Федеральное агентство по образованию
Нижегородский государственный университет им. Н.И. Лобачевского

Национальный проект «Образование»
Инновационная образовательная программа ННГУ. Образовательно-научный центр
«Информационно-телекоммуникационные системы: физические основы и математическое обеспечение»

В.Е. Алексеев, В.А. Таланов

Алгоритмы и структуры данных

Учебно-методические материалы по программе повышения квалификации «Информационные технологии и компьютерное моделирование в прикладной математике»

Нижний Новгород
2007

Учебно-методические материалы подготовлены в рамках инновационной образовательной программы ННГУ: Образовательно-научный центр «Информационно-телекоммуникационные системы: физические основы и математическое обеспечение»

Алексеев В.Е., Таланов В.А. Алгоритмы и структуры данных. Учебно-методические материалы по программе повышения квалификации «Информационные технологии и компьютерное моделирование в прикладной математике» Нижний Новгород, 2007, 105 с.

Учебное пособие состоит из двух частей, посвященных вопросам анализа и разработки алгоритмов. В первой части рассматриваются комбинаторные алгоритмы, главным образом алгоритмы на графах. Во второй части приведены методы реализации приоритетных очередей и разделенных множеств, а также описаны некоторые нетрадиционные системы счисления.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Учебное пособие состоит из двух частей, посвященных вопросам анализа и разработки алгоритмов. В первой части рассматриваются комбинаторные алгоритмы, главным образом алгоритмы на графах. Во второй части приведены методы реализации приоритетных очередей и разделенных множеств, а также описаны некоторые нетрадиционные системы счисления.

Материал настоящего пособия включает ряд тем из курса «Анализ и разработка алгоритмов», читавшегося в течение ряда лет магистрантам факультета ВМК ННГУ и студентам по направлению «Информационные технологии».

Первая часть (главы 1-4) посвящена комбинаторным алгоритмам. В первых трех главах рассматриваются методы исчерпывающего поиска на множествах комбинаторных объектов, графах, в деревьях решений. В четвертой главе излагаются элементы теории жадных алгоритмов и описывается ряд примеров. Основной принцип отбора и организации материала состоял в том, что каждый рассматриваемый пример должен нести определенную идейную нагрузку, знакомить слушателя с одним из важных изобретений или открытий в алгоритмической области. При этом предпочтение отдавалось не самым последним или рекордным алгоритмам, а простым для понимания и убедительно демонстрирующим ту или иную идею. Предполагается знакомство слушателей с базовыми понятиями теории графов.

Во второй части (главы 5-6) рассматриваются способы структурирования информации в моделях с адресуемой памятью. Одной из основных целей при разработке структур данных является формирование математических понятий, которые пока не входят в классическую математику, но требуют формального описания и математического анализа их свойств. Основной интерес здесь представляют сложностные аспекты выполнения типичных операций. Возникновение наиболее удачных структур, использующихся в различных алгоритмах, приводит к формированию так называемых абстрактных типов данных, которые позволяют вести проектирование нетривиальных алгоритмов на более высоком уровне, не упуская из виду конкретных реализаций. Методы реализации абстрактных типов данных можно рассматривать как переход от описания алгоритма с использованием прикладных или математических понятий к описанию в конкретной системе вычислений. В пособии рассматриваются методы реализации приоритетных очередей, динамически меняющихся отношений эквивалентности, приводятся примеры

использования рассматриваемых структур в алгоритмах решения некоторых задач из теории графов.

В главе 7 приводятся некоторые сведения об избыточных системах счисления, позволяющих параллельное выполнение операций при реализации в технических устройствах.

В пособии не принят какой-либо стандартный способ для описания алгоритмов. Каждое такое описание имеет целью продемонстрировать алгоритм в целом и взаимодействие его частей, не вдаваясь в излишнюю детализацию, но и не теряя ничего существенного. Иногда даются пошаговые описания разной степени подробности, иногда тексты на «псевдоязыке», использующем элементы синтаксиса языка Pascal и математическую символику.

