_0 = \langle M_0, N_0 \rangle$.

Для каждой вершины $i \in M'$ спустим до нуля длины входящих в эту вершину дуг. Так как в искомое дерево должна быть включена ровно одна дуга, входящая в i, от этого спуска значение целевой функции уменьшится на вычитаемое для любого допустимого решения, и минимальное решение останется минимальным.

Построим частичный граф $P_0 = \langle M_0, Z_0 \rangle$, используя для множества Z_0 только дуги нулевой (после выполненного пересчета) длины и взяв из них по одной дуге, входящей в каждую вершину $i \in M'$. Если этот граф окажется ориентированным деревом с корнем в i_0, оно автоматически будет оптимальным решением.

Если граф P_0 окажется не деревом, мы будем его перестраивать. Построим его диаграмму порядка $D_0 = \langle M_1, Z_0' \rangle$. Каждая вершина диаграммы есть некоторое подмножество множества M_0, так что M_1 образует разбиение множества M_0, причем вершине i_0 соответствует одноэлементное множество. Заметим, что мощность множества M_1 заведомо меньше, чем M_0.

Образуем граф $G_1 = \langle M_1, N_1 \rangle$ таким образом: сначала у каждой дуги заменим начало и конец каждой вершины M_0 соответствующими вершинами из M_1, а затем "склеим" все дуги с одинаковыми началом

и концом. Примем в качестве длины этой дуги минимальную из длин всех составляющих ее дуг. Будем строить тем же способом кратчайшее дерево путей в G_1 — создавать P_1, затем, если потребуется, D_1, по нему G_2 и т. д. В конечном счете на какой-то итерации k мы построим граф G_k, в котором найдется ориентированное дерево.

После этого пойдет обратный процесс: дерево в графе G_k (обозначим его через T_k) мы превратим в дерево T_{k-1} в графе G_{k-1}. Для этого каждую вершину в G_k, являющуюся классом эквивалентности в G_{k-1}, превратим во фрагмент дерева. Это делается очень просто: пусть M_α — множество вершин этого класса. В графе G_{k-1} оно является множеством дуг некоторого контура. Удаление из этого контура одной дуги превращает контур в путь. Само множество M_α является вершиной дерева T_k, и в него входит некая дуга j', являющаяся дугой и в графе G_{k-1}. Удалим из цикла дугу j, для которой end j = end j'. Сделав это для каждого класса, мы и получим дерево T_{k-1}. Процесс закончится построением дерева T_0 для графа G_0.

Для доказательства минимальности дерева нам достаточно проверить корректность одного перехода.

Лемма. Кратчайшее дерево путей T_0 в графе G_0 можно получить, найдя кратчайшее дерево путей T_1 в графе G_1, а затем заменив в нем каждый класс деревом, построенным из дуг нулевой длины.

Доказательство. Зафиксируем любое дерево путей $\langle M_0, N'\rangle$ и покажем, что в графе G_0 найдется дерево не большей длины, имеющее такую структуру, как сказано в лемме. Для такой структуры дерева необходимо и достаточно, чтобы в каждое из подмножеств входило только по одной дуге. Меньше быть не может, иначе получится отдельная компонента связности. Если же в какое-то подмножество входит больше чем одна дуга, то все дуги, кроме одной, можно заменить дугами нулевой длины, лежащими внутри подмножества, что разве лишь уменьшит длину дерева и не нарушит связности. Повторяя это преобразование нужное число раз, мы добьемся искомой структуры дерева. □

Теперь мы можем выписать сам алгоритм.

Алгоритм построения кратчайшего дерева путей

1. Спустить до нуля длины дуг, входящих в каждую вершину M'.
2. Построить граф P_0, взяв для каждой вершины M' по одной дуге нулевой длины.

3. Если этот граф является ориентированным деревом, завершить работу алгоритма.
4. В противном случае построить новый граф G_1, вершинами которого являются сильно связные компоненты графа P_0.
   Этот пункт определяет рекурсивность алгоритма, и мы рассмотрим его более детально.
   (a) Построить новый граф G_1.
   (b) Решить для графа G_1 задачу о кратчайшем дереве.
   (c) Заменить каждую вершину полученного дерева деревом из нулевых дуг внутри соответствующей сильно связной компоненты.

Вторая задача. Как уже было сказано, сложность задач о кратчайших деревьях может резко возрасти даже при небольшом изменении условий. Вот обещанная вторая задача, которая называется задачей Штейнера на графах *).

Задача Штейнера на графах. Пусть задан граф $G = \langle M, N \rangle$, дугам которого сопоставлены положительные веса w, начальная вершина i_0 и множество конечных вершин $M_\dagger$. Требуется построить частичное дерево $\langle M', N' \rangle$, содержащее вершину i_0 и множество $M_\dagger$, а также пути из i_0 в любую вершину $M_\dagger$ так, чтобы сумма весов его дуг $\sum_{j \in N'} w[j]$ была минимальна. □

Надо заметить, что при небольших размерах множества $M_\dagger$ задача достаточно просто решается методами динамического программирования. Мы вернемся к этому вопросу в конце курса (см. пример на с. 295).

8.9. Сетевой график и критические пути

На терминах и изобразительных средствах теории графов базируется организация выполнения сложных проектов (программ, но уже

*) Jacob Steiner (1796–1863), известный швейцарский геометр, работавший в Германии. Он сформулировал и исследовал задачу о кратчайшем соединении нескольких точек на плоскости. Задача до сих пор не имеет хороших алгоритмов решения, хотя условия локального экстремума Штейнер полностью изучил. Его условия очень просты: кратчайшее соединение должно состоять из отрезков; отрезки, имеющие общие концы, должны встречаться под углом не меньше 120°; в каждой точке соединения, не принадлежащей заданному набору точек, должно встречаться ровно три отрезка. Рассматриваемая здесь задача соединения на графах является обобщением исходной задачи Штейнера.

не в программистском смысле, хотя речь может идти и о создании большой программной системы). Каждый такой проект обычно состоит из многих отдельных работ, выполняемых разными командами исполнителей, иногда очень большими — целыми организациями. Некоторые работы связаны друг с другом условиями предшествования: работа A не может начаться, пока не закончилась работа B.

Эта область человеческой деятельности заслуживает серьезного отношения, и уже давно развиваются специальные *методы организации*, предназначенные для администрирования проекта. Эти методы должны облегчать контроль за ходом проекта, предсказание его будущего развития и трудных мест, перераспределение ресурсов.

Первоначально для наглядного представления (теперь говорят *для визуализации*) расписания использовались всякого рода графики (ленточные диаграммы Ганта *) и др. [46]). Пример ленточной диаграммы вы можете увидеть на рис. 2.7.

Для описания современных проектов, в которых связи между работами значительно сложнее, такие средства оказались недостаточно эффективными, и возникло *сетевое планирование*, в котором зависимости работ описываются средствами теории графов. Способ представления работ и зависимостей между ними оказался очень интересным. Мы начнем с построения этого графа зависимостей, а затем рассмотрим некоторые из задач, решаемых с его помощью.

Первоначально кажется вполне естественным желание представить каждую работу вершиной графа, а зависимости между работами описывать дугами, как показано на рис. 8.16, *а*. Однако это представление, называемое графиком "работы-вершины", может стать неудобным, причем в достаточно типичных ситуациях. Это видно из рис. 8.16, *b*, на котором изображены две группы работ, и каждая работа из первой группы предшествует каждой работе из второй (кроме двух исключений, сделанных специально для того, чтобы вы увидели, как трудно их найти).

Чтобы избежать таких неудобств, предложили другой подход: строить граф зависимостей между работами, *считая работы дугами*. Вариант так и называется графиком "работы-дуги". В этом подходе важно разобраться в том, что представляют собой вершины получающегося

*) Henry Laurence Gantt (1861–1919), известный американский инженер, вместе с Ф. Тейлором разрабатывавший принципы научной организации труда. Свои знаменитые диаграммы он разработал во время Первой мировой войны.

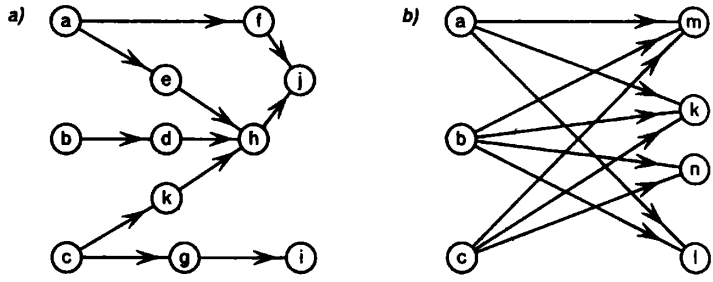


Рис. 8.16. График "работы-вершины":
a — нормальный пример; *b* — неприятный пример

графа. Они называются *событиями* и в руководствах по сетевым графикам туманно определяются как начала и концы работ. Четкого и короткого определения событий не дается, и это не случайно. Но так как важно, что у некоторых работ начала или концы становятся общими, то мы сейчас рассмотрим механизм этого объединения, чтобы прояснить, что же такое события.

Исходной информацией для построения графа служат множество работ W и заданное на нем отношение предшествования: каждой из работ $w \in W$ сопоставляется некоторое подмножество (возможно, пустое) ее предшественников P_w. Это реальные данные, в том виде, как они появляются при разработке проекта, и они могут быть неполны и противоречивы.

Противоречивость связана с возможностью контуров: работы, составляющие контур, не могут начаться и ждут друг друга. Первая задача, встающая при разработке проекта, — это поиск контуров и их размыкание. Отметим, что формализовать процесс размыкания контуров (а одна подходящая задача нам встретится в дальнейшем) в данной ситуации невозможно; априорно допустимо любое упорядочение работ, и единственная приемлемая возможность разрешения коллизии — это экспертный ее анализ, вне рамок математической модели.

После того как отношение предшествования P очищено от контуров, его следует транзитивно замкнуть, так как разработчики, предоставляющие исходную информацию, видят, как правило, только ближайших предшественников каждой работы. Транзитивное замыкание $\overline{P}$ найдет для каждой работы всех ее предшественников.

Теперь появляется возможность некоторой унификации начал работ: определения тех из них, которые имеют общий набор предшественников. Рассмотрим для этого совокупность множеств $\overline{P}_w$, $w \in W$, и удалим в этой совокупности повторы; мы получим некоторый набор множеств π_α, $\alpha \in A$, и каждому $w \in W$ будет отвечать индекс $\alpha(w)$, такой что $\overline{P}_w = \pi_{\alpha(w)}$.

Следующий шаг может вызвать у читателя некоторое недоумение, поэтому лучше немного объяснить. Нам сейчас существенно классифицировать работы и разбить их на группы, одинаково входящие или не входящие в множества π_α. У нас есть математический аппарат для такой классификации — это аппарат разбиений множеств с операцией произведения разбиений.

Каждому π_α сопоставим разбиение $\Pi_\alpha : W = \pi_\alpha \cup (W \setminus \pi_\alpha)$. Рассмотрим произведение $\overline{\Pi} = \{\rho_\beta\}_{\beta \in B}$ разбиений Π_α. Разбиение $\overline{\Pi}$ мельче любого из разбиений-сомножителей, а следовательно, каждое из множеств π_α является объединением некоторого набора множеств ρ_β. Нам этот набор понадобится. Обозначим через $B(\alpha) \subset B$ тот набор индексов, для которого $\pi_\alpha = \bigcup_{\beta \in B(\alpha)} \rho_\beta$.

Так как объединение всех множеств ρ_β совпадает с W и эти множества не пересекаются, то каждый элемент W принадлежит ровно одному из них. Иными словами, каждому $w \in W$ однозначно сопоставляется такой индекс $\beta(w) \in B$, что $w \in \rho_{\beta(w)}$.

Теперь мы имеем все, что нужно для построения первоначального графа. Построим граф $\langle M, N, T \rangle$, где $M = A \cup B$ (считая, что выбранные нами множества индексов A и B дизъюнктны), $N = W \cup F$, $T(w) = (\alpha(w), \beta(w))$, дополнительное множество дуг F будет определено позднее.

Сейчас отметим, что с дугами-работами все достаточно просто: каждая работа — это дуга, начала таких дуг — индексы из A, концы — индексы из B. Но получился граф, в котором начала и концы принадлежат разным множествам (такой граф называется двудольным). Нужны дополнительные дуги, чтобы собрать граф из фрагментов. Ими будут дуги из множества F. Они как бы собирают каждое множество π_α из элементов разбиения $\overline{\Pi}$, и множество F состоит из дуг, соответствующих всем парам (β, α), где начало дуги $\beta \in B(\alpha)$, конец дуги $\alpha \in A$. Дуги из множества F принято называть *фиктивными работами*.

Пример. Пусть множество работ состоит из 11 элементов $W = \{a, b, c, d, e, f, g, h, i, j, k\}$. Пусть $P_a = P_b = P_c = \varnothing$, $P_d = \{b\}$, $P_e = P_f = \{a\}$,

$P_g = P_k = \{c\}$, $P_h = \{b, d, e, k\}$, $P_j = \{f, g, h\}$, $P_i = \{c, g\}$. Построим транзитивное замыкание отношения P:

	a	b	c	d	e	f	g	h	i	j	k	
a												$\pi 0$
b												$\pi 0$
c												$\pi 0$
d		b										$\pi 1$
e	a											$\pi 2$
f	a											$\pi 2$
g			c									$\pi 3$
h	a	b	c	d	e						k	$\pi 4$
i			c				g					$\pi 5$
j	a	b	c	d	e	f	g	h			k	$\pi 6$
k			c									$\pi 3$

Последним столбцом в этой таблице записаны номера множеств, среди которых оказалось семь различных. Произведение соответствующих разбиений дает нам множества $\rho 1 = \{a\}$, $\rho 2 = \{b\}$, $\rho 3 = \{c\}$, $\rho 4 = \{g\}$, $\rho 5 = \{d, e, k\}$, $\rho 6 = \{f, h\}$, $\rho 7 = \{i, j\}$.

Теперь осталось построить граф. Множество F состоит из дуг, ведущих из $\rho 1$ в $\pi 2$, $\rho 2$ в $\pi 1$, из $\rho 3$ в $\pi 3$, из $\rho 1$, $\rho 2$, $\rho 3$, $\rho 5$ в $\pi 4$, из $\rho 3$, $\rho 4$ в $\pi 5$, из $\rho 1$, $\rho 2$, $\rho 3$, $\rho 4$, $\rho 5$, $\rho 6$ в $\pi 6$. Получаем граф, изображенный на рис. 8.17 (пунктирные дуги — фиктивные). В этом графе есть начальная вершина, из которой исходят дуги (начинаются работы), не имеющие предшественников, и конечная вершина — заключительное событие, в нем заканчиваются дуги, за которыми не должны следовать другие дуги. Обозначим начальную вершину графа через i_0, а заключительную через $i_\dagger$.

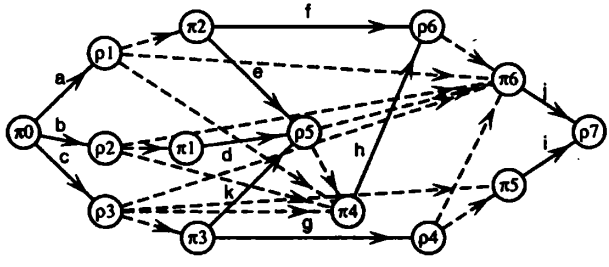


Рис. 8.17. Первоначальный сетевой график

Получившийся граф может быть существенно упрощен, и эти упрощения нужно сделать, так как предложенная здесь универсальная и логически правильная конструкция неудобна для использования.

Имеется несколько типов преобразований, которые упрощают граф. Прежде всего, если начало и конец какой-либо фиктивной дуги соединены "обходным" путем, не содержащим этой дуги, то такая дуга может быть удалена из графа. Во-вторых, если какая-либо фиктивная дуга — единственная дуга, выходящая из вершины или входящая в вершину, то дуга может быть удалена из графа склеиванием ее начала и конца в одну вершину. Эти операции можно применять в любом порядке до полного насыщения.

Пример (продолжение). Дуга $(\rho 1, \pi 6)$ может быть удалена, так как существует обходный путь $\rho 1 \to \pi 2 \to \rho 6 \to \pi 6$. Аналогично удаляются дуги $(\rho 2, \pi 6)$, $(\rho 3, \pi 6)$, $(\rho 4, \pi 6)$ и $(\rho 5, \pi 6)$, а также $(\rho 3, \pi 5)$, $(\rho 1, \pi 4)$, $(\rho 2, \pi 4)$ и $(\rho 3, \pi 4)$. А теперь можно склеить вершины $\rho 1$ и $\pi 2$, $\rho 2$ и $\pi 1$, $\rho 3$ и $\pi 3$, $\rho 4$ и $\pi 5$, $\rho 5$ и $\pi 5$, $\rho 6$ и $\pi 6$. Таким образом, окончательно получаем граф, представленный на рис. 8.18. Отметим, что этот граф изображает ту же зависимость работ, что и граф на рис. 8.16, *а*.

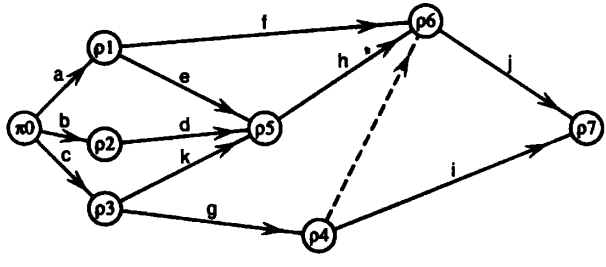


Рис. 8.18. Окончательный вид сетевого графика

Сам по себе этот граф, называемый *сетевым графиком* проекта, очень важен для общего наглядного представления о предстоящей работе. Но он может быть использован и для получения важных количественных характеристик, прежде всего, связанных со временем выполнения.

Представим себе, что каждой работе $w \in W$ сопоставляется время ее исполнения $t_w > 0$. Для единообразия фиктивным работам также сопоставляется время их выполнения, равное 0. Работа не может начаться прежде, чем будут выполнены все предшествующие ей работы, поэтому продолжительность выполнения всего проекта не может быть меньше, чем сумма продолжительностей работ, входящих в любой путь от начального до заключительного события. Сумму продолжительностей работ, входящих в путь, будем называть дальше *длиной пути*. Путь наибольшей длины называется *критическим путем*.

Критические пути и пути, близкие к ним по длине, наиболее существенны для соблюдения сроков выполнения проекта, эти пути находятся и при первоначальном анализе сетевого графика, и в процессе выполнения проекта, когда реальность уже внесла свои изменения в структуру графика и в оценку времен выполнения работ.

Критический путь вычисляется почти по методу Дейкстры, нужны совсем небольшие поправки, связанные с тем, что ищется путь максимальной, а не минимальной длины. Как и в методе Дейкстры, каждой вершине графа $i \in M$ (каждому событию) сопоставляется число $v[i]$, называемое в этой задаче *ранним наступлением* события i и равное наибольшей длине пути от начального события до i.

Ранние начала событий могут быть определены из соотношений

$$v[i] = \max\{t_j + v[\text{beg } j] \mid \text{end } j = i\}$$

и условия $v[i_0] = 0$ для начального события i_0. Время раннего наступления заключительного события называется *критическим временем*, проект не может быть выполнен быстрее, конечно, без изменения исходных данных.

Алгоритм нахождения ранних наступлений событий

Состояние вычислительного процесса. Разбиение множества вершин $M = M_0 \cup M_1 \cup M_2$. Аналогично алгоритму Дейкстры, это множества вершин с вычисленными, вычисляемыми и еще не затронутыми вычислением ранними временами наступления. Каждой вершине $i \in M$ сопоставлено целое неотрицательное число $r[i]$ — количество еще не просмотренных дуг с концом i и число $v[i]$ — раннее наступление события i.

Начальное состояние.

$$M_0 = \varnothing, \qquad M_1 = \{i_0\}, \qquad M_2 = M \setminus M_1,$$
$$v[i] = 0, \qquad r[i] = |\{j \mid \text{end } j = i\}|, \qquad i \in M.$$

Стандартный шаг.

1. Если множество M_1 пусто, завершить вычисления.
2. Если же оно не пусто, то выбрать в нем любую вершину, назовем ее i_1. Перевести эту вершину в M_0.
3. Для каждой дуги j, выходящей из i_1, и ее конца $i_2 = \text{end } j$ сравнить $v[i_2]$ и $v' = t[j] + v[i_1]$ и увеличить $v[i_2]$ до v', если v' больше. Уменьшить $r[i_2]$ на 1 и при $r[i_2] = 0$ перевести i_2 из M_2 в M_1.

Аналогично ранним наступлениям событий могут вычисляться поздние наступления. Взяв в качестве исходных моментов для каждого события критическое время проекта и считая это время уже вычисленным моментом для заключительного события, мы вычисляем заново поздние наступления для всех остальных событий по формуле

$$V[i] = \min\{-t_j + V[\text{end } j] \mid \text{beg } j = i\}.$$

Нахождение поздних моментов наступления событий вполне аналогично нахождению ранних начал.

Вычислим теперь для каждой работы j величину

$$d[j] = V[\text{end } j] - t[j] - v[\text{beg } j].$$

Эта величина называется *резервом времени работы* j. Резерв времени каждой работы неотрицателен, так как представляет собой разность длины критического пути и длины какого-то конкретного пути от начальной вершины до конечной. Работы (дуги), имеющие нулевой резерв времени, называются *критическими*.

Лемма. Каждая критическая дуга лежит на каком-нибудь критическом пути.

Доказательство очевидно.

Модель сетевого графика имеет большое значение как хороший организующий инструмент, выделение критических путей позволяет организаторам выделить область наибольшего внимания, хотя, конечно, она не исчерпывается критическими путями по многим причинам.

Прежде всего, сами продолжительности работ удается указать лишь примерно; по мере выполнения работ, когда выясняются их истинные продолжительности, сетевой график для оставшейся части проекта должен пересчитываться. Полезно несколько расширять область контроля, вводя в нее так называемые *субкритические пути*, такие, у которых резерв времени мал и небольшое изменение продолжительности может вывести их на критический путь.

Редко какие практические проекты анализируются полностью в технике критических путей *).

Дело в том, что отдельные работы, входящие в проект, обычно связаны не только технологическими предшествованиями, но и совместно используемыми и обычно дефицитными ресурсами, в частности рабочей силой и оборудованием. Правильная модель должна составлять "расписание" выполнения проекта, и в этом расписании требуется согласование выполнения работ по каждому

*) Например, в Ленинграде это были реконструкция Литейного моста и строительство Дворца спорта "Юбилейный".

из ресурсов. Могут накладываться дополнительные ограничения на выполнение работ, в частности выполнение работ в одних случаях может прерываться, а в других нет. Используемые ресурсы могут быть *складируемыми*, как деньги и материалы, и *нескладируемыми*, как рабочая сила, время используемого оборудования или электроэнергия.

Так как задача составления расписаний хорошо всем известна по школьному быту, то на каждом курсе находится один или несколько студентов, обуреваемых идеей алгоритмизации составления расписания занятий. Поэтому я должен предупредить, что это очень сложная задача. Она сложна еще и потому, что если ресурсы не дефицитны, то ручное составление расписания не составляет большого труда, а когда становится трудно, то у диспетчера-человека есть в запасе волшебное средство, которого пока нет у машины, — попросить кого-нибудь из участников системы ослабить свои ограничения.

Существует особая наука *теория расписаний*, изучающая и систематизирующая задачи такого рода, а также различные приближенные методы их решения (на точные методы надежды почти нет). Особое место среди них занимают *эвристические* методы, в которых делаются попытки описать логику и технику действий диспетчера [*)].

8.10. Теория паросочетаний и ее применения

В этом параграфе мы увидим, как терминология теории графов может работать при изучении комбинаторных свойств связей между элементами двух множеств, скажем A и B. Связи, которые имеются в виду, формально можно описать отношением на $A \times B$. Интересуют нас взаимнооднозначные отображения между подмножествами A и B.

Пример. В качестве удачной иллюстрации назовем всем известную задачу о расстановке на шахматной доске ладей, не находящихся под ударом друг друга. Множества вертикалей и горизонталей выступают в роли A и B, их прямое произведение — это множество полей доски; группа не бьющих друг друга ладей задает взаимнооднозначное отображение между использованными вертикалями и горизонталями. Например, ладьи, расположенные в a5, b7 и e4, задают отображение вертикалей {a, b, e} в горизонтали {4, 5, 7}.

Если рассматривать доски произвольного размера и ограничивать множество полей досок, на которые разрешается ставить ладьи, то это будет уже полное описание интересующей нас сейчас задачи.

Граф $\langle M, N\rangle$ называется *простым*, или *двудольным*, если множество его вершин разбито на два множества M_b и M_e и все начала дуг принадлежат M_b, а все концы — M_e.

*) От греческого *ευρισκω* — находить. Эвристикой в Древней Греции называлась система обучения в виде беседы с наводящими вопросами.

Такие графы удобно рассматривать при описании соответствий между элементами двух множеств. Одна из наиболее характерно возникающих в этой связи является задача построения полных или частичных взаимнооднозначных соответствий между элементами этих множеств, т.е. составление допустимых пар элементов (i, k), $i \in M_b$, $k \in M_e$, с неповторяющимися i и k.

Набор дуг $J \subset N$ называется *паросочетанием* *), если для любых $j_1, j_2 \in J$, $j_1 \neq j_2$, начала и концы этих дуг различны: beg $j_1 \neq$ beg j_2, end $j_1 \neq$ end j_2.

Посмотрим, как можно построить паросочетание, максимальное по числу входящих в него дуг.

Сначала будет предложен алгоритм, который строит (с постепенным увеличением размера) некоторое паросочетание, а потом окажется, что оно действительно максимально. Мы заранее отметим это свойство алгоритма в его названии.

Вместе с паросочетанием будет строиться некое специальное множество вершин. Скажем, что множество вершин M' *контролирует* дугу j, если начало или конец этой дуги принадлежит M'. Нас интересует *контрольное множество*, в котором для любой дуги j найдется контролирующая ее вершина **). Если размер паросочетания мы стремимся максимизировать, то контрольное множество естественно искать наименьшего размера.

Оказывается, размер наибольшего паросочетания и наименьшего контрольного множества совпадают. Это дает нам пример замечательного явления — *двойственности экстремальных задач*, о которой будет немного подробнее сказано в следующей главе.

В алгоритме вместе с паросочетанием будет строиться некоторое множество, составленное из начал и концов дуг, входящих в текущее паросочетание (по одной вершине от каждой дуги), так что размеры паросочетания и множества совпадают, и все дуги паросочетания контролируются. Назовем это множество *тестовым*. Работа алгоритма закончится, когда тестовое множество станет контрольным, т.е. будет контролировать все дуги из N.