ГЛАВА 1. ГЕНЕРИРОВАНИЕ КОМБИНАТОРНЫХ ОБЪЕКТОВ

Во многих задачах, формулируемых на языке дискретной математики, требуется найти в некотором конечном множестве элемент, обладающий заданными свойствами,. Такая задача в принципе может быть решена перебором всех элементов этого множества, часто этот путь плох из-за того, что множество слишком велико. Но это не всегда так и бывают ситуации, когда переборный алгоритм вполне приемлем. Например, для решения задачи коммивояжера с небольшим количеством (скажем, 7-9) городов вряд ли имеет смысл разрабатывать сложный алгоритм с использованием хитроумных приемов сокращения перебора. Программа, просто перебирающая все перестановки, даст ответ достаточно быстро, а ее разработка обойдется во много раз дешевле. Но для создания такой программы нужно иметь алгоритм систематического порождения перестановок. В общем случае задача систематического генерирования состоит в том, чтобы получить один за другим все объекты данного класса без пропусков и повторений. Для решения подобных задач часто применяется следующий план.

1. На рассматриваемом множестве объектов вводится линейный порядок.
2. Разрабатывается способ построения первого в этом порядке объекта.
3. Разрабатывается алгоритм, который любой заданный объект преобразует в следующий относительно введенного упорядочения (или сообщает, что этот объект – последний).

Часто объекты из рассматриваемого множества представляют собой конечные последовательности (кортежи, слова) символов или чисел или могут быть легко преобразованы в такие последовательности. В таком случае в качестве линейного порядка на этом множестве естественно ввести лексикографический порядок. Напомним его определение.

Пусть A – множество с уже определенным на нем отношением (строгого) линейного порядка $<$ (например, A – конечный алфавит, т.е. множество букв с заданным на нем “алфавитным порядком”, или A – множество целых чисел с обычным упорядочением их по величине), A^* – множество всех кортежей, составленных из элементов множества A. Говорят, что кортеж $(x_1, x_2, \mathbf{K}, x_n)$ лексикографически предшествует кортежу $(y_1, y_2, \mathbf{K}, y_m)$, если выполняется одно из двух:

1) существует такое i, что $(x_1, \mathbf{K}\, x_{i-1}) = (y_1, \mathbf{K}, y_{i-1})$, $x_i < y_i$;

2) $n < m$, $(x_1, \mathbf{K}\, x_n) = (y_1, \mathbf{K}, y_n)$.

Отношение лексикографического предшествования будем обозначать тем же символом $<$ (или $\leq$, если возможно равенство). Это отношение является линейным порядком на множестве A^*.

Пункт 2 вышеизложенного плана решения задачи генерирования реализуется обычно легко и остается реализовать пункт 3, т.е. разработать алгоритм построения объекта, непосредственно следующего после данного в лексикографическом порядке. Ниже будут рассмотрены такие алгоритмы для подмножеств, перестановок и сочетаний.

Иногда желательно, чтобы при генерировании следующий объект в некотором смысле как можно меньше отличался от предыдущего. В этом случае говорят, что объекты генерируются с наименьшим изменением. Лексикографический порядок обычно не удовлетворяет условию наименьшего изменения. Мы рассмотрим также некоторые алгоритмы генерирования с наименьшим изменением.

Другой класс задач генерирования составляют задачи генерирования случайных комбинаторных объектов. Необходимость в получении случайных объектов может возникнуть, например, при разработке вероятностных алгоритмов или при проведении экспериментальных испытаний программ. Один из подходов к решению подобных задач состоит в нумерации объектов из рассматриваемого множества и разработке алгоритма построения объекта по заданному номеру. Если такой алгоритм имеется, то для получения случайного объекта достаточно сгенерировать случайное целое число из заданного диапазона. Мы рассмотрим методы нумерации перестановок и сочетаний. Будет также описано применение нумерации сочетаний в теории кодирования.

Генерирование подмножеств

Рассмотрим задачу генерирования всех подмножеств множества $A = \{1,2,\mathbf{K},n\}$. Удобным способом представления подмножеств в этом и многих других случаях является характеристический вектор, который для подмножества $X \subseteq A$ определяется как $h_X = (h_1,\mathbf{K},h_n)$, где

$$h_i = \begin{cases} 1, & \text{если} \quad i \in X, \\ 0, & \text{если} \quad i \notin X. \end{cases}$$

При таком представлении подмножество задается двоичным словом $h_1 \mathbf{K} h_n$, и задача сводится к генерированию таких слов. Если слово рассматривать как двоичное представление целого числа, то получение лексикографически следующего слова равносильно прибавлению единицы к этому числу. Алгоритм, строящий следующее

слово, действует следующим образом: просматриваются символы слова, начиная с последнего; все встречающиеся единицы заменяются нулями; как только встретится нуль, он заменяется единицей и работа прекращается.

Алгоритм 1. *Лексикографически следующее подмножество*

1 $i = n$

2 **while** $(i > 0)$ **and** $(h_i = 1)$ **do** $\{h_i = 0;\ i = i - 1\}$

3 **if** $i = 0$ **then** *last* = **true** **else** $\{h_i = 1;$ *last* = **false**$\}$

Переменная *last* сигнализирует о результативности процедуры: она принимает значение **true**, если следующего слова не существует, т.е. $h_i = 1$ для всех i и значение **false**, если следующее слово построено.

Для оценки трудоемкости этого алгоритма заметим, что если тело цикла в строке 2 выполняется t раз, то общее время работы алгоритма ограничено сверху величиной вида $at + b$, где a и b – некоторые константы. Величина t зависит от обрабатываемого слова, оценим среднее ее значение. Подсчитаем суммарное число повторений цикла S_n. Цикл выполняется не менее одного раза для всех слов с $h_n = 1$, число которых равно 2^{n-1}, второй раз он выполняется для слов с $h_{n-1} = h_n = 1$, их число равно 2^{n-2}, и т. д. Получаем $S_n = 2^{n-1} + 2^{n-2} + \mathbf{K} + 2^0 = 2^n - 1$. Таким образом, на одно слово в среднем приходится $S_n / 2^n < 1$ повторений. Иначе говоря, среднее время, затрачиваемое на генерирование одного подмножества, ограничено постоянной, не зависящей от n величиной.

Рассмотрим теперь генерирование подмножеств с наименьшим изменением. Это равносильно порождению двоичных слов заданной длины в порядке, при котором каждое следующее слово отличается от предыдущего ровно в одной позиции. Такая последовательность слов называется *кодом Грея*. Точнее, код Грея – это взаимно однозначное отображение g множества чисел $\{0,1,\mathbf{K},2^n - 1\}$ в множество двоичных слов длины n, при котором слова $g(i-1)$ и $g(i)$ отличаются в одной букве при любом $i = 1,\mathbf{K},2^n - 1$. Следующая теорема описывает способ построения одного из таких отображений. Символом $\oplus$ обозначаем сложение по модулю 2.

Теорема 1. *Пусть* $x_1, x_2, \mathbf{K}, x_n$ *– цифры двоичного представления числа* x*, занумерованные слева направо, (т.е.* x_1 *– это старший разряд). Тогда отображение* g*,*

определяемое равенством $g(x) = a_1 a_2 \mathbf{K}\, a_n$, *где* $a_1 = x_1$, $a_i = x_{i-1} \oplus x_i$ *при* $i = 2, \mathbf{K}, n$, *является кодом Грея.*

Доказательство. Заметим сначала, что обратное к g отображение задается равенствами $x_i = a_1 \oplus \mathbf{K} \oplus a_i$, $i = 1, \mathbf{K}, n$. Отсюда следует, что g взаимно однозначно. Покажем, что $g(x)$ и $g(x+1)$ отличаются в одной букве. Действительно, возьмем наибольшее k, при котором $x_k = 0$. Тогда $x_{k+1} = \mathbf{K} = x_n = 1$ и цифры $x'_1, x'_2, \mathbf{K}\, x'_n$ двоичного представления числа $x+1$ вычисляются следующим образом: $x'_i = x_i$ для $i < k$, $x'_i = x_i \oplus 1$ для $i > k$. Тогда $g(x+1) = a'_1, a'_2 \mathbf{K}\, a'_n$, где

$a'_i = a_i$ при $i < k$,

$a'_k = x'_{k-1} \oplus x'_k = x_{k-1} \oplus x_k \oplus 1 = a_k \oplus 1$,

$a'_i = x'_{i-1} \oplus x'_i = x_{i-1} \oplus x_i = a_i$ при $i > k$.