Нам будет удобно обозначать через beg K и end K множества, соответственно, начал и концов любого множества дуг K:

$$\text{beg } K = \{\text{beg } j \mid j \in K\}, \qquad \text{end } K = \{\text{end } j \mid j \in K\}.$$

*) Pairmatching.

**) Контрольное множество называется также *вершинным покрытием графа*.

Алгоритм построения максимального паросочетания

Состояние вычислительного процесса. На каждом шаге алгоритма имеется текущее паросочетание N', разбитое (так определено состояние) на два подмножества $N' = N'_b \cup N'_e$. Объединение M' множеств $M'_b = \mathsf{beg}\ N'_b$ и $M'_e = \mathsf{end}\ N'_e$ составляет текущее тестовое множество. Каждой дуге $j \in N'_e$ будет сопоставлена другая дуга $\delta(j)$ — ее *дублер*, — не входящая в паросочетание и имеющая тот же конец, что и j ($\mathsf{end}\ \delta(j) = \mathsf{end}\ j$). Будет удобно приписать каждой дуге из N'_e положительный целочисленный *ранг* $\rho(j)$. Откуда же эта информация берется? Немного терпения! Каждая дуга оснащается ею при включении ее в N'_e, вначале же это множество пусто.

Начальное состояние. Первоначально множество N' пусто. Соответственно, пусты и множества N'_b, N'_e и M'.

Стандартный шаг. Если все дуги из N контролируются множеством M', работа алгоритма завершается.

Пусть нашлась дуга j_0, не контролируемая множеством M', так что $\mathsf{beg}\ j_0 \notin \mathsf{beg}\ N'_b$, $\mathsf{end}\ j_0 \notin \mathsf{end}\ N'_e$. Возможны четыре случая:

1) $\mathsf{beg}\ j_0 \notin \mathsf{beg}\ N'$, $\mathsf{end}\ j_0 \notin \mathsf{end}\ N'$.
Мы нашли дугу с началом и концом, не встречающимися в текущем паросочетании. Эту дугу можно добавить к паросочетанию. Положим $N' := N' \cup \{j_0\}$, $N'_b := N'$, $N'_e := \varnothing$, $M' := \mathsf{beg}\ N'$.

2) $\mathsf{beg}\ j_0 \notin \mathsf{beg}\ N'$, $\mathsf{end}\ j_0 \in \mathsf{end}\ N'$.
Так как $\mathsf{end}\ j_0 \notin \mathsf{end}\ N'_e$, но $\mathsf{end}\ j_0 \in \mathsf{end}\ N'$, то существует такая дуга $j_1 \in N'_b$, что $\mathsf{end}\ j_0 = \mathsf{end}\ j_1$. Переведем дугу j_1 из множества N'_b в N'_e, назначив ей дублером дугу j_0 (вот и появился дублер) и приписав им всем ранг 1.

3) $\mathsf{beg}\ j_0 \in \mathsf{beg}\ N'$, $\mathsf{end}\ j_0 \in \mathsf{end}\ N'$.
Существует такая дуга $j_1 \in N'_b$, что $\mathsf{end}\ j_0 = \mathsf{end}\ j_1$, но, кроме того, существует такая дуга $j_2 \in N'_e$, что $\mathsf{beg}\ j_0 = \mathsf{beg}\ j_2$. Переведем дугу j_1 из N'_b в N'_e, назначив ей дублером дугу j_0 и приписав им всем ранг $\rho(j_2) + 1$. Отметим, что так наращиваются постепенно цепочки из дуг и дублеров, и каждая кончается парой дуг первого ранга.

4) $\mathsf{beg}\ j_0 \in \mathsf{beg}\ N'$, $\mathsf{end}\ j_0 \notin \mathsf{end}\ N'$.
Существует такая дуга $j_2 \in N'_e$, что $\mathsf{beg}\ j_0 = \mathsf{beg}\ j_2$. Присоединение j_0 к цепочке, начинающейся с j_2, дает нам цепочку с нечетным числом дуг. В ней дуг из паросочетания на одну меньше,

чем дублеров (считая j_0). При этом начала и концы дублеров — это те же вершины, что у дуг паросочетания, а начало дублера первого ранга и конец дуги j_0 в паросочетании не встречались. Заменим дуги паросочетания их дублерами. Размер множества N' вырастет на 1, и получится новое паросочетание. Переведем все дуги паросочетания из N'_e в N'_b.

Пример. Рассмотрим двудольный граф, в котором множество начальных вершин $M_b = \{a, b, c, d, e, f, g, h\}$, а множество конечных вершин $M_e = 1 : 8$. Множество дуг N зададим для наглядности матрицей, в которой строки соответствуют элементам из M_b, а столбцы — элементам из M_e. Клетки матрицы, "занятые" знаком +, соответствуют дугам графа.

	1	2	3	4	5	6	7	8
a	+		+					
b	+							
c			+	+		+		+
d		+	+					
e	+	+	+		+		+	
f			+					
g	+		+					
h	+	+		+	+			

Чтобы сократить неинтересную часть вычислений, накопим предварительно какое-либо начальное паросочетание, отобрав дуги, подходящие под первый случай, например взяв $N' = \{a1, c3, d2, e5, h4\}$. Итак, пусть все эти дуги включены в N'_b, тестовое множество состоит из начальных вершин $M'_b = \{a, c, d, e, h\}$.

Неконтролируемая дуга $b1$ относится ко второму случаю. Переводим дугу $a1$ из N'_b в N'_e с дублером $b1$: $N'_b = \{c3, d2, e5, h4\}$, $N'_e = \{a1(b1)\}$. Теперь не контролируется дуга $a3$ (третий случай). Переводим дугу $c3$ из N'_b в N'_e с дублером $a3$: $N'_b = \{d2, e5, h4\}$, $N'_e = \{a1(b1), c3(a3)\}$. Теперь дуга $c6$ относится к четвертому случаю. Мы построили цепочку, изображенную на рис. 8.19 (дуги из паросочетаний — более жирные).

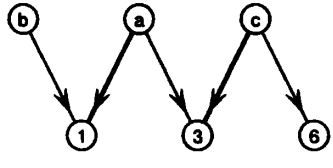


Рис. 8.19. Цепочка замен

Заменяя входящие в цепочку дуги паросочетания остальными дугами цепочки, получаем $N' = N'_b = \{a3, b1, c6, d2, e5, h4\}$. В результате двух следующих итераций дуги $a3$ и $b1$ перейдут из N'_b в N'_e, и мы получим

$$N'_b = \{c6, d2, e5, h4\}, \quad N'_e = \{a3(f3), b1(g1)\}, \quad M' = \{c, d, e, h, 3, 1\}.$$

Все дуги контролируются, процесс завершен.

Лемма. Описанный алгоритм сходится за конечное число шагов, которое не превосходит $(s+1)(s+2)/2$, где s — размер максимального паросочетания.

Доказательство. Рассмотрим в качестве характеристики состояния вычислительного процесса пару (r, k), где $r = |N'|$, $k = |N'_e|$. Возможные значения r ограничены сверху размером максимального паросочетания s, а k в любой момент не превосходит r. Вычисления начинаются с пары $(0, 0)$, а дальше на каждом шаге либо r увеличивается на 1 и k обращается в нуль (случаи 1 и 4), либо k увеличивается на 1. Таким образом, всего имеется $(s+1)(s+2)/2$ возможных значений этой характеристики процесса, и вычислительный процесс не может вернуться к уже встречавшемуся значению. Следовательно, он окончится за конечное число шагов.

Но окончиться этот процесс может только в случае, если все дуги графа контролируются. Значит, получающееся в результате работы алгоритма множество контролирует все дуги графа. □

Теорема. Размер максимального паросочетания в двудольном графе равен минимальному размеру контрольного множества.

Доказательство. Прежде всего, размер *любого* паросочетания не превосходит размера *любого* контрольного множества, так как каждая дуга из паросочетания должна контролироваться отдельной вершиной. Далее, согласно лемме, описанный алгоритм строит паросочетание N' и контрольное множество M' одного размера. Любое паросочетание не превосходит по размеру N', и следовательно, размер N' максимален. Любое контрольное множество имеет размер не меньше $|M'|$, и следовательно, размер M' минимален. □

Обозначим через $\Gamma(A)$ множество вершин, в которые по дугам графа можно перейти из множества начал дуг $A \subset M_b$:

$$\Gamma(A) = \{\mathsf{end}\ j \mid \mathsf{beg}\ j \in A\}.$$

Назовем *дефицитом* графа максимальную разность размеров множества начал и множества концов:

Следствие 1. Размер максимального паросочетания в графе равен $|M_b| - \delta$.

Доказательство. Можно искать контрольное множество в виде пары $(M_b \setminus A, \Gamma(A))$ (каждая дуга либо кончается в $\Gamma(A)$, либо начинается в дополнении M_b до A). Если так, то размер минимального множества равен

$$\min\{|M_b \setminus A| + |\Gamma(A)| \mid A \subset M_b\} = |M_b| - \max\{|A| - |\Gamma(A)|\} = \\ = |M_b| - \delta. \quad \square$$

Следствие 2. Для того чтобы размер максимального паросочетания был равен $|M_b|$, необходимо и достаточно, чтобы для любого $A \subset M_b$ размер $\Gamma(A)$ был не меньше размера A.

Доказательство. Утверждение прямо вытекает из предыдущего следствия. $\square$

Следствие 3. Пусть $a[K, L]$ — матрица из нулей и единиц, причем $|K| = |L|$. Для того чтобы существовало такое взаимнооднозначное отображение $\psi : K \to L$, что $a[k, \psi(k)] = 1$ для всех $k \in K$, необходимо и достаточно, чтобы любая подматрица $a[K', L']$, где $|K'| + |L'| > |K|$, имела ненулевые элементы *).

Доказательство. Рассмотрим двудольный граф $\langle M, N\rangle$, у которого $M_b = K$, $M_e = L$, а дуги соответствуют единицам в матрице $a[K, L]$. Взаимнооднозначное отображение, о котором говорится в теореме, — это просто паросочетание максимального размера; для того чтобы оно существовало, необходимо и достаточно, чтобы минимальный размер контрольного множества был равен $|K|$. Между тем каждой нулевой подматрице $a[K', L']$ соответствует контролирующее множество $(K \setminus K') \cup (L \setminus L')$, размер которого равен $2|K| - (|K'| + |L'|)$. Значит, для существования *полного паросочетания* необходимо и достаточно, чтобы такой подматрицы не нашлось. $\square$

Другое важное приложение этой теоремы, дающее еще один пример пары *двойственных экстремальных задач*, мы рассмотрим в виде отдельной теоремы.

Назовем две вершины графа *несравнимыми*, если в графе нет пути, которому принадлежат обе эти вершины.

*) Неравенство в формулировке означает, что сумма высоты и ширины нулевой подматрицы больше порядка исходной матрицы. Действительно, нулевая подматрица $1 \times n$ или $n \times 1$ делает нужное нам отображение невозможным. Соль в том, что также опасна и любая нулевая подматрица размера $(k + 1) \times (n - k)$.

Теорема (Дилворт*)). Минимальное число путей, которыми можно покрыть все вершины графа без контуров $\langle M, N\rangle$, равно размеру максимального множества попарно несравнимых вершин.

Доказательство. В каждом множестве попарно несравнимых вершин каждая такая вершина должна лежать на отдельном пути при покрытии множества вершин путями. Поэтому вершин в любом таком множестве не больше, чем путей в любом покрытии.

Итак, максимум не превосходит минимума, и доказательство будет завершено, если мы предъявим множество и покрытие одного размера. Мы воспользуемся теоремой о максимальном паросочетании.

Граф $\langle M, N\rangle$ задает отношение P на множестве вершин M. Рассмотрим его транзитивное замыкание $\overline{P}$ и по нему построим двудольный граф $\langle \overline{M}, \overline{N}\rangle$, где $\overline{M} = \overline{M}_b \cup \overline{M}_e$. Каждое из этих двух множеств — как бы отдельный экземпляр множества M.

Построим в графе $\langle \overline{M}, \overline{N}\rangle$ максимальное паросочетание $\overline{N}'$ и минимальное контрольное множество $\overline{M}' = \overline{M}'_b \cup \overline{M}'_e$. Пусть размер паросочетания равен s, так что $|\overline{M}'| = |\overline{N}'| = s$. Теперь уже множество $\overline{M}$ нам не будет нужно, мы вспоминаем, что каждая дуга паросочетания начинается и кончается в M. Контрольное множество M' состоит из двух подмножеств, M'_b и M'_e, суммарный размер которых равен s, и для каждой дуги из паросочетания ровно одна из инцидентных ей вершин принадлежит контрольному множеству.

Покажем, что дуги, включаемые в паросочетание $\overline{N}'$, порождают покрытие исходного графа ($|M| - s$) путями. Для этого покажем, что граф $\langle M, \overline{N}'\rangle$ состоит из ($|M| - s$) компонент связности. Возьмем граф без дуг $\langle M, \varnothing\rangle$, состоящий из $|M|$ компонент связности, в котором все вершины покрыты $|M|$ путями длины 0. Будем добавлять к нему по одной дуге паросочетания. Каждое такое добавление соединяет два простых пути в один, так как каждая вершина графа не больше одного раза служит началом дуги паросочетания и не больше одного раза — концом, и уменьшает число компонент связности на 1. После s итераций у нас останется ($|M| - s$) компонент связности, и каждая из них будет путем. Покрытие графа путями нашлось!

Теперь нам нужно раздобыть множество попарно несравнимых вершин размера ($|M| - s$). На его роль естественно претендует множество $D = M \setminus (M'_b \cup M'_e)$. Чтобы претензии были законными, нужно, прежде

*) Robert Palmer Dilworth (1914 1993), американский математик. Встречается и другая транскрипция его имени: Дилуорс. Эта теорема была опубликована в 1950 г.

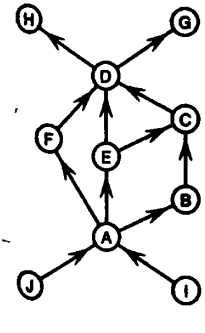
всего, чтобы размер этого множества был таким, как надо. Для этого необходимо, чтобы множества M'_b и M'_e не пересекались. Докажем это.

В самом деле, пусть они имеют общую вершину i_0. Так как $i_0 \in M'_e$, то в паросочетании имеется дуга j_1, кончающаяся в i_0, — ее начало не может принадлежать контрольному множеству. Равным образом, так как $i_0 \in M'_b$, то в паросочетании имеется дуга j_2, начинающаяся в i_0, — ее конец не может принадлежать контрольному множеству. По транзитивной замкнутости отношения $\overline{P}$ пара (beg j_2, end j_1) содержится в $\overline{P}$, а в силу сказанного она не может контролироваться. Значит, предположение о непустоте $M'_b \cap M'_e$ неверно.

Покажем теперь, что вершины множества D попарно несравнимы. Это уже просто. Существование пути из, скажем, i_1 в i_2 равносильно тому, что пара (i_1, i_2) содержится в $\overline{P}$ и при этом не контролируется, что невозможно. □

Пример. Рассмотрим граф, изображенный на рис. 8.20. Выпишем его матрицу смежности (элементы таблицы с прописными буквами), и, рассматривая эту таблицу как отношение P, транзитивно замкнем его (добавлены элементы со строчными буквами). Найдем максимальное паросочетание (элементы с буквами P и p, а остальные элементы обозначены, соответственно, буквами O и o). Его размер $s = 9$. Дуги этого паросочетания формируют три пути: $I \to A \to B \to C \to D \to G$, $J \to F$ и $E \to H$. Напомним, что это пути в транзитивно замкнутом графе, так что в исходном графе два последних пути должны выглядеть иначе: $J \to A \to F$ и $E \to D \to H$.

Контролирующее множество образуется из $M'_b = \{A, I, J\}$ и $M'_e = \{C, D, G, H\}$. Оставшиеся элементы B, E и F попарно несравнимы.



	A	B	C	D	E	F	G	H	I	J
A		P	o	o	O	O	o	o		
B			P	o			o	o		
C				P			o	o		
D							P	O		
E			O	O			o	p		
F				O			o	p		
G										
H										
I	P	o	o	o	o	o	o	o		
J	O	o	o	o	o	p	o	o		

Рис. 8.20. Покрытие графа путями

Теорема Дилворта имеет важные комбинаторные приложения. Вот одно из них, в котором графы уже даже и не видны.

Следствие. Пусть $a_1, \ldots, a_n$ — произвольная последовательность n чисел. Максимальная длина возрастающей подпоследовательности, содержащейся в ней, равна минимальному количеству невозрастающих подпоследовательностей, на которые можно разложить последовательность $a_1, \ldots, a_n$.

Доказательство следствия основано на построении подходящего графа (какого? придумайте сами) и использовании теоремы Дилворта. □

Глава 9

Экстремальные задачи

9.1. Какие задачи и методы нам уже встречались

Экстремальная задача — это задача, в которой задается некоторое множество A *допустимых решений*, а на этом множестве задается *целевая функция* $f(x)$, и требуется найти такой элемент $x_0 \in A$, на котором значение $f(x_0)$ будет (в зависимости от постановки задачи) наибольшим или наименьшим. Термин *допустимое решение* связан с тем, что обычно A является частью какого-то большего множества "просто решений" и принадлежность решения множеству A требует некоторых усилий. Искомое допустимое решение x_0 называется *оптимальным решением* или *экстремальным решением*.

На протяжении курса нам встретилось довольно много различных экстремальных задач, и прежде чем заняться новым материалом, полезно подвести итог пройденному.

В предыдущей главе нам встретились две пары задач, находившихся в *двойственности*. Первая пара составилась из задач о максимальном паросочетании и минимальном контрольном множестве, вторая, в связи с теоремой Дилворта, из задач о минимальном покрытии путями множества вершин и о максимальном наборе несравнимых вершин.

Теория двойственности представляет собой одно из лучших достижений *математического программирования* — так, к сожалению, принято называть современную теорию экстремальных задач. В геометрических терминах двойственность задач формулируется совсем по-другому, и связь геометрических и комбинаторных постановок нам продемонстрировать трудно. Но вот

простейший пример двойственных задач в евклидовом пространстве в полемической формулировке Л. В. Канторовича *).

Рассмотрим выпуклое ограниченное многогранное множество S и точку a внутри S. Проведем из a луч r и рассмотрим две следующие задачи.

Прямая задача (на максимум): найти на $r \cap S$ точку x, лежащую как можно дальше от a.

Двойственная задача (на минимум): найти плоскость p, не содержащую внутренних точек S и пересекающую r, причем как можно ближе к a.

Здесь совпадение максимума в прямой задаче и минимума в двойственной означает, что точка x, доставляющая максимум в прямой задаче, как раз и будет ближайшей к a точкой пересечения плоскости с лучом. Канторович увидел в этой паре задач неожиданное "экономическое" приложение: коэффициенты плоскости из второй задачи могут рассматриваться как своеобразные экономические показатели, *внутренние цены*. В экономических приложениях эти показатели помогают оценивать различные экономические действия и делать правильный выбор, подобно тому как в задаче о максимальном сочетании нам помогало предконтрольное множество.

Обратим внимание на еще две пары двойственных задач: одна будет подробно рассмотрена в следующем параграфе, а о другой несколько слов написано здесь.

Задача о максимальном потоке. Эта задача приобрела популярность благодаря работам Л. Форда и Д. Фалкерсона и их книге, доступной нам в русском переводе [50].

Рассматривается граф $\langle M, N\rangle$, в котором зафиксированы две вершины — *источник* i_- и *сток* i_+. Каждой дуге графа $j \in N$ сопоставлено положительное число — ее *пропускная способность* d_j. Требуется сопоставить каждой дуге неотрицательный *поток* — число x_j, не превосходящее пропускной способности этой дуги, таким образом, чтобы во всех вершинах, кроме источника и стока, выполнялись условия *баланса* или *неразрывности потока*

$$\delta(i) \triangleq \sum_{j:\mathrm{beg}\ j=i} x_j - \sum_{j:\mathrm{end}\ j=i} x_j = 0$$

и чтобы $\delta(i_-)$, чистый поток, выходящий из источника, достигал наибольшего значения.

*) Канторович, Леонид Витальевич (1912–1986), советский математик и экономист, академик, лауреат Нобелевской премии по экономике. Основные его работы и по математике и по экономике были выполнены, когда он был профессором Ленинградского университета.

Одновременно с этой задачей максимизации рассматривается задача минимизации. Назовем множество дуг $C \subset N$ *разрезом*, если любой путь из источника в сток содержит дугу из этого множества. Величиной разреза назовем сумму пропускных способностей входящих в него дуг. Естественно поставить задачу о поиске разреза с минимальной величиной, и оказывается, что эти две задачи образуют пару: максимум величины потока равен минимуму величины разреза *).

Для решения этих задач было предложено много эффективных методов, из которых особо следует выделить метод Е. А. Диница [1]. Дальнейший обзор литературы — в библиографических рекомендациях.

Несколько раз при решении экстремальных задач мы использовали *динамическое программирование*. Это были задачи о кратчайшем и о критическом пути, задача о наибольшей общей подпоследовательности двух строк, задача о верстке абзаца. Динамическим программированием мы будем заниматься и дальше. Вычислительные схемы динамического программирования разнообразны, и встречавшиеся нам методы Дейкстры и Левита для задачи о кратчайшем пути можно отнести к динамическому программированию.

В некоторых случаях нам удавалось построить *жадные алгоритмы* **) — в них оптимальное решение принималось последовательно, шаг за шагом, и сделанный выбор впоследствии уже не отменялся. В некоторых случаях, когда не удается построить точный алгоритм, который находит в задаче настоящий экстремум, ищут алгоритмы для получения решений "получше" и часто выбирают среди жадных алгоритмов.

В этой главе мы рассмотрим еще некоторые экстремальные задачи, в том числе такие, для которых хороших методов решения пока не найдено.

9.2. Бистохастические матрицы

В этом параграфе изучается один специальный класс матриц, у которого много замечательных комбинаторных свойств. Потребуется несколько новых определений.

*) По-английски эта теорема называется очень красиво: *max-flow-min-cut theorem*.

**) Исходный английский термин greedy означает *жадно глотающий*, — выбор "проглочен", и назад не вернешь. Именно такими были выборы дуг, входящих в кратчайшее остовное дерево в алгоритмах Прима и Краскала.

Рассмотрим в каком-либо многомерном линейном пространстве R точки a_i, $i \in I$, где I — некоторое непустое конечное множество индексов, и образуем линейную комбинацию этих точек $\sum_{i \in I} \lambda_i a_i$. Эта комбинация называется *выпуклой*, если $\lambda_i \geq 0$, $i \in I$, и $\sum_{i \in I} \lambda_i = 1$. Множество всех выпуклых комбинаций точек a_i, $i \in I$, называется их *выпуклой оболочкой* (выпуклая оболочка — это наименьшее выпуклое множество, содержащее данные точки).

Квадратная матрица $a[1:n, 1:n]$ называется *бистохастической*[*], если она состоит из неотрицательных элементов и сумма элементов в каждой строке и каждом столбце равна 1:

$$\sum_{j \in 1:n} a[i, j] = \sum_{j \in 1:n} a[j, i] = 1, \quad i \in 1:n.$$

Назовем *переставляющей матрицей* бистохастическую матрицу с целочисленными элементами. Переставляющая матрица — это матрица, в каждой строке и в каждом столбце которой содержится ровно по одной единице, а остальные элементы равны 0. Название связано с тем, что умножение вектора на такую матрицу осуществляет перестановку элементов этого вектора.

Теорема (Биркгоф–фон Нейман[**]). Любая бистохастическая матрица a представима в виде выпуклой комбинации переставляющих матриц.

Доказательство. Назовем множество пар индексов положительных элементов матрицы a ее *носителем* и обозначим

$$\mathrm{supp}(a) = \{(i, j) \mid a[i, j] > 0\}.$$

Покажем, что матрица a представима в виде

$$a = \lambda \times p + (1 - \lambda) \times a_1,$$

где p — переставляющая матрица, a_1 — другая бистохастическая матрица, λ — коэффициент, лежащий между 0 и 1. При этом либо $\lambda = 1$,

[*] Греческое слово *στοχασις* означает "догадка". Набор неотрицательных элементов, в сумме равных 1, ассоциируется с распределением вероятностей. И действительно, в специальном классе вероятностных процессов, о которых речь пойдет дальше, большую роль играют *стохастические* матрицы, в которых такое ограничение накладывается только на суммы по строкам. Бистохастические матрицы — это разновидность обычного для теории вероятностей объекта, имеющая свои замечательные математические свойства.

[**] Garrett Birkhoff (p. 1911), американский математик. Имя фон Неймана нам уже встречалось. Примечательно, что такая простая теорема появилась только в 1944 г.

либо матрица a_1 имеет меньший носитель, чем a:

$$\operatorname{supp}(a_1) \subset \operatorname{supp}(a), \qquad \operatorname{supp}(a) \setminus \operatorname{supp}(a_1) \neq \varnothing.$$

Образуем матрицу $\chi_a[1:n, 1:n]$, у которой элемент $\chi_a[i, j]$ равен 1, если $(i, j) \in \operatorname{supp}(a)$, и 0 в противоположном случае (так что χ_a — это своего рода характеристический вектор множества $\operatorname{supp}(a)$, представленный в виде матрицы). По следствию 3 к теореме о паросочетаниях существует такое отображение $\varphi : 1:n \to 1:n$, что $\chi_a[i, \varphi(i)] = 1$ для всех $i \in 1:n$. Действительно, если такого отображения не существует, то найдутся множества строк K и столбцов L, для которых $|K| + |L| > n$, и матрица $\chi_a[K, L]$ целиком состоит из нулей. Но тогда состоит из нулей и матрица $a[K, L]$. А это невозможно из-за особенностей бистохастических матриц, так как

$$\sum_{i \in K} \sum_{j \in 1:n \setminus L} = \sum_{i \in K} \sum_{j \in 1:n} = |K|,$$

и вместе с тем

$$\sum_{i \in K} \sum_{j \in 1:n \setminus L} \leq \sum_{i \in 1:n} \sum_{j \in 1:n \setminus L} = n - |L|,$$

и следовательно, $|K| \leq n - |L|$, что противоречит выбору этих множеств.

Имеющееся отображение φ и определяет искомую переставляющую матрицу p: $p[i, \varphi(i)] = 1$. Выберем теперь

$$\lambda = \min\{a[i, j] \mid (i, j) \in \operatorname{supp}(p)\} = \min\{a[i, \varphi(i)] \mid i \in 1:n\}.$$

Очевидно, что $0 \leq \lambda \leq 1$. Если $\lambda = 1$, то сама матрица a переставляющая. Если же $\lambda < 1$, то матрица $a - \lambda \times p$ неотрицательна, сумма ее элементов в каждой строке и каждом столбце равна $1 - \lambda$ и ее носитель меньше носителя a хотя бы на один элемент (тот или те, на которых достигается минимум в определении λ). Матрица a_1, которую мы строим, получается из $a - \lambda \times p$ умножением всех ее элементов на $(1 - \lambda)^{-1}$.

Итак, несколько меняя обозначения, получаем

$$a = a_0 = \lambda_1 \times p_1 + (1 - \lambda_1) \times a_1.$$

В той же форме можно представить и матрицу a_1:

Повторим это разложение до полного исчерпания носителя в некоторой матрице a_k. Суммирование получающихся равенств с нужными множителями и дает

$$a = \lambda_1 \times p_1 + (1-\lambda_1)\lambda_2 \times p_2 + \cdots + (1-\lambda_1) \times \cdots \times (1-\lambda_{k-1}) \times p_k.$$

□

Во многих оптимизационных задачах множество бистохастических матриц появляется как основное множество допустимых решений. Наиболее интересна среди этих задач *задача о назначениях*, которая формулируется следующим образом.

Задача о назначениях. Заданы два конечных множества K и L и матрица $a[K, L]$ (обычно предполагается, что $|K| = |L| \triangleq m$). Найти взаимнооднозначное соответствие $\varphi : K \to L$, доставляющее минимум величине

$$\sum_{k \in K} a[k, \varphi(k)].$$

Отображение φ *назначает* каждому элементу из K его "пару" из L, так что речь идет о построении паросочетания с наименьшей суммой "весов" выбранных пар. ⊠

Описываемый далее метод решения этой задачи был предложен американским математиком Х. Куном *). Кун использовал теорию паросочетаний, известную ему по работам венгерских ученых Денеша Кёнига (1884–1944) и Эйгена Эгервари (1891–1958) и назвал поэтому свой алгоритм *венгерским методом*.

В основе метода лежит возможность изменения матрицы a прибавлением постоянного слагаемого ко всем элементам какой-нибудь строки (и аналогично столбца). Действительно, в каждом допустимом решении участвует ровно один элемент из этой строки, и при таком добавлении к значению целевой функции прибавится это же слагаемое, одно и то же для всех допустимых решений. И оптимальное решение останется оптимальным, только минимальное значение целевой функции изменится.

Используя эту возможность, введем снова операцию *спуска до нуля*, которая нам встречалась в задаче о кратчайшем дереве путей.

Спустим до нуля каждую строку матрицы a, а затем и каждый столбец. В результате мы получим неотрицательную матрицу, у которой

*) Harold Kuhn, 1955. Историю венгерского метода Кун описал в [74].

в каждой строке и в каждом столбце есть нулевые элементы. Обозначим получившуюся матрицу через a_0 и рассмотрим двудольный граф, соответствующий нулевым элементам матрицы, со строками-началами дуг и столбцами-концами дуг.

Очевидно, что если нам удастся построить в этом графе полное паросочетание, то мы получим соответствие строк и столбцов матрицы, т.е. допустимое решение задачи о назначениях, причем сосредоточенное на нулевых элементах матрицы, и оптимальность решения гарантируется тем, что из-за неотрицательности элементов матрицы значение меньше нуля не получить.

Алгоритм для построения максимального паросочетания нам уже известен, нужно определить, что делать по окончании его работы.

Если максимальное паросочетание окажется полным, задача уже решена.

Если же нет, то построенное вместе с паросочетанием минимальное контролирующее множество (K', L'), $|K'| + |L'| < m$, поможет нам уменьшить значение целевой функции дополнительной редукцией.

Именно, по определению контролирующего множества для всех нулевых элементов $a_0[k, l] = 0$ имеем либо $k \in K'$, либо $l \in L'$. Следовательно, все элементы подматрицы $a_0[K \setminus K', L \setminus L']$ положительны.

Пусть $\delta = \min\{a_0[k, l]\}$ в этой подматрице. Вычтем δ из каждой строки $K \setminus K'$ и прибавим к каждому столбцу из L'. После этих изменений матрица останется неотрицательной, так как ее значения уменьшились на δ на $a_0[K \setminus K', L \setminus L']$ (что к отрицательным значениям не приведет), а в остальных частях матрицы не уменьшались. Значение же целевой функции увеличится на $\delta \cdot |L'|$ и уменьшится на $\delta \cdot |K \setminus K'|$, и так как $|K| = m$, то в итоге оно уменьшится на $\delta \cdot (m - |K'| - |L'|)$.

Эти действия выполняются до тех пор, пока не будет построено полное паросочетание.

Осталось убедиться, что этот алгоритм завершается за конечное число шагов, и оценить его трудоемкость. Для оценки числа итераций тут используется тот же прием, что и в лемме для задачи о максимальном паросочетании на с. 248, с размером максимального паросочетания, равным m. А в остальном все рассуждение остается буквально тем же, так как на каждом шаге рассмотренного в лемме процесса, от одного пересчета матрицы до другого, построение паросочетания начинается с построенного на предыдущем шаге.

Пример. Рассмотрим небольшой пример. Пусть матрица задана левой таблицей. Опуская ее строки до нуля, получаем правую таблицу:

	1	2	3	4	5	6	7	8
a	32	11	19	18	44	65	23	18
b	41	54	23	19	87	16	25	33
c	47	34	41	26	15	47	29	52
d	73	14	10	0	12	29	33	50
e	37	33	18	29	26	26	41	84
f	72	61	38	96	26	14	55	47
g	38	17	26	49	28	91	97	24
h	32	52	67	71	33	56	54	22

	1	2	3	4	5	6	7	8
a	21	0	8	7	33	54	12	7
b	25	38	7	3	71	0	9	17
c	32	19	26	11	0	32	14	37
d	73	14	10	0	12	29	33	50
e	19	15	0	11	8	8	23	66
f	58	47	24	82	12	0	41	33
g	21	0	9	32	11	74	80	7
h	10	30	45	49	11	34	32	0

Затем, опуская до нуля столбцы, получаем матрицу приведенных стоимостей и матрицу нулевых элементов

	1	2	3	4	5	6	7	8
a	11	0	8	7	33	54	3	7
b	15	38	7	3	71	0	0	17
c	22	19	26	11	0	32	5	37
d	63	14	10	0	12	29	24	50
e	9	15	0	11	8	8	14	66
f	48	47	24	82	12	0	32	33
g	11	0	9	32	11	74	71	7
h	0	30	45	49	11	34	23	0

	1	2	3	4	5	6	7	8
a		I						
b						+	−	
c					−			
d				−				
e			−					
f						−		
g		+						
h	−							+

Здесь знаками − и I выделены элементы, входящие в максимальное паросочетание, соответственно, в N'_b или в N'_e. Контролирующее множество состоит из вершин $M'_b = \{b, c, d, e, f, h\}$ и $M'_e = \{2\}$. Минимальный элемент вне контроля (т.е. в строках a и g и всех столбцах, кроме 2) — это $a[a, 7] = 3$. Вычитая этот минимум из всех элементов строк a и g и прибавляя его ко всем элементам столбца 2, получаем новую матрицу и новое паросочетание — нет! то же самое, но с новым контролирующим множеством $M'_b = \{c, d, e, h\}$ и $M'_e = \{2, 6, 7\}$.

Снова находим минимальный неперечеркнутый элемент, это $a[b, 4] = 3$, и преобразуем матрицу, прибавляя 3 к контролируемым столбцам (уже появляется какой-то жаргон) и вычитая из контролируемых строк. Теперь

	1	2	3	4	5	6	7	8
a	5	0	2	1	27	51	0	1
b	12	41	4	0	68	0	0	14
c	22	25	26	11	0	35	8	37
d	63	20	10	0	12	32	27	50
e	9	21	0	11	8	11	17	66
f	45	50	21	79	9	0	32	30
g	5	0	3	26	5	71	68	1
h	0	36	45	49	11	37	26	0

	1	2	3	4	5	6	7	8
a							+	
b				+		+		
c					−			
d								
e			−					
f								
g		+						
h	−							+

и $M'_b = \{c, e, h\}$ и $M'_c = \{2, 4, 6, 7\}$. Минимальный неперечеркнутый элемент (их несколько) равен 1. Найдите окончательное паросочетание и наилучшее назначение самостоятельно.

Упражнение 9.1. Разработать формальное описание венгерского алгоритма, используя для выполнения спусков до нуля массивы "вычитаемых" по строкам и столбцам и функцию вычисления измененного значения элемента массива типа (используем Паскаль)

```
function aReduced(i,j: integer): cost;
begin
  aReduced := a[i,j] - uRow[i] - uCol[j];
end;
```

9.3. Экстремальные задачи на множестве перестановок

Экстремальные задачи на множестве перестановок легко формулируются и очень привлекательны. В частности, и задача о назначениях имеет несколько вариантов, которые, несмотря на близость формулировок, оказались несравнимо более сложными. Представим здесь вкратце две из них, наиболее знаменитые.

Задача о бродячем торговце[*]. Торговец должен обойти m городов, побывав в каждом из них по одному разу и вернувшись в тот город, с которого начал. В какой последовательности обходить ему города, чтобы длина суммарного пути была наименьшей? ⊠

[*] Travelling salesman problem (англ.) переводится еще как *задача о коммивояжере*.

Формализуем постановку задачи. Пусть задано множество городов M и матрица расстояний между городами $c[M, M]$ (не обязательно симметричная). Требуется найти последовательность обхода городов $i = \{i_0, i_1, \ldots, i_m = i_0\}$, где $i_k \in M$, $k \in 1 : m$, $m = |M|$, в которую каждый город входил бы по одному разу, с минимальной суммой длин переходов

$$C(i) = \sum_{k \in 1:m} c[i_{k-1}, i_k].$$

Таких допустимых обходов насчитывается ровно $(m-1)!$, поскольку каждый обход — это одноцикловая перестановка, их число равно числу всех возможных перестановок из $(m-1)$ элемента. Поясним это на примере: набор (2, 6, 4, 1, 5, 3) можно рассматривать и как перестановку из 6 элементов в обычной записи, и как замкнутый путь $0 \to 2 \to 6 \to 4 \to 1 \to 5 \to 3 \to 0$, соответствующий одноцикловой перестановке из 7 элементов (2, 5, 6, 0, 1, 3, 4).

Казалось бы, мы имеем такое же множество допустимых решений, как в задаче о назначениях. Но свойства целевой функции настолько различны, что надежды на эффективный точный алгоритм пропадают *).

Вторая задача имеет то же множество состояний, но еще труднее для решения.

Квадратичная задача о назначениях. Рассматривается множество городов M и множество заводов P, которые можно построить в этих городах, по одному заводу в каждом городе. Мощности обоих множеств равны. Заданы стоимости перевозки тонны груза из любого города в любой $c[M, M]$ и объемы перевозки между каждыми двумя заводами $w[P, P]$. Требуется определить, как выгоднее всего разместить заводы для минимизации затрат на перевозки. ☐

Итак, при заданных матрицах $c[M, M]$ и $w[P, P]$ требуется найти биекцию $\varphi : M \to P$, на которой достигается минимум суммы

$$\sum_{i,j \in M} c[i, j] \times w[\varphi(i), \varphi(j)].$$

*) Отметим сразу же, что эта задача очень редко встречается в приложениях, а если и встречается, то в нахождении абсолютно минимального пути потребности нет. Например, речь может идти о выборе порядка, в котором сверлильный автомат обходит в изделии несколько тысяч точек, и требуется минимизировать суммарное время перехода между позициями. Для этой задачи есть очень хорошие приближенные методы, а речь сейчас идет о точном алгоритме.

Полюбуйтесь, какая простая формулировка, простое множество допустимых решений — все то же множество перестановок, какие скромные размеры исходных данных — две квадратные матрицы. Между тем найти точное решение этой задачи не удается уже при $|M| \approx 15$ *).

9.4. Методы улучшенного перебора

Важным методом решения дискретных экстремальных задач, в частности экстремальных задач, связанных с терминологией теории графов, является метод *неявного* перебора, или *улучшенного* перебора.

Почему "улучшенный"? Возможность обычного перебора при решении дискретной экстремальной задачи ясна всем: есть у нас, скажем, $15! \approx 1.307 \times 10^{12}$ решений, переберем их все за конечное время и отыщем наилучшее решение. Наивно надеяться на эффективность этого метода — число допустимых решений растет с ростом параметров задачи очень быстро.

Среди возможных выходов из положения — улучшение перебора отбрасыванием заведомо неоптимальных решений. Немыслимо отбрасывать отдельные решения, такая схема не может быть эффективной, нужно отбрасывать целые подмножества.

Первая эффективная схема такого рода была предложена американскими авторами **).

Рассмотрим в общем виде дискретную экстремальную задачу.

Пусть задано дискретное множество A и определенная на нем целевая функция $f(x)$. Найти элемент $x_0 \in A$, на котором достигается минимум этой функции. Обозначим минимум функции f на множестве B через $F(B)$.

Начнем с одного очень простого замечания. Пусть множество A разбито на несколько непересекающихся подмножеств:

$$A = A_0 \cup A_1 \cup \cdots \cup A_k, \quad A_i \cap A_j = \varnothing, \ i \neq j,$$

*) Я не рекомендую вам начинать свою научную карьеру с этой задачи. Во всяком случае, полезно ознакомиться с тем, что уже сделано предшественниками.

Позволю себе воспоминание: когда-то и у меня была надежда, что я могу как-то продвинуться в этой задаче. Мне посчастливилось встретиться с одним из ее авторов, Тьяллингом Купмансом (Tjalling Koopmans, 1910–1985), вместе с Л. В. Канторовичем удостоенным в 1975 г. Нобелевской премии по экономике. Я сказал ему о моих надеждах (не сбывшихся) и получил в ответ только фразу: "Это единственная в моей жизни задача, которую я опубликовал без решения".

**) J. Little, K. Murty, D. Sweeney, C. Karel, 1963. Мы ссылаемся в библиографии на русский перевод [34]. Именно эти авторы, решая задачу о бродячем торговце, ввели ставшее широко известным название "метод ветвей и границ".

причем множество A_0 минимальных элементов не содержит. Тогда легко видеть, что

$$F(A) = \min\{F(A_i) \mid i \in 1:k\}.$$

И еще одно простое замечание: пусть имеется функция φ, заданная на подмножествах множества A, такая что $\varphi(B) \le F(B)$ для любого $B \subset A$. Пусть $\overline{x}$ — какой-либо элемент множества A и $\overline{f} = f(\overline{x})$. Тогда

$$F(A) = \min\{\overline{f},\ \min\{F(A_i) \mid i \in 1:k,\ \varphi(A_i) \le \overline{f}\}\}.$$

Эти два соображения позволяют предложить следующую технологию поиска минимума.

Разобьем множество A на какие-либо подмножества A_i и на каждом из них найдем нижнюю оценку φ. Если при этом будут встречаться элементы множества A и для них будут вычисляться значения функции f (что, в общем, не обязательно), то мы запомним тот элемент, для которого значение f — наименьшее. Все подмножества, у которых оценка выше рекордного значения функции $\overline{f}$, объединим в подмножество A_0 (чтобы в дальнейшем не рассматривать).

Теперь выберем какое-либо из множеств A_i, $i > 0$. Разобьем это множество на несколько более мелких подмножеств таким образом, чтобы для каждого из них сохранялась возможность вычислить оценку φ. При этом мы будем продолжать улучшать рекордное значение $\overline{f}$, сравнивая его со всеми встречающимися вновь вычисленными значениями f. Этот процесс повторяется до тех пор, пока не будут просмотрены все множества A_i, $i > 0$.

В нашей формулировке много произвола, и эффективность метода существенно зависит от того, насколько удачно выбраны эти произвольные параметры метода. Два вопроса оказываются тесно связанными друг с другом: способ оценки снизу минимума целевой функции и способ разбиения подмножества на более мелкие подмножества. Их согласование заключается в том, что способ оценки должен быть таким, чтобы его можно было применить ко всем получающимся подмножествам.

Третий произвольный параметр метода — порядок выбора подмножеств, подлежащих разбиению. От этого порядка зависит трудоемкость метода, измеряемая и в количестве итераций, и в объеме необходимой памяти компьютера, и в трудоемкости самих действий. Обсуждением этих вопросов мы займемся позднее.

Применим теперь идеи неявного перебора к одной специфической дискретной задаче, на которой мы сможем увидеть, как все эти параметры выбираются в конкретных условиях.

Задача о размыкании контуров. Пусть задан граф*) $\langle M, N\rangle$, каждой дуге j которого сопоставлено положительное число $w[j]$ — вес этой дуги. Требуется найти такое подмножество N_0, чтобы граф $\langle M, N_0\rangle$ не имел контуров и сумма весов дуг, содержащихся в N_0, была максимальной. ⊠

Полезно наряду с этой формулировкой использовать и такую, в которой речь идет об удалении из множества N подмножества дуг с минимальным суммарным весом так, чтобы разомкнулись все контуры графа.

Введем вспомогательную задачу о размыкании контуров, в которой кроме графа $\langle M, N\rangle$ введен еще один параметр — задано множество дуг $N' \subset N$, которые удалять нельзя (естественно при этом требовать, чтобы в графе $\langle M, N'\rangle$ контуров не было). Эта задача будет полезна при введении разбиений множества допустимых решений.

Именно, представим себе, что мы решаем вспомогательную задачу с графом $\langle M, N\rangle$ и множеством неудаляемых дуг $N' \subset N$ (назовем ее задачей (N, N')). Выберем какую-либо дугу $j \in N \setminus N'$, такую что в графе $\langle M, N' \cup \{j\}\rangle$ нет контуров. Тогда все множество решений задачи (N, N') разбивается на множество решений задачи $(N \setminus \{j\}, N')$ и множество решений задачи $(N, N' \cup \{j\})$. Иначе говоря, при решении вспомогательной задачи мы можем выбросить дугу j или запретить ее выбрасывать, и в обоих случаях снова получится задача той же структуры. Вот это свойство сохранения структуры задачи нам и важно.

Оценка минимума снизу будет строиться с использованием рекурсии для всех вспомогательных задач (N, N'). Очевидно, наша исходная задача — это задача $(N, \varnothing)$. Для построения оценки мы используем три простые леммы.

Лемма 1. Пусть задан граф $\langle M, N\rangle$ с заданными положительными весами дуг $w[N]$ и выделенным в N подмножеством N_0, таким что граф $\langle M, N_0\rangle$ контуров не имеет. Пусть $v(N, N_0)$ — минимум суммы весов дуг, удаление которых из $N \setminus N_0$ размыкает все контуры графа. Пусть $N' \subset N$. Тогда $v(N, N_0) \geq v(N', N_0)$.

*) Для простоты мы убрали отображение T из списка компонент описания графа

Доказательство. Действительно, если K — множество дуг, удаление которых из N размыкает все контуры графа $\langle M, N\rangle$ с минимальным весом, то граф $\langle M, N' \setminus K\rangle$ также не имеет контуров, удаление K из N' может обойтись еще дешевле, и в минимальном решении сумма удаляемых весов может быть еще меньше. □

Лемма 2. В обозначениях леммы 1 если $N_0 \subset N_1$, то

$$\upsilon(N, N_1) \geq \upsilon(N, N_0).$$

Доказательство. Множество K, на котором достигается минимум в задаче для N_1, размыкает все контуры и в случае N_0; а минимальное для N_0 решение может быть еще экономнее. □

Лемма 3. В обозначениях леммы 1 пусть N_c — дуги некоторого цикла, $w_c = \min\{w[u] \mid u \in N_c \setminus N_0\}$ (минимальный вес, которым можно разомкнуть цикл c). Тогда $\upsilon(N, N_0) \geq \upsilon(N \setminus N_c, N_0) + w_c$.

Доказательство. Пусть K — минимальное размыкающее множество N и u_c — какая-либо дуга из N_c, содержащаяся в K. Тогда

$$\upsilon(N, N_0) = \upsilon(N \setminus \{u_c\}, N_0) + w[u_c] \geq \upsilon(N \setminus N_c, N_0) + w_c$$

(по лемме 1 уменьшение множества дуг разве лишь уменьшает оценку, а минимум w_c по определению не превосходит веса любой дуги u_c). □

Алгоритм оценки минимума. Введем функцию $est(N, N_0)$ следующим образом:

1) если граф $\langle M, N\rangle$ не содержит циклов, примем $est(N, N_0) = 0$;

2) если он содержит циклы, найдем цикл N_c и в соответствии с леммой 3 примем

$$est(N, N_0) = est(N \setminus N_c, N_0) + w_c.$$

На основе лемм 1–3 получается следующее соотношение между функциями υ и est, оправдывающее выбранное название для est (от слова estimate — оценка).

Лемма 4. $\upsilon(N, N_0) \geq est(N, N_0)$.

Доказательство. Пусть есть последовательность преобразований, переводящих граф $\langle M, N\rangle$ в граф без контуров, и порожденная этими преобразованиями последовательность множеств дуг $N_1, \ldots, N_r$. Для этой последовательности имеем по лемме 3:

и по определению:

$$est(N_k) = est(N_{k+1}) + w_{ck}$$

при $k < r$. Суммируя в первом случае неравенства, а во втором — равенства, легко получаем утверждение леммы. □

На основании этих лемм можно строить вычислительный процесс "неявного перебора". Для задания процесса нужно определить, что представляет собой его состояние, каково начальное состояние, каковы правила перехода из одного состояния в другое, включая правило остановки. В данном случае все выглядит так:

Состоянием процесса является набор пар множеств $\{(N^k, N_0^k)\}_{k \in 1:t}$. При этом множества N_0^k таковы, что граф $\langle M, N_0^k \rangle$ контуров не содержит. Каждой паре (N^k, N_0^k) сопоставляется оценка $est(N^k, N_0^k)$ и сумма весов удаленных дуг $W(N^k) = \sum_{j \in N \setminus N^k} w_j$.

Начальное состояние процесса. Набор состоит из одной пары $(N, \varnothing)$, с вычисленной для этой пары оценкой и нулевой суммой весов удаленных дуг.

Правила перехода.

1. Если в имеющемся наборе не осталось ничего, закончить процесс.
2. Выбрать в имеющемся наборе пару множеств $\{(N^{k_0}, N_0^{k_0})\}$ с наименьшей оценкой. Во множестве $N^{k_0} \setminus N_0^{k_0}$ выбрать дугу j_0 — рекомендуется взять одну из дуг, входящих в оценку данной пары множеств.
3. Рассмотреть две новые пары множеств. Первая получается просто удалением дуги j_0 из множества N^{k_0} с соответствующим изменением суммы весов удаленных дуг и с пересчетом оценки. В случае если граф с множеством дуг N^{k_0} не имеет контуров (оценка равна 0), получающаяся пара не включается в набор, но ее сумма весов удаленных дуг сравнивается с рекордной, и, если нужно, рекордное решение обновляется.
4. Вторая пара множеств получается включением дуги j_0 в множество $N_0^{k_0}$. При этом граф $\langle M, N_0^{k_0} \rangle$ полезно "переработать", выполнив все действия, очевидным образом следующие из сделанных решений, а именно исключить из N^{k_0} дуги, замыкающие контуры в этом графе, и включить в $N_0^{k_0}$ дуги, входящие в транзитивное замыкание графа $\langle M, N_0^{k_0} \rangle$. Действительно, если в этом графе уже есть путь, ведущий из вершины i_1 в вершину i_2, то

дуга, ведущая из i_1 в i_2, уже не должна удаляться из графа, а встречная дуга должна быть удалена.

5. Удалить из набора пары с оценкой, большей рекорда.

Задача о размыкании контуров встречается в ситуациях, когда требуется выбрать непротиворечивое упорядочение некоторого множества объектов (такие задачи встречаются в статистике)*).

Что же касается метода ветвей и границ, то он является очень удобным и общим средством для решения многих дискретных задач. Но нужно понимать, что эта схема еще слишком обща и слишком неопределенна, и в любом конкретном приложении она нуждается в дополнительном определении деталей (я привел еще не все детали даже для задачи о размыкании контуров).

9.5. Приближенные методы оптимизации

Не думайте, что метод ветвей и границ универсален. Можно назвать многие задачи, даже очень просто формулируемые, в которых применить его эффективно не удается. В задаче о бродячем торговце метод ветвей и границ при удачной реализации оказался эффективным, а в квадратичной задаче о назначениях, как уже отмечалось, его не удается успешно применить даже для размерности 15.

По-видимому, эффективных и одновременно универсальных способов для решения дискретных оптимизационных задач не существует. Но, даже если бы они и существовали, не всегда затраты на поиск оптимального решения окупаются выгодами от его применения, и в практических ситуациях нужно быстро выбирать решение "пооптимальнее". Поэтому наряду с точными методами, обеспечивающими настоящий экстремум, развиваются разнообразные приближенные методы, в которых поиск тем или иным способом ограничивается**).

Рассмотрим несколько таких приближенных методов.

*) Вспомнив о необходимости размыкания контуров при составлении сетевого графика, вы можете подумать и о таком применении обсуждаемой экстремальной задачи. Но это хороший пример ситуации фальшивой применимости модели — никакие искусственные "выгоды" не могут заменить поиска правильной последовательности выполнения работ.

**) Некоторые авторы различают собственно *приближенные* методы, в которых удается оценить или сознательно регулировать степень приближения к решению (как это делалось при решении задачи о минимальном числе инвертирований перестановки на с. 44), и *эвристические* методы, в которых такое регулирование невозможно. Ранее уже давалось другое объяснение термина "эвристический".

9.5.1. Метод локальных улучшений

Идея этого метода заключается в том, что для каждого решения экстремальной задачи $x \in X$ определяется каким-то образом *окрестность близких решений* $A(x)$ и на каждой итерации вычислительного процесса при заданном текущем решении x делается попытка найти в его окрестности решение, которое имело бы лучшее значение целевой функции. Если такое решение удается найти, оно само становится текущим решением, если нет, то поиск заканчивается. По аналогии с непрерывными экстремальными задачами полученное решение естественно назвать *локальным экстремумом*. Локальный экстремум находится просто почти всегда, математически трудно найти глобальный экстремум, или найти локальный экстремум и оценить его отклонение от ненайденного глобального.

Рассмотрим важный пример метода локальных улучшений — упрощение представления логической функции.

Пример (минимизация ДНФ). Вспомним задачу о минимизации представления логической функции, которую мы рассматривали в начале курса (с. 20). Логическая функция задана в виде *совершенной дизъюнктивной нормальной формы* и требуется представить ее в виде ДНФ с минимальным числом вхождений переменных. Будем прямо описывать задачу и метод в терминах граней единичного куба: множество вершин куба A задано перечислением вершин, и эти вершины нужно "склеить" в целые грани.

Мы начнем с какого-либо представления множества A в виде объединения граней и будем его последовательно преобразовывать. Преобразование предполагает существование некоторого набора допустимых операций, которые могут применяться последовательно.

Введем вначале одну операцию — *склейку граней*.

Если две грани заданы одним и тем же множеством индексов I с наборами координат y и y', отличающимися только на одном индексе $i_1 \in I$, то эти грани могут быть заменены одной гранью, соответствующей набору $I \setminus \{i_1\}$ (с общим, совпадающим, набором координат).