Таким образом, слова $a_1 a_2 \mathbf{K}\, a_n$ и $a'_1, a'_2 \mathbf{K}\, a'_n$ действительно различаются только в k-той букве. □

Приведенное доказательство дает и простой способ получения $g(x+1)$, если известны x и $g(x)$: нужно в двоичной записи числа x найти самый правый 0 и изменить букву в соответствующей позиции слова $g(x)$. Это делает следующий алгоритм, получающий на входе слово $a_1 a_2 \mathbf{K}\, a_n$ и его двоичный номер $x_1 x_2 \mathbf{K}\, x_n$ и вырабатывающий следующее слово и его двоичный номер.

Алгоритм 2. *Следующее слово кода Грея*

1 $i = n$

2 **while** $(i > 0)$ **and** $(x_i = 1)$ **do** { $x_i = 0$; $i = i + 1$ }

3 **if** $i = 0$ **then** *last* = **true** **else** { $x_i = 1$; $a_i = a_i \oplus 1$; *last* = **false** }

Как видно, этот алгоритм получен незначительной модификацией алгоритма 1, оценка трудоемкости для него та же.

Генерирование перестановок

Первой в лексикографическом упорядочении всех перестановок n элементов $\{1,2,\mathbf{K},n\}$ является, очевидно, перестановка $(1,2,\mathbf{K},n)$, а последней – перестановка $(n,\mathbf{K},2,1)$. Среди перестановок $(p_1,\mathbf{K},p_n)$ с фиксированными k первыми элементами лексикографически первой будет та, в которой $p_{k+1} < p_{k+2} < \mathbf{K} < p_n$, а

последней – та, в которой $p_{k+1} > p_{k+2} > \mathbf{K} > p_n$. Если $p = (p_1, \mathbf{K}, p_n)$ – перестановка, а $q = (q_1, \mathbf{K}, q_n)$ – лексикографически следующая за ней, причем $(p_1, \mathbf{K}, p_{k-1}) = (q_1, \mathbf{K}, q_{k-1})$, $p_k < q_k$, то p является лексикографически последней среди перестановок с фиксированными $p_1, \mathbf{K}, p_{k-1}$, q – лексикографически первой среди перестановок с фиксированными $q_1, \mathbf{K}, q_{k-1}$, а q_k – наименьший из тех элементов множества $\{p_{k+1}, \mathbf{K}, p_n\}$, которые больше p_k. Эти наблюдения приводят к алгоритму, преобразующему данную перестановку p в лексикографически следующую. Если следующей не существует, т.е. $p = (n, \mathbf{K}, 2, 1)$, то переменная *last* принимает по завершении работы алгоритма значение **true**, в противном случае – значение **false**. Через $\mathrm{T}(p_i, p_j)$ обозначается операция перестановки (транспозиция) элементов p_i и p_j.

Алгоритм 3. *Лексикографически следующая перестановка*.

1 $k = n$

2 **repeat** $k = k - 1$ **until** $(p_k < p_{k+1})$ **or** $(k = 0)$

3 **if** $k = 0$ **then** *last* = **true**

4 **else**

5 *last* = **false**

6 $j = n$

7 **while** $p_j < p_k$ **do** $j = j - 1$

8 $\mathrm{T}(p_k, p_j)$

9 **for** $i = 1$ **to** $\left\lfloor \frac{n-k}{2} \right\rfloor$ **do** $\mathrm{T}(p_{k+i}, p_{n-i+1})$

В строках 1, 2 ищется самый правый элемент p_k, правее которого имеется больший элемент. Если такого нет (цикл **repeat** прерывается при $k = 0$), то данная перестановка – последняя. В противном случае находится наименьший элемент p_j, больший p_k и стоящий правее (строки 6,7). Затем p_k и p_j меняются местами, после чего элементы $p_{k+1}, \mathbf{K}, p_n$ переставляются в обратном порядке.