Уже одна эта операция позволяет заметно упрощать ДНФ. Однако, например, функцию

$$f(x_1, x_2, x_3) = x_1x_2x_3 \vee x_1\overline{x_2}x_3 \vee x_1x_2\overline{x_3}$$

применением склейки граней можно привести только к

$$f(x_1, x_2, x_3) = x_1x_3 \vee x_1x_2\overline{x_3}$$

или

$$f(x_1, x_2, x_3) = x_1x_2 \vee x_1\overline{x_2}x_3,$$

тогда как справедливо также представление

Этот пример наводит на мысль о полезности следующих операций.

Поглощение. Если в представлении множества A грань S_1 содержится в грани S_2, то грань S_1 может быть из представления удалена.

Расширение грани. Если в представлении множества A внутри грани S_1 содержится грань S_1', которая может быть склеена с некоторой гранью S_2, то грань S_2 может быть заменена результатом склеивания. (В только что рассмотренном примере нужна именно эта операция.)

Устранение дублирования. Если все вершины какой-либо грани S содержатся в других гранях, то грань S можно из представления множества исключить.

Формулы, дальнейшие сокращения которых невозможны, называются *тупиковыми*.

Упражнение 9.2. Предложите метод локальных улучшений для задачи о бродячем торговце и для квадратичной задачи о назначениях.

Использование большего списка операций позволяет улучшить конечный результат. Если этого улучшения окажется недостаточно, то часто могут быть предложены другие операции, а окрестность возможных изменений может быть увеличена. Метод локальных улучшений является наиболее естественным, и с него обычно и начинаются попытки построения приближенного оптимизационного метода.

9.5.2. Случайный поиск

Другой подход к приближенной оптимизации — *случайный поиск*. Обычно выбор решения можно представить как последовательность выборов. Если выбирать элементарные решения с помощью какого-либо случайного механизма, то одно допустимое решение часто строится очень быстро, так что можно находить решение многократно и запоминать "рекорд", т.е. наилучшее из встретившихся решений. Этот наивный подход существенно улучшается, когда удается учесть в случайном механизме перспективность тех или иных локальных выборов. Мне известно об успешном применении такого подхода в очень серьезных практических задачах, например в задачах составления расписаний для Аэрофлота (1970-е годы).

Очень эффективным оказывается сочетание случайного поиска с локальными улучшениями — каждое допустимое решение, выбранное

случайно, затем улучшается, как в методе локальных улучшений. Таким образом, в алгоритме как бы выбираются точки из *областей притяжения* различных локальных экстремумов.

Интересный вариант сочетания этих двух методов называется *методом моделирования отжига*[*]. Отжигом называется технологический процесс, в котором металлическое изделие сильно нагревается, а затем медленно охлаждается. При этом охлаждении происходит перестройка кристаллических связей, и многие нарушения связей устраняются. Применительно к поиску оптимального решения медленное охлаждение соответствует шагам локального улучшения, а нагрев — случайному переходу в какую-либо другую точку с худшим значением целевой функции.

Нужно упомянуть также *метод табу*[**], предлагаемый в качестве надстройки над другими методами. Метод табу начинает работу с поиска локального экстремума, скажем минимума. Заключительная часть траектории поиска записывается в специальный список табу, и сохраненные списки используются для запрещения переходов в противоположном направлении.

В последние годы приобретают все большую популярность *генетические алгоритмы*, называемые так из-за ассоциации с предполагаемыми процессами формирования больших биологических молекул. Представьте себе среду, состоящую из фрагментов допустимых решений (можно назвать такую среду *первичным бульоном*[***]). Отдельные фрагменты в этом бульоне случайно встречаются, и если они хорошо компонуются в фрагмент большего размера, то соединяются. Фрагменты, которые ни с чем не компонуются, разваливаются. Возможны и другие способы рекомбинирования фрагментов, например инверсии их отдельных кусков или *кроссинговер* — обмен кусками между двумя фрагментами. Набор способов рекомбинации в конкретной задаче оптимизации может ограничиваться или пополняться другими, более удобными для нее.

---

[*] Simulated annealing.

[**] Tabu search method.

[***] Термин *первичный бульон* (англ. primordial soup) был введен русским биохимиком А. И. Опариным в 1923 г. при построении гипотезы о происхождении жизни. Так он называл вместилище (океан) предбиологических форм сложного химического состава, из которых при хаотическом комбинировании и возникли простейшие живые формы.

Таким первичным бульоном для зарождающегося информационного общества некоторые сейчас считают и Интернет.

Много более скромно выглядит создание первичного бульона в оптимизационных задачах.

Например, в задаче о бродячем торговце первоначально компот может содержать элементарные переходы из города в город, стыковаться могут два фрагмента, у которых нет общих городов кроме того, что конечный город одного фрагмента совпадает с начальным городом другого, а вероятность того, что фрагмент распадется, увеличивается с ростом длины фрагмента.

Часто рассматриваются такие версии алгоритма, в которых вместо бульона хранится совокупность полноценных допустимых решений задачи. Они составляют *популяцию*. В ней выбираются и скрещиваются отдельные "индивидуумы", порождающие (при возможных случайных мутациях) новые допустимые решения, лучшие из которых заменяют "кого-то" в популяции. Эти алгоритмы отличаются большой свободой выбора правил моделирования динамики популяций.

9.5.3. Эвристические методы

Этот подход мы уже упоминали в связи с задачами составления расписаний. В нем для выбора элементов решения используются те или иные, кажущиеся естественными, рекомендательные правила выбора, *эвристики*. Иногда говорят, что алгоритм "моделирует логику диспетчера". Часто такие правила комбинируются с условием *жадности* выбора: сделанный выбор в дальнейшем не пересматривается.

Вот простая экстремальная задача, на которой можно увидеть и эвристический метод, который является жадным, и метод локальных улучшений.

Задача о куче камней. Пусть задано множество положительных чисел w_j, $j \in N$, где $N = 1:n$ — некоторое конечное множество индексов, и натуральное число k, $k < |N|$. Рассматриваются разбиения множества N на k подмножеств N_i, $i \in 1:k$, для каждого подмножества N_i вычисляется его "вес"

$$W(N_i) = \sum_{j \in N_i} w_j,$$

а оценкой разбиения является наибольший из весов его элементов. Требуется найти разбиение с наименьшей оценкой. (Можно представить себе набор камней, которые нужно разложить в n куч по возможности одинакового веса.)

Предлагается, упорядочив веса по убыванию, последовательно просматривать их и определять для каждого j-го веса, к какому из под-

множеств его присоединять. Достаточно эффективным оказалось эвристическое правило: присоединять этот вес к множеству с наименьшим накопленным весом. Примерно так

```
for i:= 1 to k do W[i] := 0 od;
for j:= 1 to n do
  <найти индекс iMin с наименьшим значением W[i]>
  W[iMin] := W[iMin] + w[j]; index[j] := iMin
od
```

и искомое разбиение записано в массиве `index`.

В случае если разбиение оказывается все же недостаточно хорошим, можно пытаться улучшить его локальными изменениями, стараясь найти в разных элементах разбиения группы весов, перестановка которых может уменьшить оценку разбиения.

Эвристические методы особенно распространены и эффективны, когда решения нужно принимать в меняющейся ситуации и заранее неизвестно, как обстановка будет изменяться в дальнейшем. В таких случаях удобно, если можно сформулировать какое-то правило.

9.5.4. Сокращенный поиск

Более мощной разновидностью такого подхода является *сокращенный поиск*, в котором дерево вариантов, знакомое нам по методу ветвей и границ, искусственно сокращается исходя из некоторых правил, правдоподобных, но формально не обоснованных.

Глава 10

Процессы

Описывая конкретные алгоритмы, мы уже несколько раз говорили о том, что вычисление по алгоритму можно рассматривать как некоторый *процесс*, который описывается своим *множеством состояний*, *начальным состоянием* и *правилами перехода из состояния в состояние*. О множестве состояний часто ничего специфического не скажешь — обычно это конечное (или бесконечное) множество. Что же касается правил перехода, то в различных областях математики принимаются правила, различные по самой своей природе. Мы начнем с того, что рассмотрим три варианта таких правил и, соответственно, три принципиально различных процесса.

- Процесс, в котором переходы выполняются под влиянием неких внешних воздействий. Этот процесс называется *автоматом*.
- Процесс, в котором переходы выполняются под влиянием некоего случайного механизма. Таких процессов можно придумать много, простейший из них, который мы и будем рассматривать, называется *марковской цепью*.
- Процесс, в котором переходы выполняются по выбору некоего "одушевленного существа", стремящегося выбрать такой набор или последовательность решений, которые обеспечат ему максимальное значение некоторой функции от траектории процесса.

Разумеется, кроме таких чистых схем имеются и всякого рода смеси. Однако чистые схемы интересны тем, что по ним можно увидеть, на что в соответствующей области науки обращается особое внимание и насколько традиции этой конкретной области влияют на изучение одного и того же математического объекта.

Сейчас термин "процесс" широко используется в программировании. Мы постараемся описать некоторые важные ситуации такого использования и внутри программы, и во взаимодействии отдельных частей программной системы, и на уровне управления ходом вычислений в компьютере.

10.1. Конечные автоматы

Что такое автомат, будем определять постепенно. Пусть заданы:

M – конечное непустое множество, элементы которого называются *состояниями автомата*;

A – конечное непустое множество *внешних воздействий* на автомат;

B – множество *ответов автомата* на внешние воздействия.

Обычно множество A называется *входным алфавитом* автомата, а множество B — *выходным алфавитом*.

Автомат — это процесс, который рассматривается в дискретные моменты времени (такты работы) и в каждый момент времени получает внешние воздействия. В зависимости от воздействия и своего текущего состояния процесс переходит в новое состояние и вырабатывает свой ответ *). Нас интересует поведение автомата во времени: его переходы из одного состояния в другое и реакции на воздействия.

Первоначально, в нулевом такте, автомат находится в некотором заданном начальном состоянии $i_0 \in M$. Каждый следующий такт t начинается с того, что в автомат поступает некоторый элемент a_t множества A, и в зависимости от этого элемента и текущего состояния процесса i_{t-1} процесс переходит в новое состояние i_t. Попутно вырабатывается действие автомата b_t. Таким образом, в канонических терминах автомат переводит входную строку алфавита A в выходную строку алфавита B.

Правила перехода автомата в новое состояние и выработки им действий определяются в простейшем случае таблицами. Есть более удобные и компактные способы задания правил, но сейчас нам важна не форма зависимости, а набор параметров, определяющий выбор. Итак, пусть правила перехода в новое состояние и выбора действия определяются отображениями $T : M \times A \to M$ и $D : M \times A \to B$.

*) Так определенный автомат называется *автоматом Мили* (George H. Mealey, 1955). Часто используется и другой вариант, называемый *автоматом Мура* (Edward Forrest Moore, 1956). Автомат Мура можно считать некоторым упрощением автомата Мили, поэтому мы его рассматривать не будем.

Рассмотрим несколько простых примеров.

Пример 1. *Сканирующий автомат.* Задается последовательность символов, в которой требуется выделить запись целого числа (в десятичной системе счисления). Эта запись состоит из последовательности цифр, которая окаймлена нецифровыми символами (кроме знаков - и +, которые могут войти в запись числа, и знака . (точка), который использовать запрещено). Для упрощения я выбрал представление числа, в котором запрещено многократное появление знаков - и + и появление точки.

По завершении выделения числа автомат выдает результат и переходит в состояние готовности к следующему сканированию. В случае появления запрещенной последовательности цифр автомат выдает сообщение об ошибке и также переходит в состояние готовности.

Итак, можно выделить следующие состояния автомата:

*S*0 – состояние начальной готовности — нет еще никакой информации о сканируемом числе, если какие-либо символы и поступили, то они не были использованы;

*S*1 – состояние чтения числа — прочтен символ - или +, который определил знак и запретил появление любого символа, кроме цифры;

*S*2 – состояние чтения числа — прочтена хотя бы одна цифра.

Различие состояний *S*1 и *S*2 — в их реакции на появление нецифрового символа; когда известен только знак числа, рано вырабатывать результат.

Хотя на входе могут появляться любые символы и с точки зрения вычисления числа существенно, например, какая именно цифра появилась, для состояния автомата и вырабатываемых действий важны только категории символов — *Цифра* (Ц), *Знак* (З), *Прочий символ* (П).

Теперь можно составить таблицу переходов (левая таблица):

	Ц	З	П
*S*0	*S*2	*S*1	*S*0
*S*1	*S*2	*S*0	*S*0
*S*2	*S*2	*S*0	*S*0

	Ц	З	П
*S*0	*D*2	*D*1	*D*0
*S*1	*D*3	*D*4	*D*5
*S*2	*D*3	*D*4	*D*6

Правая таблица указывает действия, которые должны быть выполнены в той или иной ситуации. Перечислим их в порядке появления:

*D*0 – начальное сканирование — ничего не делать;

*D*1 – прочтен знак числа — установить нулевое значение числа и знаковый множитель, зависящий от прочтенного символа;

*D*2 – прочтена первая цифра — установить знаковый множитель, равный +1, и начальное значение числа по прочтенной цифре;

*D*3 – прочтена очередная цифра — пересчитать значение числа по прочтенной цифре, т. е. умножить на 10 и прибавить цифру;

*D*4 – прочтен знаковый символ, который уже не нужен, — это ошибка;

*D*5 – прочтен посторонний символ, до того как начато чтение числа;

*D*6 – прочтен посторонний символ, завершивший чтение числа, — выдать в качестве результата произведение числа на знаковый множитель.

В качестве упражнения измените автомат таким образом, чтобы в записи числа допускались одиночные пробелы между цифрами и после знака числа.

Пример 2. *Печатающий автомат*[*]. Имеется возрастающая последовательность натуральных чисел, которая должна быть напечатана. В случае, когда больше двух чисел идет подряд, их нужно печатать как диапазон значений, т.е. печатать начальное и конечное значение группы чисел, идущих подряд, и знак тире между ними. Например, ряд чисел

```
7,11,12,13,14,23,27,28,32,33,34,35,36,43
```

должен выглядеть так:

```
7,11-14,23,27,28,32-36,43.
```

На вход автомата поступает последовательность символов из такого трехбуквенного алфавита:

```
Начало группы (Н), Продолжение группы (П), Конец данных (К).
```

В нашем примере последовательность чисел превращается в последовательность букв входного алфавита:

```
Н,Н,П,П,П,Н,Н,П,Н,П,П,П,П,Н,К.
```

Автомат имеет следующие состояния:

*S*0 – начальное состояние;

*S*1 – прочтено начало группы;

*S*2 – группа содержит два числа;

*S*3 – группа содержит больше чем два числа.

Таблицы переходов и действий:

	Н	П	К
*S*0	*S*1	*S*0	*S*0
*S*1	*S*1	*S*2	*S*0
*S*2	*S*1	*S*3	*S*0
*S*3	*S*1	*S*3	*S*0

	Н	П	К
*S*0	*D*1	*D*0	*D*0
*S*1	*D*2	*D*3	*D*4
*S*2	*D*5	*D*3	*D*6
*S*3	*D*7	*D*3	*D*8

[*] Такая задача возникает при подготовке авторских и предметных указателей в книге: автор отмечает ссылки, которые должны войти в указатель, эти элементарные ссылки собираются вместе с номерами страниц, сортируются по словам — "статьям указателя" и номерам страниц, а затем для каждого слова вырабатывается список диапазонов.

Переходы достаточно ясны. Опишем действия:

*D*0 – ошибка, в нулевом состоянии может появиться лишь начало группы;

*D*1 – запомнить поступившее начало группы;

*D*2 – напечатать имеющееся начало группы и запятую, затем выполнить действие D1;

*D*3 – запомнить поступившее число как конец группы;

*D*4 – напечатать имеющееся начало группы и точку;

*D*5 – напечатать имеющуюся группу из двух чисел и запятую, затем выполнить действие D1;

*D*6 – напечатать имеющуюся группу из двух чисел и точку;

*D*7 – напечатать имеющуюся группу и запятую, затем выполнить действие D1;

*D*8 – напечатать имеющуюся группу и точку.

Легко видеть, что эти довольно многочисленные действия на самом деле компонуются из простых стандартных действий. Можно сопоставить каждому элементарному действию двоичный разряд — *флаг* — и набирать необходимые действия в виде битовой последовательности требуемых флагов.

Пример 3. *Декодирующий автомат.* Хороший пример автомата дает код Хаффмена (см. с. 122): зададимся целью построить алгоритм, декодирующий текст, который записан этим кодом. Рассмотрим пример, использованный при описании алгоритма Хаффмена. Были построены кодовые последовательности

```
_ 111    а 010    и 000    е 1100   о 1000   н 0110   т 0010 к 0011
в 11010  л 10100  д 10101  т 10010  р 10011  з 110110 с 101100
б 101101 п 011110 й 011100 у 011101 х 1101110 ы 1101111 ы 1011110
ч 1011111 ж 1011100 ю 1011101 я 0111110 ь 0111111
```

и кодовое дерево, изображенное на рис. 10.1. Знак пробела на рисунке изображен звездочкой. Чтобы получить из этого дерева граф переходов автомата,

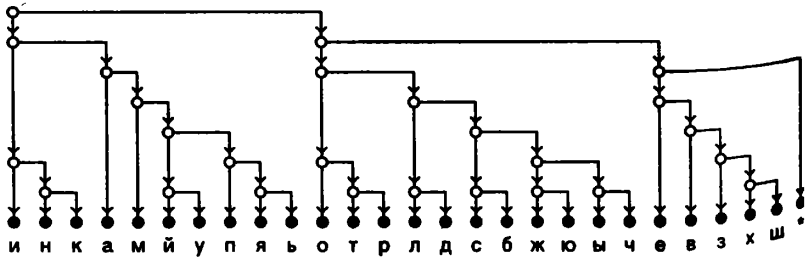


Рис. 10.1. Дерево кода Хаффмена

нужно каждую дугу, ведущую в тупиковую вершину (в черный кружок), заменить дугой, ведущей в корень дерева (не делайте этого, а только представьте себе). Вся рабочая информация для декодирующего автомата легко представляется следующей таблицей:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	1	2	и	н	а	м	7	й	п	я	11	12	о	т	15	л	17	с	19	ж	ы	22	е	в	з	х
1	10	4	3	к	5	6	8	у	9	ь	21	14	13	р	16	д	18	б	20	ю	ч	_	23	24	25	ш

где для простоты число обозначает новое состояние автомата, а другой знак — вырабатываемый символ и 0 в качестве нового состояния.

В случае поочередного кодирования элементов из двух алфавитов (как это может произойти, например, в графических файлах, когда изображение раскладывается на монохромные полоски, кодируемые парами "цвет–длина") автомат составляется из двух декодирующих деревьев и по достижении конечного состояния в одном из деревьев переходит в корень другого дерева.

Пример 4. *Автомат для поиска образца в строке.* Пусть задан образец — строка $p[1:m]$ и текст $t[1:n]$. Построим автомат, состояниями которого будут числа из диапазона $0:m-1$. Пребывание автомата в состоянии k означает, что при сканировании текста обнаружено совпадение префикса $p[1:k]$ образца с концом просмотренной части текста (т.е. с $t[s-k+1:s]$, где s — индекс последнего просмотренного символа). При продолжении сканирования текста символ $t[s+1]$ сравнивается с $p[k+1]$. Если они совпадают, то при $k+1=m$ процесс завершается — образец нашелся, а при меньшем k автомат переходит в состояние $k+1$. Если же символы не совпали, то автомат переходит в некоторое состояние k', определяемое наибольшим совпадением конца префикса $p[1:k+1]$ с меньшим префиксом. Эти значения $k'(k+1)$ зависят только от образца p, могут быть вычислены заранее и сведены в таблицу переходов.

Например, при поиске образца *aabbaabaab* в строке над алфавитом $\{a, b\}$ таблица переходов выглядит следующим образом:

Строка		*a*	*a*	*b*	*b*	*a*	*a*	*b*	*a*	*a*	*b*
Состояние	0	1	2	3	4	5	6	7	8	9	
Символ *a*	1	2	2	1	5	6	2	8	9	2	
Символ *b*	0	0	3	4	0	0	7	4	0	0*	

В начале сканирования текста автомат находится в состоянии 0. Символы последовательно проверяются, и автомат, как описано, от каждого символа переходит в новое состояние. Пока автомат не попадет в состояние 0*, никаких внешних действий не выполняется (такой автомат называется *распознающим*).

На рис. 10.2 тот же автомат представлен его *графом переходов*: состояниям автомата соответствуют вершины, а переходы из состояния в состояние представлены дугами.

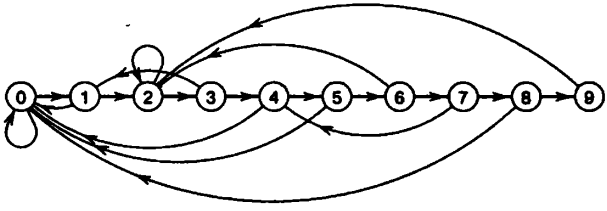


Рис. 10.2. Граф переходов распознающего автомата. Жирнее нарисованы дуги, повышающие совпадение строк

Упражнение 10.1. Проследите, как происходит поиск *aabbaabaab* при проверке строки $t = abaabbaabbaabaabaaa$.

Упражнение 10.2. Как нужно изменить определение автомата, чтобы он искал все вхождения строки в текст?

Упражнение 10.3. Как изменится автомат, если текст t может содержать символы, не входящие в образец?

Упражнение 10.4. Попробуйте построить автомат для поиска в тексте строки *abaabaaabaaaab*.

Упражнение 10.5. Предложите метод для построения автомата по образцу (отметим, что состояние k', в которое автомат переходит при несовпадении очередных символов текста и образца, определяется предварительной обработкой для упоминавшегося на с. 103 метода Кнута–Морриса–Пратта).

Пример 5. *Автомат для регулярного выражения.* Проверка соответствия строки регулярному выражению может также осуществляться автоматом. Назовем строку, удовлетворяющую данному регулярному выражению, регулярной. Состояниями нашего автомата будут всевозможные *регулярные префиксы* — строки, которые могут быть продолжены до регулярных строк. Поступающий в автомат очередной символ приписывается к текущему регулярному префиксу, после чего получившаяся строка урезается (слева) до максимального регулярного префикса.

Посмотрите на таблицу автомата для регулярного выражения `.*(saprophyle|proposal).*`. Попробуйте нарисовать граф переходов этого автомата. Автоматы широко используются при разработке трансляторов для алгоритмических языков (по этому поводу см., например, [6]), при описании которых часто применяются рекурсивные описания типа регулярных выражений.

Таблица автомата для регулярного выражения

Состояние	#	*s*	*a*	*p*	*r*	*o*	*h*	*y*	*l*	*e*	*
.*	0	1	0	11	0	0	0	0	0	0	0
.*s	1	1	2	11	0	0	0	0	0	0	0
.*sa	2	1	0	3	0	0	0	0	0	0	0
.*sap	3	1	0	11	4	0	0	0	0	0	0
.*sapr	4	1	0	11	0	5	0	0	0	0	0
.*sapro	5	1	0	6	0	0	0	0	0	0	0
.*saprop	6	1	0	11	0	0	7	0	0	0	0
.*saproph	7	1	0	11	0	0	0	8	0	0	0
.*saprophy	8	1	0	11	0	0	0	0	9	0	0
.*saprophyl	9	1	0	11	0	0	0	0	0	10	0
.*saprophyle	10	19	19	19	19	19	19	19	19	19	19
.*p	11	1	0	11	12	0	0	0	0	0	0
.*pr	12	1	0	11	0	13	0	0	0	0	0
.*pro	13	1	0	14	0	0	0	0	0	0	0
.*prop	14	1	0	11	0	15	0	0	0	0	0
.*propo	15	16	0	11	0	0	0	0	0	0	0
.*propos	16	1	17	11	0	0	0	0	0	0	0
.*proposa	17	1	0	11	0	0	0	0	18	0	0
.*proposal	18	19	19	19	19	19	19	19	19	19	19
Ok	19	19	19	19	19	19	19	19	19	19	19

10.2. Марковская цепь

Термин *цепь* был введен в теорию вероятностей А. А. Марковым-старшим, который рассмотрел схему "испытаний, связанных в цепь". Речь идет о последовательности случайных испытаний, в которой существует зависимость между отдельными испытаниями, в каком-то смысле самая простая после полной взаимной независимости: зависимость определяется между соседними элементами последовательности; так и получается "цепь испытаний".

Рассмотрим процесс с множеством состояний M и со следующим правилом перехода из состояния в состояние: когда процесс находится в состоянии $i \in M$, делается случайное испытание, которое определяет, в какое состояние на следующем такте процесс должен перейти. Зависимость результата испытания от состояния i находится в распределении вероятностей его исходов: каждому i сопоставляется набор вероятностей $p_i(j)$, с которыми процесс попадет в состояния $j \in M$.

При этом, разумеется,

$$\sum_j p_i(j) = 1, \qquad i \in M.$$

Если рассматривать процесс в последовательные моменты времени, начиная с момента 0, и задать начальное распределение вероятностей $p^{(0)}$, где $p^{(0)}(i)$ — вероятность находиться в начальный момент в состоянии i, то можно определить вероятность появления траектории процесса $i_0, i_1, \ldots, i_t$ как произведение

$$P(i_0, i_1, \ldots, i_t) = p^{(0)}(i_0) \times \prod_{k \in 1:t} p_{i_{k-1}}(i_k).$$

Таким образом мы определили распределение вероятностей на множестве траекторий заданной длины t; эти вероятности, очевидно, неотрицательны и их сумма равна 1. В последнем легко убедиться, например, по индукции, используя соотношение

$$P(i_0, i_1, \ldots, i_t) = P(i_0, i_1, \ldots, i_{t-1}) \times p_{i_{t-1}}(i_t).$$

Используя это выражение для вероятности траектории, можно сосчитать и всевозможные вероятности более сложных событий. Например, как подсчитать вероятность $p^{(t)}(i)$ того, что на шаге t процесс будет находиться в состоянии i? Подсчитаем ее сначала для $t = 1$. Имеем:

$$p^{(1)}(i) = \sum_j p^{(0)}(j) \times p_j(i).$$

Осознав, что набор вероятностей $p_j(i)$ образует просто матрицу, в которой j может считаться индексом строки, а i — индексом столбца (она называется *матрицей переходных вероятностей* марковской цепи), мы сможем сказать, что вектор $p^{(1)}$ получается умножением*) вектора $p^{(0)}$ на матрицу переходных вероятностей P. Для вычисления $p^{(2)}$ мы получим аналогичную формулу

$$p^{(2)} = p^{(0)} \times P \times P = p^{(0)} \times P^2$$

*) Если вы привыкли к тому, что бывают векторы-строки и векторы-столбцы, вы обнаружите, что в данном случае векторы p должны быть столбцами. Мне представляется, что без этого традиционного разделения можно обойтись, считая, что отдельно определено умножение вектора (не столбца и не строки, а просто вектора) на матрицу, слева и справа, и эти умножения отличаются друг от друга и от умножения матрицы на матрицу, хотя и обозначаются одинаково.

(причем в последнем случае 2 обозначает уже квадрат матрицы P). Теперь уже ясно, что мы можем ожидать для шага t формулу

$$p^{(t)} = p^{(0)} \times P^t.$$

Проверьте ее справедливость сами.

Оказывается, векторы $p^{(t)}$ при росте t в достаточно общих и естественных предположениях стабилизируются и сходятся к некоторому вероятностному вектору $\overline{p}$, который можно назвать *стационарным распределением* цепи. Стационарность проявляется в том, что, взяв $p^{(0)} = \overline{p}$, мы получим $p^{(t)} = \overline{p}$ для любого t. Вот простейшие условия, для которых имеет место сходимость к стационарному распределению.

Теорема. Если все элементы матрицы переходных вероятностей P положительны, то при t, стремящемся к бесконечности, вектор $p^{(t)}$ стремится к вектору $\overline{p}$, являющемуся единственным решением системы

$$p \times P = p, \qquad \sum_i p_i = 1.$$

Доказательство. Обозначим через d минимальный элемент матрицы P. По предположению $d > 0$. Введем для удобства функцию f, переводящую вероятностный вектор в другой: $f(p) = p \times P$. Покажем прежде всего, что отображение f сжимающее, т. е. что отношение "расстояния" между векторами $f(p)$ и $f(q)$ к расстоянию между векторами p и q не превосходит некоторого значения ρ, где $\rho < 1$. Расстояние в данном случае — это просто некоторая числовая мера отклонения одного вектора от другого. Нужно только хорошо выбрать эту меру, мы предлагаем попробовать

$$r(p, q) = \sum_i |p_i - q_i|,$$

и посмотрим сейчас, чем такая мера хороша. Введем матрицу D, элементы которой на d меньше, чем у P. Учитывая, что у вероятностных векторов сумма компонент равна 1, и используя определение d, получаем для любого вектора p

$$f(p) = d \times \mathbf{1} + p \times D$$

(напомним, что $\mathbf{1}$ — это вектор из единиц). Таким образом,

и далее

$$r(f(p), f(q)) = \sum_i \Big| \sum_j p_j D_{ji} - \sum_j q_j D_{ji} \Big| \le \sum_i \sum_j |p_j D_{ji} - q_j D_{ji}| =$$
$$= \sum_j \sum_i D_{ji} |p_j - q_j| = r(p, q) \times (1 - md).$$

Здесь мы можем принять $\rho = 1 - md < 1$ (подумайте, почему так определенное ρ не будет отрицательным). Из получившегося неравенства

$$r(pP, qP) \le \rho \times r(p, q)$$

следует для любого n

$$r(pP^n, qP^n) \le \rho^n \times r(p, q),$$

откуда легко получается сходимость последовательности $p^{(0)}P^n$ при любом начальном векторе $p^{(0)}$. Переходя в формуле $p^{(n)} = p^{(n-1)}P$ к пределу, мы получаем для предельного вектора $\overline{p}$ уравнение $\overline{p} = \overline{p}P$.

Единственность решения этого уравнения легко следует из того же неравенства. Действительно, если p_1 и p_2 — два решения, то

$$r(p_1, p_2) = r(p_1P, p_2P) \le \rho \times r(p_1, p_2),$$

что возможно только при совпадении этих решений. □

Следствие. Если при каком-нибудь $n > 0$ все элементы матрицы P^n положительны, то утверждение теоремы остается в силе.

Доказательство. Матрицу P в доказательстве сходимости можно заменить матрицей P^n и доказать сходимость в последовательностях $\{p_{kn+s}\}_k$ для любого $s \in 0 : n - 1$. Далее, воспользовавшись тем, что матрица P^{n+1} также положительна, мы увидим сходимость и в последовательностях $\{p_{k(n+1)+s}\}_k$, а отсюда уже и сходимость всей $\{p_k\}_k$. □

Сходимость векторов $p^{(t)}$ к $\overline{p}$ связана с замечательным статистическим свойством марковской цепи: если рассмотреть траекторию марковской цепи длины n: $i_1, i_2, \ldots, i_n$ и подсчитать на этой траектории частоты попадания цепи в различные состояния

$$k_n(i), \quad i \in M, \qquad \sum_{i \in M} k_n(i) = n,$$

то нормированные частоты сходятся в некотором "вероятностном" смысле к вектору $\overline{p}$. Это свойство называется *эргодичностью* марковской цепи *). Точный смысл сходимости трудно определить потому, что эта сходимость верна для *почти всех траекторий*, но существование траекторий, которые этим свойством не обладают, возможно.

Много информации о поведении марковской цепи дает *граф переходов*, который строится по матрице переходных вероятностей. Вершины графа — состояния цепи, а дуги соответствуют положительным элементам матрицы: если $P[i, j] > 0$, то паре (i, j) соответствует дуга, ведущая из i в j. Можно сказать, что марковская цепь — это *случайное блуждание* по ее графу переходов.

Построим диаграмму порядка графа переходов. Как мы помним, она представляет собой граф без контуров, вершины которого суть элементы некоторого разбиения множества M. Так как в диаграмме порядка нет контуров, то в ней существуют тупиковые вершины, т. е. вершины, из которых дуги не выходят.

Подмножества M, соответствующие тупиковым вершинам диаграммы порядка, называются *эргодическими классами* марковской цепи. Если ограничиться в качестве множества состояний цепи одним эргодическим классом, то цепь будет эргодической.

Пример. Пусть $S = \{A, B, C, D, E, F, G, H, I, J, K, L\}$ и матрица переходных вероятностей $P[S, S]$ задана приведенной таблицей:

	A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*
A	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
B	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
C	0.2	0.2	0.0	0.0	0.1	0.0	0.0	0.0	0.5	0.0	0.0	0.0
D	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
E	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
F	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
G	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
H	0.5	0.0	0.0	0.0	0.0	0.2	0.0	0.3	0.0	0.0	0.0	0.0
I	0.0	0.0	0.5	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
J	0.0	0.0	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.6	0.0
K	0.0	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0	0.8	0.0	0.0
L	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.5

В этих нулях трудно разбираться, а нам сейчас нужны только места положительных элементов, поэтому построим таблицу переходов, а по ней граф переходов и диаграмму порядка (рис. 10.3).

*) От греческих слов *εργον* — работа и *οδος* — путь.

	A	B	C	D	E	F	G	H	I	J	K	L
A										+		
B		+										
C	+	+			+				+			
D									+			
E												+
F	+											
G										+		
H	+					+		+				
I			+		+							
J							+				+	
K						+				+		
L					+							+

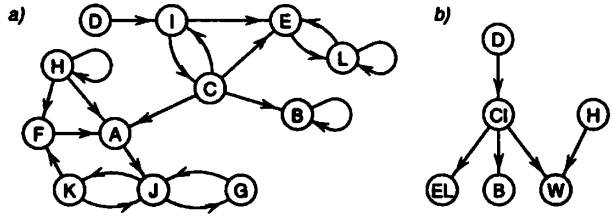


Рис. 10.3. Граф переходов (*a*) и его диаграмма порядка (*b*)

Из диаграммы порядка ясно видны три эргодических класса: $S_1 = \{E, L\}$, $S_2 = \{B\}$, $S_3 = \{A, G, K, F, J\}$ (последний обозначен буквой W). Состояния, вроде B, образующие одноэлементные эргодические классы, называются *поглощающими* состояниями.

Множество S_3 разбито на такие два подмножества: $S_{31} = \{A, G, K\}$ и $S_{32} = \{F, J\}$, что переходы из состояния в состояние всегда вызывают переход из одного подмножества в другое. Говорят, что это эргодический класс с периодом 2. В общем случае при периоде (числе подмножеств) d получаем циклическое движение по подмножествам $S_1 \to S_2 \rightsquigarrow \cdots \to S_d \to S_1$.

Разбиение S_3 на два подмножества будет видно лучше, если выписать подматрицу $p[S_3, S_3]$, переставив в ней строки и столбцы:

Отметим, что в матрице $P^2[S_3, S_3]$ все ненулевые элементы будут сосредоточены в диагональных блоках $P^2[S_{31}, S_{31}]$ и $P^2[S_{32}, S_{32}]$, а у матрицы $P^2[S_3, S_3]$ все элементы диагональных блоков положительны.

С вероятностью 1 траектория марковской цепи оказывается в одном из эргодических классов и выйти из него уже не может. По этой причине состояния эргодических классов принято называть *существенными*, а все прочие состояния — несущественными *).

Период эргодического класса S_i определяется как общий наибольший делитель длин контуров в графе переходов этого класса. Можно показать, что если период равен 1, то для любого достаточно большого t все элементы матрицы $P^t[S_i, S_i]$ положительны. Этого достаточно для применения следствия.

При периоде d, большем 1, когда эргодический класс S_i разбивается на d "фазовых" подклассов S_{is}, $s \in 0 : (d-1)$, нужно рассматривать матрицу $\overline{P}[S_i, S_i] = P^d[S_i, S_i]$. Ее подматрица $\overline{P}[S_{is}, S_{is}]$ при любом фиксированном s будет стохастической матрицей цепи с единственным эргодическим классом с периодом 1. Следовательно, вероятностное распределение будет сходиться в каждом фазовом подклассе отдельно.

Рассмотрим теперь несколько примеров, где полезно использовать технику марковских цепей.

Пример 1. *Схема обслуживания с отказами.* Рассмотрим систему, обслуживающую поступающие запросы, например сервер. Система состоит из 4 одинаковых блоков, например модемов, к которым поступают звонки от пользователей — *поток запросов на обслуживание*. Если все четыре блока заняты, запрос теряется. Если же есть свободные блоки, то один из них принимает запрос и становится занятым до завершения выполнения запроса.

Моделируя эту систему как марковскую цепь, мы принимаем в качестве ее состояния количество занятых блоков. Будем считать время дискретным и шаг по времени выберем достаточно маленьким так, чтобы можно было пренебречь вероятностью поступления двух запросов за один шаг. Теперь обозначим через α вероятность поступления одного запроса, тогда вероятность непоступления запросов будет, очевидно, равна $1 - \alpha$.

Может показаться странным, но время обслуживания также удобно считать случайным и состоящим из независимых продолжений **). Если говорить более точно, то мы считаем, что запрос с вероятностью β обслуживается за

*) Мне такая терминология представляется неоправданно резкой. Есть много задач, в которых существенны именно эти "несущественные" состояния, так как единственное поглощающее состояние очевидно, и интересно не стационарное пребывание в нем, а сам переход в это состояние. Да и мы, пока живем, находимся в несущественных (по этой терминологии) состояниях.

**) Это предположение очень удобно для математических выкладок и хорошо согласуется с практическими наблюдениями над реальными системами обслуживания.

один шаг, а с вероятностью $1-\beta$ обслуживается после этого шага как новый запрос. Это дает вероятность $(1-\beta)\beta$ для обслуживания за два шага, $(1-\beta)^2\beta$ — для обслуживания за три шага и т. д.

Теперь нетрудно написать матрицу переходных вероятностей

$$P[0:4,0:4]=\begin{pmatrix} 1-\alpha & \alpha & 0 & 0 & 0 \\ \beta & 1-\alpha-\beta & \alpha & 0 & 0 \\ 0 & 2\beta & 1-\alpha-2\beta & \alpha & 0 \\ 0 & 0 & 3\beta & 1-\alpha-3\beta & \alpha \\ 0 & 0 & 0 & 4\beta & 1-4\beta \end{pmatrix}$$

и увидеть, что она имеет единственный эргодический класс, и следовательно, система $\overline{p}=\overline{p}\times P$ в классе вероятностных векторов имеет единственное решение. Выпишем уравнения этой системы явно

$$\begin{aligned} \overline{p}_0\cdot(1-\alpha)+\overline{p}_1\cdot\beta &= \overline{p}_0, \\ \overline{p}_0\cdot\alpha+\overline{p}_1\cdot(1-\alpha-\beta)+\overline{p}_2\cdot 2\beta &= \overline{p}_1, \\ \overline{p}_1\cdot\alpha+\overline{p}_2(1-\alpha-2\beta)+\overline{p}_3\cdot 3\beta &= \overline{p}_2, \\ \overline{p}_2\cdot\alpha+\overline{p}_3\cdot(1-\alpha-3\beta)+\overline{p}_4\cdot 4\beta &= \overline{p}_3, \\ \overline{p}_3\cdot\alpha+\overline{p}_4\cdot(1-4\beta) &= \overline{p}_4, \end{aligned}$$

или после несложных преобразований

$$\begin{aligned} -\overline{p}_0\cdot\alpha+\overline{p}_1\cdot\beta &= 0, \\ \overline{p}_0\cdot\alpha-\overline{p}_1\cdot(\alpha+\beta)+\overline{p}_2\cdot 2\beta &= 0, \\ \overline{p}_1\cdot\alpha-\overline{p}_2(\alpha+2\beta)+\overline{p}_3\cdot 3\beta &= 0, \\ \overline{p}_2\cdot\alpha-\overline{p}_3\cdot(\alpha+3\beta)+\overline{p}_4\cdot 4\beta &= 0, \\ \overline{p}_3\cdot\alpha-\overline{p}_4\cdot 4\beta &= 0, \end{aligned}$$

откуда, полагая $\alpha/\beta=\gamma$, получаем

$$\overline{p}_1-\gamma\overline{p}_0=2\overline{p}_2-\gamma\overline{p}_1=3\overline{p}_3-\gamma\overline{p}_2=4\overline{p}_4-\gamma\overline{p}_3=0,$$

и следовательно,

$$\overline{p}_1=\overline{p}_0\gamma,\quad \overline{p}_2=\gamma^2\overline{p}_0/2,\quad \overline{p}_3=\gamma^3\overline{p}_0/3,\quad \overline{p}_4=\gamma^4\overline{p}_0/4,$$

и из условия нормировки

$$\overline{p}_0=C=(1+\gamma+\gamma^2/2+\gamma^3/3+\gamma^4/4)^{-1}.$$

Итак, мы нашли набор π_i — вероятностей того, что в стационарном режиме в системе будет занято i блоков. В частности, долю времени $\overline{p}_4=C\cdot\gamma^4/4$ в системе заняты все блоки, и поступающие в это время запросы пропадают (таким образом, $\overline{p}_4$ определяет не только долю времени полной занятости, но и долю пропадающих запросов).

Полученные здесь формулы легко переносятся на случай произвольного числа блоков, и мы можем провести экономический анализ системы: сопоставить затраты на дополнительные блоки и даваемый ими прирост работоспособности системы.

Пример 2. *Моделирование сочетаний слов в тексте.* В книге [22] описан поучительный эксперимент. Рассмотрим текст, состоящий из слов w_i, $i \in 1:n$. Представим себе процесс, в котором состояниями являются пары слов, так что когда процесс находится в состоянии (w', w'') и поступает (откуда? это мы определим потом) слово w''', то процесс переходит в состояние (w'', w''').

Дальше берется какой-нибудь достаточно длинный текст и в предположении, что описанный процесс является марковской цепью, этот текст используется для оценки переходных вероятностей. В соответствии с переходными вероятностями можно строить траектории марковской цепи. Получающийся текст похож на связный текст много больше, чем совершенно случайный текст в экспериментах лапутской Академии наук в "Путешествиях Гулливера".

Пример 3. *Марковская модель работы программы.* Это пример того, как в трудной ситуации приходится идти на заведомо неправильную, неадекватную реальности, модель для того, чтобы получить хоть какой-то результат. Предположим, что мы анализируем какую-то программу и хотим оценить временные параметры ее работы — среднее время исполнения программы, частоту выполнения отдельных блоков и т.д. Составим граф переходов этой программы (ага! у нас есть удобный термин), в котором вершинами будут точки ветвления алгоритма (условные переходы), а дугами — пути вычислений между этими вершинами. Теперь сделаем переходы случайными, для чего припишем дугам, выходящим из каждой вершины, какие-либо вероятности (это вероятности выбора дуг при условии, что процесс находится в данной вершине, так что сумма вероятностей дуг, выходящих из каждой вершины, равна единице).

Например, в алгоритме Краскала мы можем выделить вершины:

start — s, начало работы, возможен переход к состоянию a;

arc — a, получение новой дуги, возможен переход к e или к c;

check — c, анализ дуги, возможен переход к i или к a;

include — i, вставка дуги, возможен переход к a или к e;

end — e, завершение работы.

В результате получится граф, изображенный на рис. 10.4. Осталось только задать (с очевидными обозначениями) вероятности переходов p_{ac}, p_{ci} и p_{ie}, и тогда можно будет изучать, хотя бы грубо, продолжительности переходов между вершинами, вероятности попадания в различные состояния и другие средние характеристики процесса. Такие описания имеют особое значение для

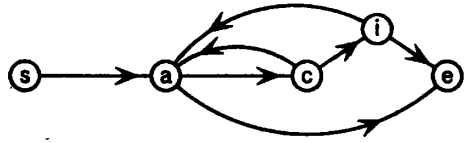


Рис. 10.4. Граф переходов для алгоритма Краскала

программ, обслуживающих непрерывно работающее производство, — в них важно оценить, достаточно ли вычислительных мощностей для своевременного принятия управленческих решений.

10.3. Управляемые процессы

Процесс *принятия решений* — это третий характерный тип процесса. В нем состояние меняется в результате *управляющих воздействий* особого участника модели, который так и называется: *лицо, принимающее решения*, или ЛПР *). Классический пример процесса принятия решений дает схема динамического программирования, разработанная американским математиком Ричардом Беллманом **). Мы уже встречали схему динамического программирования, когда рассматривали, например, задачу о кратчайшем пути, задачу о критическом пути в сетевом графике, задачу о разбиении абзаца на строки, задачу о поиске наибольшей общей подпоследовательности.

Главная "изюминка" динамического программирования — включение решаемой экстремальной задачи в класс однотипных задач, которые следует решать вместе. Увидеть этот класс, рассматривая отдельную задачу, иногда очень легко, иногда трудно. Чтобы видеть стало легче, нужно накопить достаточный опыт. В конце параграфа мы рассмотрим несколько примеров, но вначале нас будет интересовать модель динамического программирования как *процесс принятия решений*.

*) Английский термин decision maker более удобен, но, к сожалению, существующие русские слова *решатель*, *управляющий* и т. п. уже заняты и имеют другой смысл. Подходящее по смыслу и используемое сейчас слово *администратор* буквально воспроизводит английский термин.

**) Здесь нужно хотя бы только упомянуть об еще одном американском математике Абрахаме Вальде (Abraham Wald, 1902–1950), который примерно в то же время, что и Беллман, разрабатывал теорию последовательного принятия решений в статистике. Вальд ввел понятие *решающей функции* как правила, описывающего действия *Статистика* (так он называл лицо, принимающее решения и играющее против *Природы*) во всех гипотетически возможных ситуациях.

Этот процесс находится на каждом шаге в одном из состояний s множества состояний S, и "управляющий процесса" принимает *решение* q, учитывая при этом номер шага и состояние процесса. В зависимости от q (а зависимость от s спрятана в самом решении, будем считать, что $q \in Q_s$ и множества Q_s для различных s не пересекаются) процесс переходит в новое состояние $\sigma(q)$. Процесс продолжается до тех пор, пока не попадет в специальное *терминальное* (завершающее) состояние $s_\dagger$ или не кончится ресурс времени. Так получается *траектория процесса*, которая состоит из последовательности пар $\tau = \{(s_i, q_i)\}_{i \in 0:t-1}$, заканчивающейся терминальным состоянием. Для единообразия добавим множество $Q_{s_\dagger}$ из одного терминального решения, оставляющего процесс в том же терминальном состоянии, и положим $Q = \bigcup_s Q_s$.

Предположим, что на множестве таких последовательностей задана функция $f(\tau)$, АМ-тивная в смысле определения с. 96, т. е. существуют такие функции $\varphi, \psi : Q \to R$, что

$$f(\tau) = \varphi(\mathsf{head}\ \tau) + \psi(\mathsf{head}\ \tau) \times f(\mathsf{tail}\ \tau),$$

если строка непустая и задано значение $f(\{s_\dagger\})$.

Функция f вводится как *целевая функция* процесса; "управляющий" стремится выбирать решения так, чтобы построить траекторию с экстремальным (для определенности минимальным) значением f.

Пример 1. *Задача о кратчайшем пути.* Эта задача нам уже встречалась (см. с. 228), в ней требуется построить в графе $\langle M, N \rangle$ кратчайший путь от начальной вершины i_0 до заданной вершины $i_\dagger$. Здесь множеством состояний является M, а в каждом состоянии $i \in M$, кроме терминального состояния $i_\dagger$, множество возможных решений — это множество дуг, начинающихся в i:

$$Q_i = \{u \in N, \mathsf{beg}\ u = i\}.$$

Каждому решению $u \in Q_i$ сопоставляется новое состояние $\sigma(u) = \mathsf{end}\ u$, конец дуги u. Целевая функция в этой задаче аддитивна, мы минимизируем сумму длин дуг, входящих в путь, так что $\varphi(u) = l[u]$.

Пример 2. Рассмотрим экстремальную задачу, в которой сочетаются элементы обсуждавшейся уже задачи о рюкзаке и переходов из состояния в состояние, таких как в автоматах и марковских цепях. Имеется множество состояний M, процесс первоначально находится в одном из этих состояний i_0 и переходит из состояния в состояние по выбору управляющего. При переходе из состояния i в состояние j управляющий получает приз стоимостью c_{ij} и тратит целочисленный ресурс w_{ij}. Требуется найти траекторию (последовательность переходов), на которой суммарный затраченный ресурс не превосходил бы

заданной границы b, а суммарная стоимость полученных призов была максимальна.

Какой класс задач здесь можно ввести, чтобы наша задача принадлежала ему и решение отдельных задач помогало друг другу? Естественно рассмотреть задачи с различными значениями границы b и с различными начальными состояниями. Тогда множеством состояний процесса будет множество $0 : b \times M$. Для любого $\beta \in 0 : b$ и любого $i \in M$ можно образовать состояние (β, i) и сказать, что нас интересует задача (b, i_0). Любое состояние $(0, i)$ можно считать терминальным состоянием.

В каждом нетерминальном состоянии (β, i) в качестве решений рассматривается множество $J_{\beta,i}$ таких $j \in M$, для которых $w_{ij} \leq \beta$. При выборе решения j процесс переходит в состояние $(\beta - w_{ij}, j)$. Если множество $J_{\beta,i}$ пусто, то состоянию (β, i) сопоставляется решение, переводящее процесс в терминальное состояние.

На траекториях этого процесса определена целевая функция — суммарная стоимость полученных призов. Эта функция, очевидно, аддитивна и, следовательно, АМ-тивна. Чтобы не делать все время оговорок, будем считать, что минимизируется взятая с минусом суммарная стоимость.

Основное средство решения задач динамического программирования — это совместное решение всего класса задач и составление уравнения (обычно рекуррентного соотношения), позволяющего связать зависимостью минимальные значения целевой функции в разных состояниях процесса. В задачах динамического программирования выполняется следующий предложенный Р. Беллманом *принцип оптимальности*:.

В каждом состоянии процесса хвост оптимальной траектории, *т. е. траектории, минимизирующей значение целевой функции, является оптимальной траекторией для состояния, получающегося из данного состояния после первого шага.*

Будем теперь варьировать начальное состояние s_0. Обозначим через $v(s)$ минимальное значение целевой функции для процесса с начальным состоянием $s_0 = s$, т. е.

$$v(s) = \min\{f(\tau) \mid q_0, q_1, \ldots\}.$$

У нас получилась некоторая функция от s, которую принято называть *функцией Беллмана*. Минимум в приведенной формуле берется по всем допустимым последовательностям решений $q_0, q_1, \ldots$ Его можно представить как последовательный минимум: "внешний" минимум по q_0, а "внутренний", при фиксированном q_0, по оставшейся

последовательности *) $q_1, q_2, \ldots$

$$v(s) = \min_{q_0} \{\min\{f(\tau) \mid q_0, q_1, q_2, \ldots\}\}.$$

Цель такого расщепления понятна, мы сейчас воспользуемся свойством АМ-тивности целевой функции

$$v(s) = \min_{q_0} \{\min\{\varphi(\text{head } \tau) + \psi(\text{head } \tau) \times f(\text{tail } \tau) \mid q_1, q_2, \ldots\}\}.$$

Слагаемое $\varphi(\text{head } \tau)$ и неотрицательный множитель $\psi(\text{head } \tau)$ не зависят от параметров, по которым берется внутренний минимум, поэтому их можно вынести за минимум (поскольку, очевидно,

$$\min\{a + b \cdot g(x) \mid x\} = a + b \cdot \min\{g(x) \mid x\},$$

если $b \geq 0$), получая

$$v(s) = \min_{q_0} \{\varphi(\text{head } \tau) + \psi(\text{head } \tau) \times \min\{f(\tau') \mid s_0(\tau') = \sigma(q_0)\}\}.$$

Теперь видно, что выражение для внутреннего минимума является в соответствии с определением функции v значением для $v(\sigma(q_0))$, так что мы получаем

$$v(s) = \min_{q_0} \{\varphi(s, q_0) + \psi(s, q_0) \cdot v(\sigma(q_0))\}.$$

Это и есть уравнение Беллмана для общей задачи динамического программирования.

Уравнение Беллмана дает прямой путь вычисления функции Беллмана: достаточно вычислять функцию Беллмана для всех состояний в порядке, обеспечивающем их правильное вычисление. В случае если "правильного порядка" вычисления значений не существует, т. е. если в графе переходов процесса существуют контуры, может быть использована техника итеративного решения.

Пример 2 (продолжение). Продолжим рассмотрение примера 2. Посмотрим, каково будет уравнение Беллмана для сформулированной задачи. Зададим функцию Беллмана таблицей $v[0 : b, M]$ и положим $v[0, i] = 0$. Далее при $\beta > 0$ имеем

$$v[\beta, k] = \begin{cases} 0, & J_{\beta,i} = \varnothing, \\ \min\{c_{ij} + v[\beta - w_{ij}, j]\}, & j \in J_{\beta,i}. \end{cases}$$

Использование этого соотношения с β, меняющимся от 1 до b, последовательно заполняет всю таблицу.

*) По-моему, это похоже на условные вероятности.

Можно ввести еще одну таблицу, скажем $d[0:b, 0:n]$, и полагать $d[\beta, k]$ равным какому-то невозможному значению i_{nil}, если множество $J_{\beta,i}$ пусто, и равным j, на котором достигается минимум, во втором случае. Такая *решающая функция* позволит построить потом оптимальную траекторию, используя следующий *обратный ход*.