Для обоснования алгоритма обозначим полученную в результате его работы перестановку через p' и предположим, что имеется перестановка q такая, что $p < q < p'$. Тогда

$$(p_1, \mathbf{K}, p_{k-1}) = (q_1, \mathbf{K}, q_{k-1}) = (p'_1, \mathbf{K}, p'_{k-1}), \quad p_k \le q_k \le p'_k = p_j.$$

Элемент p_j выбирается так, что среди $p_{k+1}, \mathbf{K}, p_n$ нет элементов, больших p_k и меньших p_j, поэтому либо $q_k = p_k$, либо $q_k = p_j$. Но в первом случае не может быть $p < q$, так как p – лексикографически последняя среди перестановок с первыми элементами $p_1, \mathbf{K}, p_k$, а во втором случае не может быть $q < p'$, так как p' – лексикографически первая среди перестановок с первыми элементами $p_1, \mathbf{K}, p_{k-1}, p_j$.

Оценим трудоемкость этого алгоритма. Если $f(p)$ – общее число операций при обработке перестановки p, а $t(p)$ – число операций транспозиции, то, как легко видеть, $f(p) \le at(p) + b$, где a и b – некоторые константы. Поэтому мы получим представление о скорости роста общего числа операций, если оценим число транспозиций. Это число зависит от перестановки, поэтому мы оценим среднее число транспозиций на одну перестановку: $t_n = \dfrac{T_n}{n!}$, где T_n – суммарное число транспозиций, совершаемых при получении всех $n!$ перестановок порядка n.

Если, начиная с $p = (1,2, \mathbf{K}, n)$, генерировать все перестановки одну за другой с помощью этого алгоритма, то вначале будут получены все $(n-1)!$ перестановок, у которых $p_1 = 1$. На это будет затрачено T_{n-1} транспозиций. Последней из этих перестановок будет $(1, n, \mathbf{K}, 2)$. При переходе от нее к следующей, $(2,1,3, \mathbf{K}, n)$, будет сделано $1 + \left\lfloor \dfrac{n-1}{2} \right\rfloor \le \dfrac{n+1}{2}$ транспозиций. Затем последует группа из $(n-1)!$ перестановок с $p_1 = 2$ и т. д. На каждую из n таких групп будет затрачено T_{n-1} транспозиций, а на каждый переход между группами – не более чем $\dfrac{n+1}{2}$ транспозиций. Поэтому получаем

$$T_n \le nT_{n-1} + \frac{n^2 - 1}{2}.$$

В качестве начального условия можно взять $T_0 = 0$.

Если X_n – решение рекуррентного уравнения

$$X_n = nX_{n-1} + \frac{n^2 - 1}{2} \qquad (1)$$

с начальным условием $X_1 = 0$, то $T_n \le X_n$. Для решения уравнения (1) можно воспользоваться методом суммирующего множителя. Этот метод применяется для решения рекуррентных уравнений вида

$$z_n = a_n z_{n-1} + b_n, \qquad (2)$$

где a_n и b_n – заданные последовательности; состоит он в следующем. Если умножить обе части уравнения (2) на число $s_n = \frac{1}{a_1 a_2 \mathbf{K}\, a_n}$, а затем сделать замену $y_n = s_n z_n$, то уравнение (2) приведется к виду

$$y_n = y_{n-1} + s_n b_n,$$

а последнее решается простым суммированием:

$$y_n = s_n b_n + s_{n-1} b_{n-1} + \mathbf{K} + s_1 b_1 + y_0.$$

Если применить это к уравнению (1), то $s_n = \frac{1}{n!}$, а $y_n = \frac{X_n}{n!}$ – та самая величина, которая нас интересует! Получаем

$$t_n \le y_n = \frac{n^2 - 1}{2n!} + \frac{(n-1)^2 - 1}{2(n-1)!} + \mathbf{K} + \frac{1^2 - 1}{2 \cdot 1!} =$$

$$= \frac{1}{2}\left(\frac{1}{(n-2)!} + \frac{1}{(n-3)!} + \mathbf{K} + \frac{1}{1!} + \frac{1}{0!} + 1 - \frac{1}{n!} \right) < \frac{1}{2}(e+1).$$

Таким образом, среднее число операций, затрачиваемых на получение одной перестановки, есть $O(1)$, т.е. ограничено сверху величиной, не зависящей от n.