Алгоритм обратного хода

Состояние вычислительного процесса. Накопленная траектория T, оставшийся ресурс веса β, начальное состояние i.

Начальное состояние. $T = \{i_0\}$, $\beta = b$, $k = i_0$.

Стандартный шаг.

1. Если $i_0 = i_{\text{nil}}$, закончить.
2. Положить $j = d[\beta, i]$. Добавить состояние j к траектории T, уменьшить β на w_{ij}, положить i равным j.
3. Вернуться к пункту 1.

При прямом вычислении иногда нет необходимости в полном хранении функции Беллмана для всех состояний: ее можно вычислять "послойно", заменяя одну итерацию другой в одном и том же массиве, а в некоторых случаях, когда эта функция оказывается кусочно-постоянной, ее самое и даже оптимальные решения как функцию от состояния процесса достаточно хранить лишь частично. Мы использовали эти приемы при решении задачи о максимальном совпадении двух строк (с. 108), другое важное их применение — при решении с помощью динамического программирования задач оптимального раскроя, когда требуется в отрезок или прямоугольник уложить детали требуемых размеров с максимальной общей ценой (эта задача является естественным обобщением рассматривавшейся в главе 4 задачи о рюкзаке).

Непосредственное вычисление решения уравнения Беллмана во многих случаях неэффективно или даже невозможно из-за того, что множество состояний слишком велико. Кроме уже названного приема "послойного вычисления" для экономии места и времени счета можно использовать "каскадный метод", в котором решение находится только в тех состояниях, которые потребовались в процессе вычислений [*)].

Правильно рассматривать принципы динамического программирования как канву, т.е. только как основную идею для действий в конкретных случаях, имея в виду, что само уравнение Беллмана

[*)] Такой принцип организации принято называть lazy evaluation, буквально, *ленивое исполнение*.

численные методы для его решения могут в них сильно отличаться от базовой схемы. Рассмотрим пример такой конкретной задачи, уже встречавшийся нам раньше (см. с. 236).

Пример. *Задача Штейнера для направленного дерева.* Чтобы яснее увидеть, что принцип динамического программирования дает все-таки несколько больше, чем задача о кратчайшем пути, рассмотрим его применение к одной из постановок задачи Штейнера на графах. Принцип оптимальности будет несколько отличаться от предыдущего по форме, но идея будет все той же.

Итак, пусть задан граф $\langle M, N\rangle$, начальная вершина $i_0 \in M$ и множество терминальных (т. е. конечных) вершин $M_\dagger \subset M$, а для каждой дуги $j \in N$ задана ее положительная длина l_j. Требуется найти частичное дерево $\langle M', N'\rangle$ с кратчайшей суммой длин дуг, содержащее пути от i_0 до любой вершины из $M_\dagger$.

Мы намереваемся решать такие задачи для любой начальной вершины $i \in M$ и любого множества терминальных вершин $M_T \subset M_\dagger$, так что предлагаемый метод будет приемлем только в случае, когда можно хранить в памяти таблицу размера $|M| \times 2^{|M_\dagger|}$.

Пары (i, M_T) будут состояниями процесса, а в таблице будут храниться вычисляемые для всех состояний значения функции Беллмана.

Обозначим через $v(i, M_T)$ минимальную сумму длин дуг графа, содержащего пути от i до любой вершины из M_T. В том случае, когда множество M_T состоит из одной вершины, скажем i_1, этот граф легко найти — это кратчайший путь из i в i_1. Возьмем теперь произвольное состояние (i, M_T). Находясь в нем и предполагая, что $i \notin M_T$, мы можем выбрать следующие решения:

1. Перейти по какой-либо дуге, начинающейся в i, в другую вершину.
2. Разветвить путь: выбрать разбиение терминального множества M_T: $M_T = \bigcup_k M_{Tk}$ и для каждого состояния (i, M_{Tk}) решить такую же задачу.

Таким образом, мы получаем следующее уравнение Беллмана:

$$v(i, M_T) = \min\left\{v_{\text{cont}}(i, M_T \setminus \{i\}), v_{\text{part}}(i, M_T \setminus \{i\})\right\},$$

где

$$v_{\text{cont}}(i, M_T) = \min\left\{l_j + v(\text{end } j, M_T) \,|\, \text{beg } j = i\right\},$$

$$v_{\text{part}}(i, M_T) = \min\left\{v(i, A) + v(i, M_T \setminus A) \,|\, A \subset M_T\right\}.$$

В последнем соотношении достаточно перебирать разбиения множества M_T на два непустых подмножества, имея в виду, что каждое из них можно разбивать дальше.

Функцию v удобно находить, вычисляя все минимумы одновременно, т. е. составляя все необходимые суммы и сравнивая их с текущими значениями функции в разных состояниях. Но, по-видимому, проще прямо написать алгоритм. Обратите внимание на то, какими средствами уменьшается перебор возможных разбиений множества M_T.

Построение частичного направленного дерева

Состояние вычислительного процесса. Будем считать, что исходное множество $M_\dagger$ достаточно мало, чтобы можно было его подмножества (их характеристические векторы) представлять целыми числами. Пусть R — множество всех подмножеств M_T и одновременно множество всех целых чисел от 0 до $2^{|M_\dagger|} - 1$. Состояние вычислительного процесса мы представляем массивом $v[M, R]$ и числом r, задающим очередное подмножество.

Начальное состояние. Первоначально $r = 0$, что соответствует пустому множеству, все $v[i, \{i\}] = 0$ для $i \in M_\dagger$, остальные $v[i, M']$ принимают бесконечные значения.

Стандартный шаг.

1. Контроль завершения. Если увеличение r невозможно, завершить расчет. В противном случае увеличить r на единицу, перейдя к новому подмножеству.
2. Подготовка к поиску кратчайших путей. Образовать множество M_1 (в смысле алгоритма Дейкстры), поместив в него все вершины i, для которых $v[i, r] < \infty$.
3. Поиск путей по алгоритму Дейкстры.
4. Склеивание подмножества r с другими подмножествами. Для всех $r' < r$, таких что множества r и r' не пересекаются, положить $r'' = r + r'$ и для всех $j \in R$ "постараться уменьшить" $v[j, r'']$ до уровня $v[j, r] + v[j, r']$.

Разумеется, должны еще фиксироваться в другом массиве и те решения, которые привели к найденным минимальным значениям, как это демонстрировалось выше в связи с задачей о рюкзаке. Мы опустили эту часть алгоритма для простоты.

В данном примере важно было само определение состояния процесса, правил принятия решений, а также отличающийся от классического, беллмановского, ветвящийся характер процесса.

Имеются варианты управляемых процессов, в которых новое состояние процесса является случайным и только распределение вероятностей, с которыми появляется то или иное состояние, зависит от принимаемых решений. Такие модели называются *марковскими процессами решения*. Их практические приложения можно найти в так называемом управлении запасами, где нужно принимать решение о покупке товара для обеспечения недетерминированного спроса, и в задачах оптимального управления системами, состоящими из ненадежных элементов.

Упражнение 10.6. На плоскости задан n-угольник без самопересечений. Используя динамическое программирование, проведите в нем $n - 3$ непересекающиеся диагонали так, чтобы сумма их длин была минимальна.

Процессы динамического программирования хорошо описывают модели *управляемого движения* и управления физическими объектами, в том числе с непрерывным временем и непрерывным множеством состояний и управлений, причем сохраняется сама идея локальной оптимальности решений, принимаемых на каждом шаге пути.

Еще одно важное развитие идеи динамического программирования представляется *динамическими играми* — управляемыми процессами, в которых на выбор траектории влияют действия двух (или даже большего числа) лиц, принимающих решения исходя из различных, не совпадающих, а в случае двух лиц зачастую противоположных интересов; естественно, что эти участники игры называются *игроками*. Здесь возникают интересные модели накопления каждым игроком информации о состоянии игры, о действиях противника и даже о собственных действиях игрока.

10.4. Вычислительные процессы

Каждый раз, рассматривая какой-либо алгоритм, мы описывали вычислительный процесс и определяли, каково его состояние, как задается начальное значение этого состояния и каковы правила перехода. Мы считали, что такой подход достаточно естествен, и избегали дальнейших объяснений. Теперь настало время поговорить о вычислительных процессах и их состояниях подробнее.

Но прежде всего нужно сказать об использовании понятия процесса в современных операционных системах. Операционная система может управлять выполнением нескольких задач одновременно, и для нее отдельная задача — это какая-то безликая единица (иногда несколько). Такая единица и называется процессом.

Взаимоотношения операционной системы с процессами определяются тем, что отдельные процессы нуждаются в различных *ресурсах* вычислительной системы: во времени процессора, в каналах обмена, в ресурсе оперативной памяти. Операционная система и отвечает за организацию взаимодействия процессов, именно, за распределение исполняемой работы между процессорами, за синхронизацию действий, за обмен информацией между отдельными процессами, за синхронизацию доступа к общим данным и дефицитным ресурсам. Функционирование системы зависит прежде всего от правил действий в сложных ситуациях, правил, которые основываются на эвристических обсуждениях рационального поведения и на экспериментальной их проверке (проблематика, как мы уже видели, близкая дискретной математике). Создание операционных систем облегчается разработкой специальных программных механизмов,

удобных для описания и для программирования (семафоры, почтовые ящики — понятия из быта и техники — используются очень широко). Теперь более подробно.

Распределение времени между задачами. Сравнительно рано быстродействие компьютеров при счете существенно превысило скорости обмена с внешней средой (диск, а тем более магнитная лента или устройство ввода/вывода), и у компьютеров появилось свойство "многозадачности" (multitasking): многозадачная операционная система ведет одновременно несколько задач, предоставляя им по очереди "кванты времени" и прерывая выполнение задачи по истечении кванта, а также если эта задача запросила медленное действие обмена. При определении *политики управления задачами* в подобной операционной системе внимание разработчиков обращалось на создание достаточно простых правил переключения с задачи на задачу. В частности, были созданы системы *динамически изменяющихся приоритетов* задач.

Отдельный аспект — управление оперативной памятью, где используется так называемый *механизм подкачки страниц*. Вся свободная память делится на достаточно большие фрагменты, *страницы*. Для каждого выполняемого задания требуется определенное число таких страниц, и требуемое их число может превысить возможности компьютера. Несоответствие устраняется использованием внешней памяти — копии всех страниц хранятся на диске, и если активная в данный момент задача требует страницу, отсутствующую в оперативной памяти, страница вызывается, вытесняя другую. Получается своеобразный процесс принятия решений: выбор вытесняемой страницы зависит от управляющего блока системы. Для подкачки характерна нехватка надежной информации, решающая процедура не знает, какая страница потребуется в ближайшем будущем.

Во многих чертах управление заданиями близко к упоминавшимся уже процессам массового обслуживания, а в некоторых вычислительных системах, предназначенных для работы в реальном времени, типа транспортных касс или управления воздушным движением в аэропортах, это сходство становится полным родством.

Распределение задач между процессорами. В многопроцессорных системах внимание концентрируется на разделении заданий и их частей между отдельными процессорами. Здесь основным вопросом становится возможность ускорения выполнения задания за счет разделения работы. Для эффективного разделения работы внутри одного задания требуется, конечно, иметь больше информации о том, какова его структура, так что подготовка к разделению работы должна проводиться внутри задачи.

Захват ресурсов. Сам термин concurrent processes — *конкурирующие процессы*, который у нас обычно переводится — *параллельные процессы*, показывает, что у процессов может идти борьба за что-то. Действительно, они

нуждаются в одних и тех же ресурсах, и количество этих ресурсов ограничено. Для того чтобы процесс мог владеть захваченным ресурсом монопольно, используется специальный программный механизм, называемый семафором. Процесс может занять ресурс, только если семафор ресурса открыт. При захвате ресурса семафор закрывается, при освобождении — открывается.

При последовательном захвате нескольких ресурсов могут возникать *тупиковые ситуации* (deadlock), когда несколько ресурсов делят между собой отдельные части необходимого им ресурсного комплекта и ни один из них не обладает общим комплектом *). Здесь основной интерес — выработка достаточно простых правил поведения для опознания и разрешения таких коллизий.

10.4.1. Обычный вычислительный процесс как процесс

В отличие от процессов, описывающих какие-либо реальные явления, физические, химические или биологические, например такие, как переход электрона с одной атомной орбиты на другую, кристаллизация, деление клетки и т. п., вычислительный процесс дает своему создателю значительно большую свободу выбора состояний процесса **). При одной и той же вычислительной схеме процесс может использовать многие варианты представления данных и включать в состояние дополнительную информацию, которая предназначается для упрощения вычислений и обновления состояния.

Вспомним, как при переборе 0—1 векторов, с изменением вектора на каждой итерации ровно в одном разряде (с. 33), вместе с основным вектором мы пересчитывали вспомогательный — он был нужен для определения места изменения в основном векторе и облегчал расчеты.

При проектировании вычислительного процесса выбор структуры его состояния представляется нам одним из важнейших шагов. Поэтому не следует торопиться с этим выбором. Нужно, прежде всего, определить, какие механизмы участвуют в вычислениях, какие операции с информацией предстоит выполнять и насколько часто, как соизмеряются трудоемкость вычислений и объем памяти в данной конкретной задаче (здесь не требуется точных цифр, важен порядок данных при сопоставлении). Только после этого нужно выбирать конкретное представление данных.

Рассмотрим несколько примеров.

*) Фигурально выражаясь, один процесс надел левый ботинок, другой — правый, и теперь оба не могут дождаться освобождения парного ресурса.

**) Есть еще процессы *управления* — экономические, обучающие и др. Они занимают промежуточное положение, в них наряду с понятной "физической" обычно присутствует достаточно произвольная "вычислительная" составляющая.

Пример 1. *Алгоритм Краскала.* Вернемся к алгоритму Краскала для построения кратчайшего остовного дерева (см. с. 211). В нем идет просмотр дуг, и при проверке очередной дуги требуется установить, одной или разным компонентам связности принадлежат начало и конец этой дуги. Как уже говорилось, организовать эту проверку можно по-разному, и от формы проверки зависит структура состояния процесса.

Можно помнить и пересчитывать для каждой вершины номер компоненты, которой она принадлежит. В этом случае в состояние процесса входит только массив номеров или что-нибудь его заменяющее. Обновление информации требует полного просмотра этого массива, что дорого.

Можно дополнительно соединять вершины, принадлежащие одной компоненте, в цепной список. В этом случае проверка очень проста: сравниваются два элемента массива, и корректировка информации требует исправления в каждой вершине одной из компонент связности. Организация цепного списка требует массива ссылок вершин друг на друга.

Можно соединять вершины каждой компоненты связности не в цепной список, а в дерево. Для того чтобы вычислить номер компоненты, нужно будет просто подниматься вверх по дереву, до достижения корневой вершины, номер которой и определит компоненту. При такой организации данных мы экономим на трудоемкости исправлений и на объеме информации — достаточно для каждой вершины иметь одну ссылку вверх, а массив номеров не нужен, но увеличиваем затраты времени на вычисление номеров компонент.

Возможен и промежуточный вариант, при котором часть дерева, которая была использована для поиска компоненты связности какой-либо вершины, преобразуется, подтягиваясь ближе к корню дерева. Здесь структура состояния процесса будет такой же, как в предыдущем варианте.

Пример 2. *Алгоритм Дейкстры для поиска кратчайших путей.* В этом алгоритме, который уже рассматривался на с. 228, решающее влияние на трудоемкость оказывает способ хранения множества M_1, содержащего вершины, расстояние до которых вычисляется. Основные операции над этим множеством включают в себя:

`getnext` — извлечение из M_1 вершины с наименьшим расстоянием до нее;

`include` — перевод вершины из M_2 в M_1 с запоминанием расстояния до нее;

`update` — изменение расстояния до вершины, содержащейся в M_1;

`open` — первоначальное создание пустого множества.

Состояние процесса зависит, как уже отмечалось, главным образом от механизма работы с множеством M_1.

Мы предпочитаем организацию M_1 в виде набора "ведер", в которых хранятся исправления значений $v[i]$, $i \in M_1$. В каждом ведре хранится цепной

список таких исправлений с определенным диапазоном значений v. Когда требуется внести какое-то исправление, соответствующая запись вносится в соответствующее ведро, самым простым образом, в стековом режиме, в голову цепного списка.

Пример 3. *Задача о максимальном паросочетании.* На с. 246 подробно описан процесс вычислений, в котором используется несколько множеств вершин. Интересно посмотреть, как реализуются все эти множества и их перестройки по ходу процесса. Здесь будет описан вариант алгоритма, который более удобен в вычислительном отношении, но слишком труден для первоначального изложения. Именно, будут объединены случаи 1 и 4, а также 2 и 3, так что у нас останутся только случаи увеличения паросочетания и перевода контрольной точки с начала на конец дуги.

Всю информацию о вычислительном процессе мы можем разделить на четыре достаточно независимых объекта *).

Исходные данные — произвольно организованная совокупность пар, связывающих строки и столбцы. Эти данные не изменяются в ходе вычислений, они только просматриваются в циклическом порядке, скажем, операцией `getPair`, записывающей номера строки и столбца соответственно в `i0` и `j0`. Отдельно от этих исходных данных хранятся число строк `nR`, число столбцов `nC` и число пар `nPair`.

Контроль завершения — число `iCount` пар, просмотренных без изменения паросочетания и контрольного множества. Когда это число превысит число пар `nPair`, процесс нужно завершить.

Паросочетание — размер текущего паросочетания `nSize`, массивы номеров `RtoC`, сопоставляющий каждой строке выбранный для нее столбец, и `CtoR`, сопоставляющий каждому столбцу выбранную для него строку, булевские массивы `bRowCtl` и `bColCtl`. Массивы номеров задают (избыточно, но это удобно) связи между строками и столбцами, булевские массивы определяют контролируемые строки и столбцы. В "пустых местах" массивов номеров записано специальное "невозможное значение" `NOTVALUE`.

Стек дублеров — это стек записей, каждая из которых состоит из трех целочисленных полей. Стек имеет канал связи с внешним миром через три целочисленные переменные, две из которых, `j0` и `i0`, уже упоминались, а третья, `i1`, содержит номер строки-дублера для `i0`. Обычные операции `push` и `pop` используются, чтобы включить запись в стек из этого канала или извлечь в него запись из стека. Конечно же, существует операция начального создания стека и проверки его пустоты.

*) Попутно мы получили хорошее упражнение в объектно-ориентированном программировании. Изменения в алгоритме появились у меня под влиянием упражнений в функциональном программировании.

Теперь можно описывать алгоритм.

Построение максимального паросочетания

Состояние вычислительного процесса. Состояние процесса уже описано.

Начальное состояние. Исходные данные вводятся и готовятся к просмотру. Создается пустой *стек дублеров.* Для *контроля завершения* устанавливается нулевое значение `iCount`. Для *паросочетания* размер `nSize` полагается равным 0, все элементы массивов `RtoC` и `CtoR` получают `NOTVALUE`, все элементы массивов `bColCtl` и `bRowCtl` получают значение `False`.

Стандартный шаг.

1. Контроль завершения. Если `iCount = nPair`, завершить процесс.
2. Чтение исходных данных. Взять очередную пару, записать индекс ее строки в `i0` и индекс столбца в `j0`.
3. Проверка контролируемости пары. Если строка `i0` или столбец `j0` контролируются, увеличить счетчик `iCount`, вернуться к контролю завершения. В противном случае установить нулевое значение счетчику `iCount`.
4. Поиск элемента паросочетания со столбцом `j0`. Положить `i1` равным значению `CtoR[j0]`. Если это `NOTVALUE`, то ЭЛЕМЕНТА НЕТ. В противном случае ЭЛЕМЕНТ ЕСТЬ.
5. Случай ЭЛЕМЕНТ ЕСТЬ. Загрузить в стек новую запись. Снять пометку контроля со строки `i1` и установить ее для столбца `j0`. Вернуться к контролю завершения.
6. Случай ЭЛЕМЕНТА НЕТ. Записать индексы `i0` и `j0` во временные переменные `iTemp` и `jTemp`. Разобрать (прочесть) весь стек, выполняя для каждой записи следующее действие: если `i1 = iTemp`, то связать в паросочетании строку `iTemp` и столбец `jTemp` и заново записать индексы `i0` и `j0` во временные переменные `iTemp` и `jTemp`. Завершив разборку стека, еще раз связать в паросочетании строку `iTemp` и столбец `jTemp`. Снять контрольные пометки у всех столбцов, а пометки у строк установить только в случаях, когда у строки есть пара. Увеличить счетчик размера паросочетания.

Упражнение 10.7. Детализируйте приведенный выше алгоритм, используя описанные операции и элементы данных.

Упражнение 10.8. Пусть три замкнутых множества на прямой, A, B, C, заданы списками интервалов, как это описано на с. 114. Разработайте алгоритм для вычисления аналогичного представления множества $(A \cap B) \cup (A \cap C) \cup (B \cap C)$ с однократным просмотром исходных списков.

10.4.2. Процесс как часть алгоритма

Некоторые алгоритмические конструкции прямо выполняются в виде процессов, которые можно *создавать* и *вызывать*. Таковы, например, датчики случайных чисел, процессы обмена данных с последовательными файлами и, в частности, *информационными потоками*, процессы грамматического анализа.

Многие системные функции выполняются по аналогии с таким обменом, часто можно встретить пары действий с названиями типа `FindFirst` (*найти первое...*) и `FindNext` (*найти следующее...*). Так выглядит поиск образцов в тексте, просмотр базы данных, во многих случаях обработка списков.

Часто процессы соединяют в конвейерную линию (pipeline), в которой результаты работы одного процесса (выходной поток) принимаются другим процессом в качестве исходных данных (входной поток). Такие конвейеры характерны для операционной системы UNIX и для программ, перерабатывающих текст (компиляторы, интерпретаторы, перекодировщики).

Пример (для тех, кому нужен UNIX). Командная строка

```
ls b*f.p | awk '{print "rm " substr($1,1,length($1)-1) "q"}' | sh
```

соединяет в конвейер три действия. Первое отбирает в текущем каталоге все имена файлов с расширением p, начинающиеся на b и заканчивающиеся на f, и составляет из них список, направляя его в свой выходной поток. Второе, используя интерпретатор языка AWK (в апострофах написана исполняемая им программа), читает этот список из своего входного потока и для каждого имени файла, например для имени `blef.p`, составляет и печатает (посылает в выходной поток) команду удаления аналогичного файла с расширением q (`rm blef.q`). Третье действие — вызов программной оболочки для исполнения всех созданных команд. Конвейеры есть и в MS DOS, но используются редко, а для UNIX это обычное средство.

Процессами удобно представлять отдельные механизмы алгоритма, и объектно-ориентированный стиль программирования облегчает такое представление, причем для многих типовых механизмов имеются готовые программные компоненты. Есть задачи, в которых сам алгоритм удобнее конструировать в терминах взаимодействующих процессов.

Так, в задачах дискретной оптимизации из-за несовершенства рассматриваемых моделей, т.е. из-за того, что красивая математическая модель оказывается не вполне соответствующей "грубой реальности",

часто случается, что оптимальное решение математической задачи для практики неприемлемо. Один из возможных выходов из этого положения — строить *субоптимальные решения*: находить не одно какое-то оптимальное решение, и даже не все оптимальные решения, а последовательность решений, т.е. перебирать их начиная от оптимального и далее в порядке ухудшения значения целевой функции до тех пор, пока не найдется решение, удовлетворяющее практическим требованиям. Для перебора субоптимальных решений удобно создавать специальные процессы — *перечислители*. Оказывается, удобно строить перечислители друг из друга, с помощью специальных действий. Этих действий оказывается совсем немного, они очень просты и легко программируются [42].

Легко построить пример такого процесса, в котором заранее задано некоторое множество чисел и они выдаются по запросу в порядке возрастания (неубывания) *). Ничего сложного в таком механизме нет — мы создадим и заполним этими числами упорядоченный массив и будем при нем хранить *указатель текущего индекса*, сдвигающийся на единицу при каждом вызове. Пример очень полезен как способ задания исходных данных при формировании более сложных процессов.

Теперь введем действия над процессами.

Элементарное преобразование решения за счет *прибавления* к текущему значению константы и вообще вычисления любой монотонно неубывающей функции от текущего значения служат примером такого преобразования процесса в процесс.

Вполне естественно вводится операция *слияния* двух или большего числа процессов, аналогичная операции слияния упорядоченных массивов, с которой мы встречались в параграфе 4.8 и при описании сортировки фон Неймана (с. 153): имеется несколько перечислителей и известны очередные значения, "предлагаемые" каждым из них; при каждом вызове операции из этих очередных значений выбирается наименьшее, оно объявляется результатом операции, а у перечислителя, который произвел это значение, вызывается очередное значение. Обозначим эту операцию через MERGE(P, Q), если сливаются процессы P и Q.

В терминах уже названных операций легко полностью описать всю сортировку фон Неймана (понимая ее как перечислитель, вырабатывающий в порядке возрастания элементы массива): разобьем исходный

*) Выбрав порядок выдачи чисел по возрастанию, мы определили, что будем рассматривать субоптимальные решения в задаче минимизации.

массив на упорядоченные отрезки и для каждого из них образуем процесс перебора, получившиеся процессы сольем попарно и т. д.

Таким образом, имея, например, массив

```
16, 31, 37; 11, 19, 25; 3, 14, 18, 27, 51; 28, 17, 12;
```

мы разобьем его на четыре упорядоченных куска R_1, R_2, R_3 и Q_4 (они завершаются знаком ;), причем в качестве Q_4 взят для разнообразия кусок, упорядоченный по убыванию, и его мы будем просматривать с конца. Процесс $P = \mathsf{MERGE}(R_1, R_2, R_3, Q_4)$ — это перечислитель, который выдает результат.

Если бы мы хотели пользоваться только слиянием пар процессов, то могли бы действовать следующим образом. Определим процессы $P_{12} = \mathsf{MERGE}(R_1, R_2)$, $P_{34} = \mathsf{MERGE}(R_3, Q_4)$ и $P = \mathsf{MERGE}(P_{12}, P_{34})$. При первом вызове процесса P он вызовет процессы P_{12} и P_{34}, P_{12} вызовет процессы R_1 и R_2, получит значения 16 и 11 и выберет 11 как меньшее, P_{34} вызовет процессы R_3 и R_4, получит значения 3 и 12 и выберет из них 3, наконец, процесс P, получив 11 и 3, выберет 3 как первое значение отсортированного массива. Получение каждого следующего значения потребует только замены использованных значений: P вызовет только P_{34}, а тот, в свою очередь, только R_3.

Вот еще одна практическая ситуация: на плоскости задается точка P_0, и плоскость разбивается на квадраты одинакового размера, например единичные с целочисленными координатами вершин. Пусть для определенности каждый квадрат идентифицируется координатами левого нижнего угла, например $Z_{3,2}$. Расстоянием от точки P_0 до квадрата называется расстояние от P_0 до ближайшей точки квадрата. Требуется создать процесс, перечисляющий квадраты в порядке увеличения их расстояния до P_0.

Разработка этого процесса сильно упрощается, если выделить квадрат Z_P, содержащий P_0, а остальные квадраты разбить на 8 множеств, как это показано на рис. 10.5, где в каждом множестве показаны направления изменения координат (так, $(+, -)$ — x возрастает, а y убывает; $(\bullet, +)$ — y возрастает, а x не меняется). Для каждого из этих квадратов разрабатывается свой процесс перебора, а операция слияния объединяет их *).

*) Эта задача с небольшими изменениями предлагалась на Всероссийской олимпиаде по программированию для школьников в 1999 г. Более простым в реализации оказался другой метод, сочетающий накопление в буфере больших "порций" квадратов, их сортировку и выдачу по требованию.

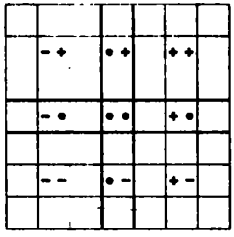


Рис. 10.5. Разбиение плоскости на 8 множеств

Еще одно важное действие над перечислителями — это *фильтрация*, отбор значений. Проще всего описать фильтрацию на примере. Представьте себе, что задан граф и требуется найти кратчайший путь из вершины i_0 в вершину i_1, содержащий дугу j. Будем перебирать пути из i_0 в i_1 в порядке возрастания длины и делать это до тех пор, пока не встретится путь, содержащий j. Превратим это единичное действие в процесс, который при каждом запросе продолжает свою работу, — он будет перечислителем путей из i_0 в i_1, содержащих j.

Вот еще одна операция, которая реализуется сложнее, но значительно увеличивает круг решаемых таким способом задач. Представьте себе, что мы рассматриваем все значения, вырабатываемые процессом P_1, и все значения, вырабатываемые процессом P_2, составляем их всевозможные суммы и формируем процесс, который просматривает эти суммы в порядке возрастания. Операция над P_1 и P_2, образующая этот новый процесс, называется их *суммированием*.

Суммирование процессов очень просто программируется, если использовать *границу Парето* (мы встречали уже этот термин). Ведь мы рассматриваем декартово произведение двух множеств — значений обоих процессов. На каждом шаге процесс-сумма просматривает еще не выбранные пары и выбирает из них ту, для которой значение суммы наименьшее. Очевидно, что нет смысла перебирать все множество оставшихся пар, достаточно просматривать только его границу Парето. Покажем использование этой границы на примере.

Пример. Пусть наши процессы — это просто два массива чисел с неубывающими значениями, скажем $P_1 = [4, 11, 14, 19, 26, 28]$ и $P_2 = [6, 9, 12, 17, 23, 25, 44, 51]$, в проектируемом нами процессе должно быть $6 \times 8 = 48$ значений.

Но для выработки первого значения не требуется перебирать все 48 возможностей, первое значение получается как сумма минимальных значений в P_1 и P_2, это $4 + 6 = 10$; единственная пара $(1, 1)$ составляет границу Парето

множества $(1:6) \times (1:8)$. После удаления (использования) пары $(1,1)$ границу Парето составляют две пары $(1,2)$ и $(2,1)$, при очередном запросе нужно выбирать между $4+9=13$ и $11+6=17$, выбирается $(1,2)$. Теперь границу составляют три пары: $(1,3)$, $(2,2)$ и $(2,1)$, из которых пара $(2,2)$ в границу Парето не входит, так как мажорируется парой $(2,1)$. В третий раз выбор идет между $4+12=16$ и $11+6=17$, выбирается $(1,3)$. Можно составить таблицу, которую вы сами легко можете продолжить дальше.

k	Граница Парето	Выбор	Значение
1	$(1,1):10$	$(1,1)$	$4+6=10$
2	$(1,2):13, (2,1):17$	$(1,2)$	$4+9=13$
3	$(1,3):16, (2,1)$	$(1,3)$	$4+12=16$
4	$(1,4):21, (2,1)$	$(2,1)$	$11+6=17$
5	$(1,4), (2,2):20, (3,1):20$	$(2,2)$	$11+9=20$
6	$(1,4), (2,3):23, (3,1)$	$(3,1)$	$14+6=20$
7	$(1,4), (2,3), (3,2):23, (4,1):25$	$(1,4)$	$4+17=21$
8	$(1,5):27, (2,3), (3,2), (4,1)$	$(2,3)$	$11+12=23$
9	$(1,5), (2,4):28, (3,2), (4,1)$	$(3,2)$	$14+9=23$
10	$(1,5), (2,4), (3,3):26, (4,1)$	$(4,1)$	$19+6=25$

Видите, как просто идут вычисления. И нет никакой разницы, откуда берутся исходные данные: из пары массивов или из других перечислителей.

Операция сложения перечислителей чрезвычайно важна: если множество A представлено как прямое произведение двух множеств $A = A_1 \times A_2$ и целевая функция аддитивна в том смысле, что $f(a) = f((a_1, a_2)) = f_1(a_1) + f_2(a_2)$, то перечислитель элементов A является суммой перечислителей для A_1 и A_2.

К сожалению, нет места для описания этой темы в бо́льших подробностях. Отметим только, что в [42] описаны особенности перебора субоптимальных решений для ряда экстремальных задач, которые мы рассматривали здесь, в частности для перебора остовных деревьев в порядке возрастания их длины, перебора корневых деревьев, кратчайших путей между двумя вершинами графа, назначений в задаче о назначениях и др. Но одну из этих задач хочется предложить читателю в виде упражнения.

Упражнение 10.9. Придумайте способ перечисления перестановок в порядке возрастания целевой функции задачи о минимуме скалярного произведения, описанной на с. 42.

Упражнение 10.10. Разработайте реализацию операции суммирования перечислителей, использующую границу Парето.

10.4.3. Сопрограммы

Когда выше мы рассматривали взаимодействующие процессы, в их взаимодействии была видна некоторая подчиненность, подобная подчиненности вызовов процедур в обычных программах: один процесс был вызывающим, "хозяином", а другой — вызываемым, "слугой". Значительно богаче возможностями и интереснее ситуации, когда алгоритмы описываются в терминах процессов, взаимодействующих более равноправно.

Наиболее отчетливо взаимодействие отдельных процессов встретилось нам при рассмотрении алгоритма Дейкстры для поиска кратчайших путей, именно — при рассмотрении "ведерного" механизма хранения исправлений. Посмотрим теперь на этом примере, что́ может означать равноправие процессов.

Пример. *Алгоритм Дейкстры* (продолжение). В этом алгоритме взаимодействуют два процесса: процесс D, ведущий вычисления "по Дейкстре", и процесс M, хранящий информацию о множестве M_1. Их взаимодействие определяется перечисленными при описании стандартными действиями. Сначала напомним, как процессы работают по-старому.

Процесс D, главный процесс, можно описать так (сделаны некоторые упрощения, в частности введен массив кратчайших расстояний, всем элементам которого, кроме начальной вершины $i0$, сопоставлены бесконечные значения).

```
for ( i in M) do v[i] := infinity od; % бесконечные расстояния до
v[i0] := 0;                           %  всех дуг кроме начальной
saveUpdate(i0,0);                     % запомнить исправление
while ( askUpdate(vUpdate) ) do
   i := vUpdate.vert;                 % вершина этого исправления
   for ( arc in ArcsFrom(i) ) do      % просмотр выходящих дуг
     i1 := endvert(arc);              % конец дуги arc
     v1 := v[i] + cost[arc];          % расстояние до i1
     if ( v1 < v[i1] )  then          % если оно меньше прежнего
       v1 := v1; inarc[i1] := arc;
       saveUpdate(i1,v1)              % запомнить исправление
     fi
od od
```

Процесс D в цикле запрашивает исправления у процесса M и, пока получает эти исправления, обычным образом их обрабатывает.

Ритм работы процесса M подчинен потребностям вызывающего процесса D, и следовательно, M работает от запроса до запроса, и отдельный шаг

его деятельности выглядит примерно так:

```
clean(resUpdate);              % место для результата поиска
continueSearch := True;
while ( continueSearch ) do % цикл поиска
                                        % если ведро пустое,
if ( emptyBucket(kBucket) ) then    % сменить ведро
  continueSearch := getNextBucket(kBucket)
                        % отсутствие ведра прекращает поиск
else
  vUpdate := getUpdate(kBucket);  % взять очередное исправление
  if ( valid(vUpdate) ) then       % если оно не устарело
    saveUpdate(vUpdate,resUpdate);
      % вернуть найденное исправление как результат
    continueSearch := FALSE        % и кончить цикл поиска
  fi;
  collectGarbageUpdate(vUpdate)    % собрать мусор
 fi
od
return resUpdate;
```

Приведены только существенные детали. Фрагмент использует некоторые статические переменные, которые входят в состояние рассматриваемого процесса и при инициализации должны получить свои начальные значения. Обратите внимание на сложные многоступенчатые действия при поиске очередного исправления: поиск в том же ведре, что и раньше, поиск нового ведра, проверка существования еще одного ведра.

Поглядите, насколько удобнее было бы описывать действия процесса *M* с его точки зрения, насколько упрощается логика просмотра: просматриваются подряд все ведра, из каждого ведра выбираются одно за другим все исправления, отбрасываются "устаревшие", т.е. относящиеся к уже обработанным вершинам, а не устаревшие передаются по очередному запросу в качестве результата операции.

Эту схему можно описать так, что будут ясно видны циклы просмотра ведер и исправлений.

```
for (kBucket in SetOfBukets) do  % просматриваем все ведра
                        % для каждого исправления в ведре
  for ( vUpdate in SetOfUpdates(kBucket) ) do
                        % если это исправление не устарело
  if ( valid(vUpdate) )  then
    useUpdate(vUpdate)  % передать его для использования
  fi;
  collectGarbageUpdate(vUpdate) %  собрать мусор
od od
```

Проблема только в том, как наладить согласование этих двух процессов. Выход нужно искать в программных средствах для создания таких процессов, которые исполняются независимо друг от друга и обмениваются друг с другом информацией, *не прекращая своей работы, а только приостанавливая ее*. В таких терминах мы должны представлять себе следующую структуру действия `useUpdate` у процесса *M*:

```
useUpdate(vUpdate) :=:      % эквивалентность текстов
  send vUpdate;             % передача информации во внешний мир
  wait;                     % пауза, по завершении которой
                            % процесс продолжится
```

Все дело в пересылке информации и паузе. В свою очередь, и у процесса *D* должна быть возможность делать паузу, которая должна находиться внутри действия `askUpdate`. Оно, как и `useUpdate`, предусматривает общение с другим процессом:

```
askUpdate(vUpdate) :=:
  wait;                     % пауза, по завершении которой
                            % процесс продолжится
  receive vUpdate;          % получение информации из внешнего мира
```

Посмотрите, насколько просты стали эти процессы и их взаимодействие.

Для оформления подобных связей между процессами в современных операционных системах существует конструкция, называемая *нитью управления*[*]. Отдельным процессам выделяются нити управления, и эти нити могут приостанавливать и продолжать работу своих процессов.

В нашем примере нам следует просто завести две нити — для *D* и для *M*.

Отметим, что программирование, ориентированное на прерываемые и возобновляемые процессы, становится почти неизбежным при создании диалоговых систем (типа Windows-приложений), и еще предстоит найти для него удачные средства. И здесь многонитевое проектирование в виде независимых процессов, ведущих совместную работу с прерываниями и возобновлениями, может быть перспективным, особенно, если учесть появление процессоров, поддерживающих специально работу со многими нитями (HyperThreading).

[*] Исходный термин thread (англ.), к сожалению, часто переводится как "поток управления". Это не только увеличивает лингвистическую нагрузку на термин "поток", неизбежный при переводе терминов stream и flow, но и затрудняет нам понимание сопутствующих термину визуальных и иных ассоциаций (например, в Интернете значок с катушкой ассоциируется, как получается, с *потоком* писем на одну тему, а мы бы сказали, что с *нитью беседы*).

Глава 11

Связи дискретного и непрерывного анализа

11.1. Введение. Конкретная математика

Материал этого короткого раздела перекликается с обстоятельной и очень интересной книгой Р. Грэхема, Д. Кнута и О. Паташника *Конкретная математика* [16]. Слово CONCRETE здесь используется не в своих обычных значениях (кроме КОНКРЕТНЫЙ это еще и БЕТОННЫЙ), а как комбинация слов CONtinuous и disCRETE. Авторы предложили эту книгу в качестве материала, продолжающего начальный курс дискретной математики.

Поскольку мы не вправе рассчитывать на то, что курс действительно будет продолжен, хочется хотя бы упомянуть о проблематике и подходах конкретной математики и в начальном курсе. Здесь мы рассмотрим, и только на примерах, всего два вопроса. Это *производящие функции* и *асимптотика* формул.

11.2. Производящие функции

Сначала мы познакомимся с одной очень продуктивной идеей нахождения явного вида общего члена числовых последовательностей, основанной на идеях классического анализа. Эта идея называется методом *производящих функций* — мы встречались с ней, когда обсуждали

формулу Муавра и ее применения к различным соотношениям между биномиальными коэффициентами.

11.2.1. Общая идея

Общая идея очень проста. Пусть задана последовательность $\{a_n\}$, и мы хотим найти общую формулу для a_n. Введем формально функцию

$$\mathcal{A}(x) = \sum_n a_n x^n.$$

Иногда удается использовать свойства последовательности $\{a_n\}$ для получения в каком-либо виде уравнения для функции $\mathcal{A}(x)$. Если нам удастся решить это уравнение и обосновать выкладки (уже зная результат), мы можем из разложения результата в степенной ряд получить элементы искомой последовательности.

11.2.2. Числа Фибоначчи

Здесь мы найдем точную формулу для чисел F_n, введенных на с. 57; она выглядит несколько неожиданно.

Следуя общей идее, рассмотрим производящую функцию

$$\mathcal{F}(x) = \sum_{n=0}^{\infty} f_n x^n.$$

Выполняя формальные преобразования, имеем

$$\begin{aligned}\mathcal{F}(x) &= f_0 + f_1 x + \sum_{n=2}^{\infty} f_n x^n = \\ &= f_0 + f_1 x + \sum_{n=2}^{\infty} f_{n-1} x^n + \sum_{n=2}^{\infty} f_{n-2} x^n = \\ &= f_0 + f_1 x + \sum_{n=1}^{\infty} f_n x^{n+1} + \sum_{n=0}^{\infty} f_n x^{n+2}.\end{aligned}$$

Мы видим, что вторая сумма — это в точности $x^2\mathcal{F}(x)$, а первая — это $x\big(\mathcal{F}(x) - f_0\big)$. Таким образом, мы имеем

или

$$\mathscr{F}(x) = 1 + x + x^2\mathscr{F}(x) + x\big(\mathscr{F}(x) - 1\big)$$

и окончательно:

$$\mathscr{F}(x) = \frac{1}{1 - x - x^2}.$$

Эту функцию легко разложить в степенной ряд. Действительно,

$$1 - x - x^2 = (1 - \alpha x)(1 - \beta x),$$

где

$$\alpha = \frac{1 + \sqrt{5}}{2}, \qquad \beta = \frac{1 - \sqrt{5}}{2}.$$

Представляя $\mathscr{F}(x)$ в виде

$$\mathscr{F}(x) = \frac{A}{1 - \alpha x} + \frac{B}{1 - \beta x},$$

получаем

$$A = \frac{\alpha}{\alpha - \beta}, \qquad B = \frac{-\beta}{\alpha - \beta},$$

и так как для малых x

$$\frac{1}{1 - \gamma x} = \sum_{k=0}^{\infty} \gamma^k x^k,$$

окончательно имеем

$$\mathscr{F}(x) = \sum_{k=0}^{\infty} \frac{\alpha^{k+1} - \beta^{k+1}}{\alpha - \beta} x^k.$$

Итак,

$$f_k = \frac{1}{\sqrt{5}}\left[\left(\frac{1 + \sqrt{5}}{2}\right)^{k+1} - \left(\frac{1 - \sqrt{5}}{2}\right)^{k+1}\right].$$

Ясно, что ряд для $\mathscr{F}(x)$ абсолютно сходится при достаточно малых x, и все выкладки обоснованы. Вы можете проверить непосредственно, что числа f_n действительно удовлетворяют первоначальному соотношению.

Предел отношения f_{n+1}/f_n при $n \to \infty$ равен, очевидно, α. Это число известно как "золотое сечение": если от прямоугольника, у которого отношение длин сторон равно α, отрезать квадрат, то у оставшегося прямоугольника отношение длин сторон будет таким же.

11.2.3. Числа Каталана

Другая известная последовательность — числа Каталана *). Эти числа появляются в следующей задаче.

Предположим, что требуется вычислить сумму $S_0 + S_1 + \cdots + S_n$ и можно складывать любые два рядом стоящих числа и ставить результат на их место. Требуется найти число различных последовательностей действий.

Отметим, что каждая такая последовательность действий может быть представлена двоичным деревом с $n+1$ терминальной вершиной, а также "правильной" расстановкой n пар скобок.

Обозначим искомое число способов через c_n. Ясно, что последнее вычисление в любом случае будет сложением сумм для двух последовательных подмножеств вида $0:k$ и $(k+1):n$, и схемы с различными k принципиально различны. Но для данного k мы должны иметь c_k вариантов левой части и c_{n-k-1} вариантов правой части, так что

$$c_n = \sum_{k=0}^{n-1} c_k c_{n-k-1},$$

где $c_0 = 1$. Введем производящую функцию

$$C(x) = \sum_{n=0}^{\infty} c_n x^n$$

и рассмотрим формально представление для ее квадрата. Имеем

$$C^2(x) = \sum_{m=0}^{\infty} c_m x^m \times \sum_{n=0}^{\infty} c_n x^n =$$

$$= \sum_{m,n=0}^{\infty} c_m c_n x^{m+n} = \sum_{r=0}^{\infty} \sum_{m=0}^{r} c_m c_{r-m} x^r = \sum_{r=0}^{\infty} c_{r+1} x^r.$$

Таким образом, учитывая, что $c_0 = 1$, имеем квадратное уравнение

$$C(x) = xC^2(x) + 1,$$

решение которого дает

$$C(x) = \frac{1 \pm \sqrt{1-4x}}{2x}.$$

*) Eugène Charles Catalan (1814–1894), бельгийский математик, член-корреспондент Петербургской академии наук.

Разложение в ряд для функции $f(x) = \sqrt{1-4x}$ получается непосредственно по формуле Тейлора

$$f(x) = f(0) + \sum_{k=1}^{\infty} \frac{f^{(k)}(0)}{k!} x^k.$$

Имеем

$$\frac{d^k}{dx^k}(1-4x)^{\frac{1}{2}} = \frac{1}{2} \cdot \left(\frac{1}{2} - 1\right) \dots \left(\frac{1}{2} - k + 1\right)(1-4x)^{\frac{1}{2}-k}(-4)^k =$$

$$= -2^k \cdot 1 \cdot 3 \cdots (2k-3)(1-4x)^{\frac{1}{2}-k} = -2(k-1)!\, C_{2k-2}^{k-1}\,(1-4x)^{\frac{1}{2}-k}$$

и окончательно

$$f(x) = 1 - 2\sum_{k=1}^{\infty} \frac{1}{k} C_{2k-2}^{k-1} x^k.$$

Подставляя этот ряд в формулу для $C(x)$, мы должны сначала выбрать знак перед квадратным корнем (это нетрудно, знак минус нас устраивает больше) и провести необходимые выкладки. Получаем окончательно

$$C(x) = \sum_{k=1}^{\infty} \frac{1}{k} C_{2k-2}^{k-1} x^{k-1} = \sum_{k=0}^{\infty} \frac{1}{k+1} C_{2k}^{k} x^k.$$

11.3. Асимптотика

В заключение курса рассмотрим один интересный пример предельного поведения комбинаторного выражения.

Рассмотрим формулу для числа успехов в n испытаниях в схеме Бернулли: если вероятность успеха в одном испытании равна p, то вероятность m успехов в n независимых испытаниях равна

$$P_n(m) = C_n^m p^m (1-p)^{(n-m)}.$$

Как мы знаем, математическое ожидание числа успехов равно np, а дисперсия $np(1-pq)$, так что будет достаточно естественно рассматривать значения m, близкие к np. Обозначим $1-p=q$ и введем новую переменную $\delta_m = m - np$. Тогда

В формуле

$$P_n(m) = \frac{n!}{m!(n-m)!}\, p^m (1-p)^{(n-m)}$$

применим формулу Стирлинга*) для $k!$

$$k! = (2\pi)^{\frac{1}{2}} k^{k+\frac{1}{2}} e^{-k}(1 + o(k))$$

и получим

$$P_n(m) \approx \left\{\frac{n}{2\pi m(n-m)}\right\}^{\frac{1}{2}} \left(\frac{np}{m}\right)^m \left(\frac{nq}{n-m}\right)^{(n-m)} =$$

$$= \left\{\frac{n}{2\pi(np+\delta_m)(nq-\delta_m)}\right\}^{\frac{1}{2}} \left(1+\frac{\delta_m}{np}\right)^{-(np+\delta_m)} \left(1-\frac{\delta_m}{nq}\right)^{-(nq-\delta_m)}.$$

Выделим последние два сомножителя. Логарифмируя их и обозначая результат через T, получаем

$$T = -(np+\delta_m)\ln\left(1+\frac{\delta_m}{np}\right) - (nq-\delta_m)\ln\left(1-\frac{\delta_m}{nq}\right).$$

Для $x < 1$ имеем

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots,$$

так что (нам будет достаточно двух первых членов разложения)

$$T = -(np+\delta_m)\left(-\frac{\delta_m}{np} - \frac{\delta_m^2}{2n^2p^2} + \cdots\right) - (nq-\delta_m)\left(-\frac{\delta_m}{nq} - \frac{\delta_m^2}{2n^2q^2} + \cdots\right).$$

Мы видим, что члены, содержащие δ_m в первой степени, пропадают

$$T_1 = -np\frac{\delta_m}{np} + nq\frac{\delta_m}{nq} = 0,$$

и главным становится член второго порядка

$$T_2 = -\frac{\delta_m^2}{np} + \frac{\delta_m^2}{2np} - \frac{\delta_m^2}{nq} + \frac{\delta_m^2}{2nq} = -\frac{\delta_m^2}{2n}\left(\frac{1}{p}+\frac{1}{q}\right) = -\frac{\delta_m^2}{2npq}.$$

*) James Stirling (1692–1770), шотландский математик.

Пренебрегая элементами, содержащими δ_m в более высоких степенях, имеем

$$P_n(m) \approx \frac{1}{\sqrt{2\pi npq}} \exp\left(-\frac{\delta_m^2}{2npq}\right).$$

Введем обозначения

$$h = (npq)^{-1/2}, \qquad x_m = h\delta_m.$$

Окончательно получаем следующую теорему.

Теорема (Муавр–Лаплас). Если стремление $m, n \to \infty$ таково, что $x_m^3 n^{-1/2} \to 0$, то справедливо следующее асимптотическое отношение:

$$P_n(m) \approx \frac{h}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right).$$

Изучение асимптотического поведения комбинаторных выражений, точное и оценочное, чрезвычайно важно для оценки эффективности алгоритмов.

Приложение

Библиографические рекомендации

Почти по всем обсуждаемым в этой книге вопросам можно рекомендовать переизданное в 2000 г. многотомное сочинение Д. Кнута *Искусство программирования* [20–22] и объемистую книгу Т. Кормена, Ч. Лейзерсона и Р. Ривеста *Алгоритмы: построение и анализ* [28] *).

Учебников по дискретной математике в последнее время появилось очень много, их тематика удивительно разнообразна, так как согласия о предмете этой дисциплины еще нет. Интересна приложениями дискретной математики доволько старая книга [21]

К главе 1 (Некоторые определения из теории множеств)

Общие определения теории множеств появляются сейчас чуть ли не в младших классах школы. На всякий случай назовем книгу Н. К. Верещагина и А. Шеня *Начала теории множеств* [11] (текст этой книги доступен через Интернет).

К главе 2 (Строки фиксированной длины)

В связи с определением многомерного куба хотелось бы особо отметить, как удачно используется это понятие при построении многопроцессорных вычислительных комплексов. Представив себе, что, например, 32 отдельных процессора помещены в вершинах 5-мерного куба, мы легко поймем слова о том, что

*) Недавно Д. Кнут начал публиковать выпусками материалы долгожданного четвертого тома [70, 26], а издательство "Вильямс" оперативно (на мой взгляд, даже слишком) издает переводы этих выпусков. Со второй книгой тоже произошли изменения — добавился еще один автор, К. Штайн. Большего сказать не могу, так как до нового издания еще не добрался.

связи этих процессоров по ребрам куба образуют остовное дерево (рис. П.1). Такая связь процессоров в больших системах сейчас широко используется. Первое известное нам предложение о такой организации вычислительных машин было сделано в 1962 г. новосибирским геометром, выпускником Ленинградского университета (ныне академиком РАН) Ю. Г. Решетняком [41].

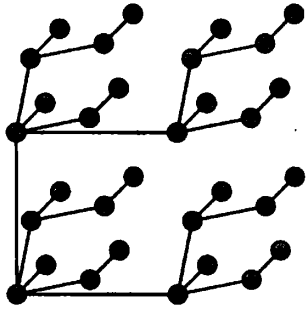


Рис. П.1. Связь процессоров по ребрам 5-мерного единичного куба

По вопросам кодирования, включая историю кодов, можно рекомендовать обширную книгу Чарльза Петцольда *Код* (М.: Микрософт Пресс, 2001. 495 с.). Многие специальные коды (Unicode, штрих-коды, код Брайля, стандартный формат вещественной арифметики) хорошо описаны в Интернете.

Книг по комбинаторике очень много. Можно назвать сильно различающиеся по содержанию и стилю изложения книги М. Айгнера *Комбинаторная теория* (М.: Мир, 1982), Н. Я. Виленкина *Комбинаторика* (М.: Наука, 1969), А. Кофмана *Введение в прикладную комбинаторику* (М.: Наука, 1975), Дж. Риордана *Введение в комбинаторный анализ* (М.: Изд-во иностр. лит., 1963), К. А. Рыбникова *Введение в комбинаторный анализ* (М.: Изд-во МГУ, 1972), М. Холла *Комбинаторика* (М.: Мир, 1970). Упомянем особо очень интересную книгу Р. Стенли *Перечислительные задачи в комбинаторике* (М.: Мир, 1990), содержащую много свежего материала.

Вопросы перебора комбинаторных объектов нужно смотреть в вышедших выпусках 4-го тома Д. Кнута [26]. Один из этих выпусков полностью посвящен перебору размещений, о котором мы не смогли даже упомянуть. А в выпуске, посвященном перебору 0–1 векторов (кортежей), можно найти исчерпывающую информацию о коде Грея и его истории.

По-видимому, лучшим из небольших пособий по вычислительной комбинаторике является книга Витольда Липского [33].

По математической логике книг много. Назовем наш университетский учебник Н. К. Косовского [29].

О Блезе Паскале и его занятиях теорией вероятностей можно много интересного прочесть в небольшой книге Альфреда Реньи *Письма о вероятности* (М.: Мир, 1970).

По поводу чисел Фибоначчи лучше всего адресоваться к брошюре Н. Н. Воробьева [12].

К главе 3 (Элементарная теория вероятностей)

По теории вероятностей есть много замечательных книг. Можно рекомендовать элементарное введение в теорию вероятностей Б. В. Гнеденко и А. Я. Хинчина [15], книгу Ю. Неймана [38], двухтомник В. Феллера [49]. Очень рекомендую небольшой задачник Ф. Мостеллера [36].

Разговор о теории вероятностей мы продолжим в связи с теорией марковских цепей, которая излагается в главе 10.

К главе 4 (Строки переменной длины)

Очень много информации о строковых алгоритмах с применениями к задачам дешифровки "биологических строк" — информационным цепочкам ДНК — содержится в книге Д. Гасфилда [13], перевод которой недавно издан. В частности, в этой книге излагается вариант метода Бойера–Мура, предложенный Апостолико и Джанкарло. Их метод требует линейного времени работы и линейной памяти.

По регулярным выражениям и их использованию в программировании есть специальная книга Дж. Фридла *Регулярные выражения* (*Библиотека программиста*. СПб.: Питер, 2001).

Книги по динамическому программированию см. в рекомендациях к главе 10.

Про технику работы с разреженными матрицами можно прочесть в книгах [47] и [39]. Специальные вопросы работы с такими матрицами в алгоритмах линейного программирования (там требуется специфический набор операций) см., например, в [37].

К главе 5 (Сжатие и защита информации)

Книгу В. В. Семенюка я уже упоминал. Жалко, что она труднодоступна из-за малого тиража, но можно надеяться, что ее переиздадут. Более доступна и покрывает более широкий круг вопросов только что вышедшая книга Д. Ватолина и др. *Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео* (М.: Диалог-МИФИ, 2002).

Много разнообразных способов кодирования описано в книге Бауэра и Гооза [7].

По кодам, исправляющим ошибки, можно назвать книги Р. В. Хэмминга *Теория кодирования и теория информации* (М.: Радио и связь, 1983); Ф. Дж. Мак-Вильямса, Н. Дж. Слоэна *Теория кодов, исправляющих ошибки* (М.: Связь, 1979); У. Питерсона, Э. Уэлдона *Коды, исправляющие ошибки* (М.: Мир, 1976).

По криптографии можно рекомендовать вышедшую в 1999 г. вторым изданием книгу под редакцией В. В. Ященко [10], но в последнее время книг по криптографии стало в книжных магазинах много. Особо отметим доступную через Интернет книгу А. Менезеса, П. ван Ооршота и С. Вэнстона *Handbook of Applied Cryptography* [77] и обзор асимметричных алгоритмов в статье А. В. Лунина и А. А. Сальникова *Перспективы развития и использования асимметричных алгоритмов* (`http://www.ssl.stu.neva.ru/psw/crypto/Lunin24.html`).

К главе 6 (Информационный поиск и организация информации)

Сортировки описаны во многих книгах. Сошлемся опять на третий том издания Д. Кнута, в котором вы найдете богатый материал по всем вопросам, затронутым в этой главе. По сортировкам можно еще назвать книгу Г. Лорина *Сортировка и системы сортировки* (М.: Наука, 1983).

Современные методы построения суффиксного массива описаны в статье К.-Б. Шурмана и Й. Стое [80]. Авторы приводят интересную статистику времени построения суффиксного массива для больших строк на разных вариантах программ. Так, если всю Библию представить в виде одной строки длиной 4 MB при 63 различных символах, то на персональном компьютере с 512 MB памяти и процессором Intel со скоростью 1.3 GHz под операционной системой Linux суффиксный массив строится за 1.5–2 секунды, а для человеческой хромосомы № 22 (32 MB при 5 различных символах) — за 16–40 секунд.

К главе 8 (Теория графов)

Из книг по теории графов мне больше всего нравится книга К. Бержа [8]. Назовем еще недавно вышедшую книгу В. А. Емеличева и др. *Лекции по теории графов* (М.: Наука, 1990), книги О. Оре *Теория графов* (М.: Наука, 1968) и А. А. Зыкова *Основы теории графов* (М.: Наука, 1987). Совсем недавно вышла большая книга В. Н. Касьянова и В. А. Евстигнеева *Графы в программировании* (СПб.: БХВ-Петербург, 2003. 1104 с.).

Преимущественно экстремальным задачам на графах посвящены книги Ю. М. Ермольева, И. М. Мельника *Экстремальные задачи на графах* (Киев: Наукова думка, 1968), Д. Филлипса, А. Гарсиа-Диаса *Методы анализа сетей* (М.: Мир, 1984), Л. Форда, Д. Фалкерсона *Потоки в сетях* (М.: Мир, 1966).

Работы Э. Дейкстры на английском языке можно найти в Интернете по адресу `http://www.cs.utexas.edu/users/EWD`.

Приложения теории графов к электротехническим расчетам описаны (кроме подробного изложения общей теории графов) в книге М. Свами, К. Тхуласирамана *Графы, сети и алгоритмы* (М.: Мир, 1984).

Литература по сетевому планированию давно не пополнялась. Из многочисленных изданий прежних лет можно назвать книгу С. И. Зуховицкого и И. А. Радчик [19]. Для первоначального знакомства с теорией расписаний хорошо подходит книга В. С. Танаева и В. В. Шкурбы *Введение в теорию расписаний* (М.: Наука, 1975).

Интернет наполнен примерами "сетевых графиков", которые состоят из тоскливого перечня "работ", взаимосвязи которых никак не уточняются. Полезно посмотреть в интернете же статью Евгения Васильчука "В марафон по графику" из "Российской Бизнес-газеты", № 525 от 27 сентября 2005 г. (сайт `http://www.rg.ru/2005/09/27/setevye-grafiki.html`). Встречавшиеся нам описания популяризируемой системы Microsoft Project уделяют основное внимание техническим возможностям этого программного продукта и мало подходят для обучения сути дела.

По аналогии с графами были введены *гиперграфы*, о которых можно прочитать в обзорной статье А. А. Зыкова (Успехи матем. наук. 1974. Т. 29. Вып. 6. С. 89–154) и в небольшой книге Ю. А. Сушкова *Связность гиперграфов* (СПб.: Изд-во СПбГУ, 2002. Тир. 190 экз.).

К главе 9 (Экстремальные задачи)

Нельзя не упомянуть классическую работу Л. В. Канторовича [20] об использовании принципа двойственности и двойственных переменных в экономических исследованиях. Эта книга стала библиографической редкостью, но сейчас готовится новое издание.

Что же касается задачи о максимальном потоке через сеть, то наиболее эффективным считается алгоритм Push-Relabel, разработанный Б. В. Черкасским и А. В. Гольдбергом [60]. Недавно мне довелось познакомиться с интересным приложением этого алгоритма — для задачи определения формы котлована при разработке месторождения открытым способом. Метод оказался значительно более эффективным, чем исходный метод Лича–Гроссмана [76].

Интересную интерепретацию двойственности задач о максимальном потоке и минимальном разрезе нашел Эйкерс [54] для случая плоского графа. Задача о минимальном разрезе становится задачей о кратчайшем пути, разрезающем все переходы из источника в сток.

Эта трактовка двойственной задачи не является у Эйкерса основной темой. В его статье была высказана гипотеза, что задача о максимальном потоке в случае плоского графа может быть сведена к тривиальной последовательными преобразованиями, известными в электротехнических расчетах как

преобразования звезда–треугольник. Епифанов [18] доказал справедливость этой гипотезы, а потом, заинтересовавшись геометрической трактовкой преобразований, нашел аналогичное построение в книге известного немецкого геометра Штейница [82] *) и указал на эту связь в своей кандидатской диссертации. Память о работах Эйкерса и Епифанова жива: в Интернете обнаружилось довольно много ссылок на них. В частности, в 1989 г. появилась статья Клауса Трюмпера [83] с более простым доказательством справедливости гипотезы Эйкерса.

Классическим руководством по проблемам трудоемкости считается обзор М. Гэри и Д. Джонсона *Вычислительные машины и трудно решаемые задачи* (М.: Мир, 1982). Недавно вышла книга Ch. H. Papadimitriou, *Computational complexity* (Addison-Wesley, 1994).

Назовем еще обзор В. Е. Лихтенштейна *Модели дискретного программирования* (М.: Наука, 1971) и учебник М. М. Ковалева *Дискретная оптимизация* (Минск: Изд-во БГУ, 1977).

Небольшой обзор приближенных алгоритмов оптимизации (в терминах непрерывных задач) есть в недавно вышедшей книге С. Осовского *Нейронные сети для обработки информации* (М.: Финансы и статистика, 2002). Много ссылок на метод табу и его модификации можно найти в Интернете.

К главе 10 (Процессы)

Ссылки на литературу по теории автоматов следует начать со сборника статей *Автоматы* под редакцией К. Шеннона и Дж. Маккарти. Этот сборник, вышедший и переведенный в 1956 г., содержит упоминавшиеся нами статьи С. Клини и Э. Мура и много других классических статей. Из более поздних можно назвать книги В. М. Глушкова *Синтез цифровых автоматов* (М.: Физматгиз, 1962), Н. Е. Кобринского и Б. А. Трахтенброта [27].

Много информации можно найти в совсем недавних книгах А. А. Шалыто *SWITCH-Технология: Алгоритмизация и программирование задач логического управления* (СПб.: Наука, 1998) и *Логическое управление: методы аппаратной и программной реализации алгоритмов* (СПб.: Наука, 2000).

По цепям Маркова можно рекомендовать книгу Дж. Кемени и Дж. Снелла *Конечные цепи Маркова* (М.: Наука, 1970) и книгу В. Феллера *Введение в теорию вероятностей и ее приложения* (Т. 1. М.: Мир, 1984). С дальнейшим развитием теории марковских цепей в сторону приложений можно познакомиться в книге А. Кофмана и Р. Крюона *Массовое обслуживание* (М.: Мир, 1965).

*) Книга была подготовлена к печати через 6 лет после смерти Штейница другим известным математиком Гансом Радамахером, который даже дописал недостающие части. В 1976 г. она была переиздана без изменений, научный интерес к ней сохраняется до сих пор.

По динамическому программированию нужно вначале упомянуть *первоисточники* — книги Р. Беллмана *Динамическое программирование* (М.: Изд-во иностр. лит., 1960) и Р. Беллмана и С. Дрейфуса *Прикладные задачи динамического программирования* (М.: Наука, 1965).

Перевод замечательной книги А. Вальда *Статистические решающие функции* был опубликован в сборнике *Позиционные игры* (М.: Физматгиз, 1967). Отдельно была переведена его более простая книга *Последовательный анализ* (М.: Физматгиз, 1960).

Очень ясное описание процессов и нитей в современных операционных системах можно найти в книге Д. Соломона и М. Руссиновича *Внутреннее устройство Microsoft Windows 2000* (М.: Русская редакция, 2001).

К главе 11 (Связи дискретного и непрерывного анализа)

Первая ссылка здесь (даже *первейшая*) — это уже упоминавшаяся книга [16]. Много материала по использованию производящих функций в комбинаторике есть в уже называвшейся книге Стенли.

Очень эффектно применение производящих функций в анализе систем массового обслуживания. Оно описано в книге А. Я. Хинчина [51]. Много комбинаторных приложений производящих функций можно найти в книге С. К. Ландо *Лекции о производящих функциях* (М.: МЦНМО, 2007. 144 с.).

Библиография

[1] Адельсон-Вельский Г. М., Диниц Е. А., Карзанов А. В. *Потоковые алгоритмы.* М.: Наука, 1975. 120 с.

[2] Адельсон-Вельский Г. М., Ландис Е. М. *Один алгоритм организации информации* // Доклады АН СССР. 1962. Т. 146. С. 263–266.

[3] Андреева Е., Фалина И. *Информатика: Системы счисления и компьютерная арифметика.* М.: Лаборатория базовых знаний, 1999. 256 с.

[4] Арлазаров В. Л., Диниц Е. А., Кронрод М. А., Фараджев И. А. *Об экономном построении транзитивного замыкания ориентированного графа* // Доклады АН СССР. 1970. Т. 194. С. 487–488.

[5] Аникеич А. А., Грибов А. Б., Сурин С. С. *Сменно-суточное планирование работы грузовых автомобилей на ЭВМ.* М.: Транспорт, 1976. 152 с.

[6] Ахо А., Сети Р., Ульман Дж. *Компиляторы: Принципы, технологии, инструменты.* М.: Вильямс, 2001. 768 с.

[7] Бауэр Ф. Л., Гооз Г. *Информатика: вводный курс.* М.: Мир, 1990. Ч. 1. 336 с.; Ч. 2. 423 с.

[8] Берж К. *Теория графов и ее применения.* М.: Изд-во иностр. лит., 1962. 319 с.

[9] Бураго А. Ю., Кириллин В. А., Романовский И. В. *ФОРТ — язык для микропроцессоров.* Л.: Знание, 1989. 36 с.

[10] *Введение в криптографию* / Под общей редакцией В. В. Ященко. М.: МЦНМО: "ЧеРо", 1999. 272 с.

[11] Верещагин Н. К., Шень А. *Начала теории множеств.* М.: МЦНМО, 1999. 128 с.

[12] Воробьев Н. Н. *Числа Фибоначчи.* М.: Наука, 1992. 192 с.

[13] Гасфилд Д. *Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология* / Пер. с англ. И.В. Романовского. СПб.: Невский Диалект, 2003. 654 с.

[14] Гиндикин С. Г. *Алгебра логики в задачах.* М.: Наука, 1972. 288 с.

[15] Гнеденко Б. В., Хинчин А. Я. *Элементарное введение в теорию вероятностей.* М.: Наука, 1976. 168 с.

[16] Грэхем Р., Кнут Д., Паташник О. *Конкретная математика. Основание информатики.* М.: Мир, 1998. 708 с.

[17] Дейт К. Дж. *Введение в системы баз данных.* Киев; М.: Диалектика, 1998. 784 с.

[18] Епифанов Г. В. *Сведение планарного графа к ребру преобразованиями звезда-треугольник* // Доклады АН СССР. 1966. Т. 166. С. 13–17.

[19] Зуховицкий С. И., Радчик И. А. *Математические методы сетевого планирования*. М.: Наука, 1965. 296 с.

[20] Канторович Л. В. *Экономический расчет наилучшего использования ресурсов*. М.: Изд-во АН СССР, 1959. 344 с.

[21] Кемени Дж., Снелл Дж., Томпсон Дж. *Введение в конечную математику*. М.: Изд-во иностр. лит., 1963. 486 с.

[22] Керниган Б., Пайк Р. *Практика программирования*. СПб.: Невский Диалект, 2001. 381 с.

[23] Кнут Д. Э. *Искусство программирования*. Т. 1. Основные алгоритмы. М.: Издат. дом "Вильямс", 2000. 720 с.

[24] Кнут Д. Э. *Искусство программирования*. Т. 2. Получисленные алгоритмы. М.: Издат. дом "Вильямс", 2000. 832 с.

[25] Кнут Д. Э. *Искусство программирования*. Т. 3. Сортировка и поиск. М.: Издат. дом "Вильямс", 2000. 832 с.

[26] Кнут Д. Э. *Искусство программирования*. Т. 4. М.: Вильямс, 2007 *).
Вып. 2: *Генерация всех кортежей и перестановок*. 146 с.
Вып. 3: *Генерация всех сочетаний и разбиений*. 200 с.
Вып. 4: *Генерация всех деревьев. История комбинаторной генерации*. 156 с.

[27] Кобринский Н. Е., Трахтенброт Б. А. *Введение в теорию конечных автоматов*. М.: Физматгиз, 1963. 404 с.

[28] Кормен Т., Лейзерсон Ч., Ривест Р. *Алгоритмы: построение и анализ*. М.: МЦНМО, 1999. 960 с.

[29] Косовский Н. К. *Элементы математической логики и ее приложения к теории субрекурсивных алгоритмов*. Л.: Изд-во ЛГУ, 1981. 192 с.

[30] Лавров С. С., Гончарова Л. И. *Автоматическая обработка данных, хранение информации в памяти ЭВМ*. М.: Наука, 1971. 160 с.

[31] Левенштейн В. И. *Двоичные коды с исправлением выпадений, вставок и замещений символов* // Доклады АН СССР. 1965. Т. 163. С. 707–710.

[32] Левит Б. Ю., Лившиц В. Н. *Нелинейные сетевые транспортные задачи*. М.: Транспорт, 1972. 144 с.

[33] Липский В. *Комбинаторика для программистов*. М.: Мир, 1988. 213 с.

[34] Литл Дж., Мурти К. Г., Суини Д., Кэрел К. *Алгоритм для решения задачи о коммивояжере* // Экономика и матем. методы. 1965. Т. 1. С. 94–107.

*) См. также [70].

[35] Миркин Б. Г., Родин С. Н. *Графы и гены.* М.: Наука, 1977. 240 с.

[36] Мостеллер Ф. *Пятьдесят занимательных вероятностных задач с решениями.* М.: Физматлит, 1971. 104 с.

[37] Муртаф Б. *Современное линейное программирование: Теория и практика.* М.: Мир, 1984. 224 с.

[38] Нейман Ю. *Вводный курс теории вероятностей и математической статистики.* М.: Наука, 1968. 448 с.

[39] Писсанецки С. *Технология разреженных матриц.* М.: Мир, 1988. 410 с.

[40] Прим Р. К. *Кратчайшие связывающие сети и некоторые обобщения* // Кибернетический сборник. 1961. Вып. 2. С. 95–107.

[41] Решетняк Ю. Г. *О задаче соединения элементов вычислительной системы* // Вычислительные системы: Сб. трудов Ин-та математики СО АН СССР, Новосибирск. 1962. Вып. 3. С. 17–30.

[42] Романовский И. В. *Субоптимальные решения.* Петрозаводск: Изд-во Петрозавод. ун-та, 1998. 96 с.

[43] Романовский И. В. *Как построить цепной код* // Компьютерные инструменты в образовании. 2001. Вып. 3–4. С. 45–50.

[44] Рябко Б. Я. *Сжатие информации с помощью стопки книг* // Проблемы передачи информации. 1980. Т. 16. Вып. 4. С. 16–21.

[45] Семенюк В. В. *Экономное кодирование дискретной информации.* СПб.: Издание ГИТМО (ТУ), 2001. 115 с.

[46] Танаев В. С., Шкурба В. В. *Введение в теорию расписаний.* М.: Наука, 1975. 256 с.

[47] Тьюарсон Р. *Редкозаполненные матрицы.* М.: Мир, 1977. 190 с.

[48] Ульман Дж. *Основы систем баз данных.* М.: Финансы и статистика, 1983. 334 с.

[49] Феллер В. *Введение в теорию вероятностей и ее приложения.* Т. 1. М.: Мир, 1984. 528 с.

[50] Форд Л., Фалкерсон Д. *Потоки в сетях.* М.: Мир, 1966. 276 с.

[51] Хинчин А. Я. *Работы по математической теории массового обслуживания.* М.: Физматгиз, 1963. 236 с.

[52] Яглом А. М., Яглом И. М. *Вероятность и информация.* М.: Наука, 1973. 512 с.

[53] Aho A. V., Ullman J. D. *Foundations of Computer Science.* Computer Science Press, 1992. 765 p.

[54] Akers S. B., Jr. *The use of wye-delta transformations in network simplification* // Operations Research. 1960. Vol. 8. P. 311–323.

[55] Arya S., Mount D.M., Narayan O. *Accounting for boundary effects in nearest neighbor searching* // Discrete and Computational Geometry. 1996. Vol. 16. P. 155–176.

[56] AWK.FAQ — Часто задаваемые вопросы по языку AWK // www.faqs.org/faqs/computer-lang/awk/faq.

[57] Bentley J.L., Sleator D.D., Tarjan R.E., Wei V.K. *A locally adaptive data compression scheme* // Communications of ACM. 1986. Vol. 29. P. 330–330.

[58] Burrows M., Wheeler D.J. *A block-sorting lossless data compression algorithm* // DIGITAL systems Research Center. 1994. Res. report 124.

[59] Chen Y.L., Chin Y.H. *The quickest path problem* // Computers and Operations Research. 1990. Vol. 17. P. 153–161.

[60] Cherkassky B.V., Goldberg A.V. *On implementing the push-relabel method for the maximum flow problem* // Algorithmica. 1997. Vol. 19. P. 390–410.

[61] Cherkassky B.V., Goldberg A.V., Silverstein C., *Buckets, Heaps, Lists, and Monotone Priority Queues* // SIAM J. on Comput. 1999. Vol. 28. P. 1326–1346.

[62] Chu Y.J., Liu T.H. *On the shortest arborescence of a directed graph* // Sci. Sinica. 1965. Vol. 14. P. 1396–1400.

[63] Denardo E.V., Fox B.L., *Shortest-route methods: 1. reaching, pruning, and buckets* // Operations Research. 1979. Vol. 27. P. 161–186.

[64] Dijkstra E.W. *A note on two problems in connection with graphs* // Numerische Math. 1959. Bd. 1. S. 269–271.

[65] Fredman M.L., Tarjan R.E.. *Fibonacci heaps and their uses in improved network optimization algorithms* // J. of ACM. 1987. Vol. 34. P. 596–615.

[66] Heath F.G. *Origins of the binary code* // Scientific American. 1972. Vol. 227. P. 76–83.

[67] Hirschberg D.S. *Algorithms for the longest common subsequence problem* // J. of ACM. 1977. Vol. 24. P. 664–675.

[68] Karp R., Rabin M. *Efficient randomized pattern matching algorithms* // IBM Journal Res. Development. 1987. Vol. 31. P. 249–260.

[69] Kececioglu, J.D., Sankoff, D. *Exact and approximation algorithms for sorting by reversal* // Algorithmica. 1995. Vol. 13. P. 160–210.

[70] Knuth D.E. *The Art of Computer Programming.* Vol. 4. Fasc. 0B: A draft of Section 7.1.1: Boolean Basics. Stanford University, Addison Wesley. 2007. 88 p.

[71] Knuth D.E., Morris J.H., Pratt V.B. *Fast pattern matching in strings* // SIAM J. on Comput.. 1977. Vol. 6. P. 323–350.

[72] Kruskal J. *On the shortest spanning tree of a graph and the travelling salesman problem* // Proc. Amer. Math. Soc.. 1956. Vol. 2. P. 48–50.

[73] Kuhn H. W. *The Hungarian method for assignment problems* // Naval Res. Logist. Quart. 1956. Vol. 3. P. 253–258.

[74] Kuhn H. W. *On the origin of the Hungarian method* // History of Mathematical Programming. CWI–North-Holland, 1991. P. 77–81.

[75] Lelewer Debra A., Hirschberg D. S. *Data compression* // ACM Computing Surveys. 1987. Vol. 19. P. 261–296.

[76] Lerchs H., Grossmann I. F. *Optimum design of open-pit mines* // Transactions of the Canadian Institute of Mining. 1965. Vol. 68. P. 17–24.

[77] Menezes A., van Oorschot P., Vanstone S. *Handbook of Applied Cryptography*. CRC Press, 1996. 780 p.

[78] Rivest R. L., Shamir A., Adleman L. M. *A method for obtaining digital signatures and public key cryptosystems* // Communications of ACM. 1978. Vol. 21. P. 120–126.

[79] Rosen J. B., Su S.-Z., Xue G.-L. *Algorithms for the quickest path problem and the enumeration of quickest paths* // Computers and Operations Research. 1991. Vol. 18. P. 579–584.

[80] Schurmann K.-B., Stoye J. *An incomplex algorithm for fast suffix array construction* // Software–Practice and Experience. 2007. Vol 37. P. 309–329.

[81] Singh S. *The code book: the evolution of secrecy from Mary Queen of Scots to quantum cryptography*. Doubleday, 1999. 402 p.

[82] Steinitz E., Rademacher H. *Vorlesungen Über die Theorie der Polyeder unter Einschluss der Elemente der Topologie*. Berlin: Springer, 1934. VIII, 351 p.

[83] Truemper K. *On the delta-wye reduction for planar graphs* // J. of Graph Theory. 1989. Vol 13. P. 141–148.

[84] Vuillemin J. *A data structure for manipulating priority queues* // Communications of ACM. 1978. Vol. 21. P. 309–315.

[85] Walker A. J. *An efficient method for generating discrete random variables with general distribution* // ACM Trans. Math. Software. 1977. Vol. 3. P. 253–256.

[86] Welch T. A. *A technique for high-performance data compression* // Computer. 1984. Vol. 17. P. 8–19.

[87] Ziv J., Lempel A. *A universal algorithm for sequential data compression* // IEEE Trans. Inform. Theory. 1977. Vol. 23. P. 327–343; *Compression of individual sequences via variable-rate coding* // Ibid. 1978. Vol. 24. P. 530–536.

Алфавитный указатель

Учебное издание

РОМАНОВСКИЙ Иосиф Владимирович

Дискретный анализ

Редактор *О. М. Рощиненко*

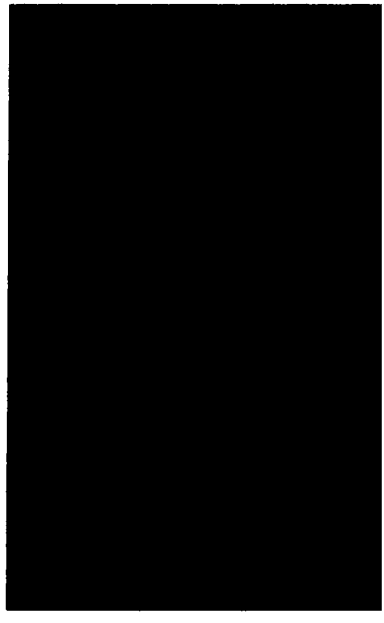
Издательство "Невский Диалект". 195220, Санкт-Петербург, Гражданский пр., 14.
Издательство "БХВ-Петербург". 198005, Санкт-Петербург, Измайловский пр., 29.

Подписано в печать 29.02.2008. Формат 60×88 1/16. Бумага газетная. Печать офсетная.
Гарнитура TimeRoman. Усл. печ. л. 20,53. Тираж 2000 экз. Заказ № 190

Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12

大 **дà** 1) большой; великий; высокий (*ростом*); крупный, огромный; вырасти; 2) старший (*по возрасту, положению*); быть старше; 3) сильно, очень, весьма; громко; особенно; преувеличенно; 4) увеличивать, расширять; преувеличивать; напускать на себя важность; 5) *сокр.* высшее учебное заведение, университет...

из китайско-русского словаря



ISBN 5-7940-0138-0