Генерирование сочетаний

Сочетания из n элементов по k – это k-элементные подмножества множества $\{1, \mathbf{K}, n\}$. Их число, как известно, равно $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Поскольку сочетания –

неупорядоченные выборки, мы можем располагать их элементы в произвольном порядке. Условимся располагать их всегда в порядке возрастания. Тогда сочетание – это кортеж $(a_1, a_2, \mathbf{K}, a_k)$, элементами которого являются числа от 1 до n, причем $a_1 < a_2 < \mathbf{K} < a_k$.

Если $a_k < n$, то лексикографически следующим за сочетанием $(a_1, a_2, \mathbf{K}, a_k)$ будет $(a_1, a_2, \mathbf{K}, a_k + 1)$. Если, $a_{k-1} < n-1$, то следующим будет $(a_1, a_2, \mathbf{K}, a_{k-1} + 1, a_{k-1} + 2)$. Вообще, если i таково, что $a_j = n - k + j$ для всех $j > i$, $a_i < n - k + i$, то следующим сочетанием будет

$$(a_1, a_2, \mathbf{K}, a_{i-1}, a_i + 1, a_i + 2, \mathbf{K}, a_i + k - i + 1).$$

Если же такого i не существует, то данное сочетание – последнее. Это приводит к такому алгоритму.

Алгоритм 4. *Лексикографически следующее сочетание*

```
1   i = k + 1
2   repeat i = i − 1 until (a_i < n − k + i) or (i = 0)
3   if i = 0 then last = true
4   else
5       last = false
6       b = a_i + 1
7       for j = 0 to k − i do a_{i+j} = b + j
```

Общее число действий, совершаемых этим алгоритмом, ограничено сверху линейной функцией от числа повторений цикла в строке 2. Подсчитаем суммарное число $C_{n,k}$ повторений этого цикла при генерировании всех сочетаний из n по k. Для каждого из этих $\binom{n}{k}$ сочетаний цикл выполняется не менее одного раза. Второй раз он повторяется для тех сочетаний, у которых $a_k = n$. Таких сочетаний имеется ровно $\binom{n-1}{k-1}$. Третий раз цикл будет повторен для тех сочетаний, у которых $a_k = n$, $a_{k-1} = n-1$, их имеется ровно $\binom{n-2}{k-2}$, и т. д. Общее число повторений, таким образом, равно

$$C_{n,k} = \binom{n}{k} + \binom{n-1}{k-1} + \mathbf{K} + \binom{n-k+1}{1}.$$

Применяя тождество $\binom{a}{b} = \binom{a+1}{b} - \binom{a}{b-1}$, получим

$$C_{n,k} = \binom{n+1}{k} - \binom{n}{k-1} + \binom{n}{k-1} - \binom{n-1}{k-2} + \mathbf{K} + \binom{n-k+2}{1} - \binom{n-k+1}{0} =$$

$$= \binom{n+1}{k} - 1.$$

Для среднего числа повторений цикла на одно сочетание получаем оценку

$$c_{n,k} < \frac{\binom{n+1}{k}}{\binom{n}{k}} = \frac{n+1}{n+1-k}.$$

Нумерация перестановок и сочетаний

Формулы для вычисления лексикографических номеров перестановок и сочетаний можно получить с помощью общего способа, предложенного Ковером (M.T. Cover). Пусть A – конечный алфавит с заданным на нем линейным порядком $<$, $S \subseteq A^n$. Обозначим через $n_S(x_1, x_2, \mathbf{K}, x_i)$ число слов с префиксом $x_1 x_2 \mathbf{K} x_i$ в множестве *S*. Тогда номер слова $x = x_1 x_2 \mathbf{K} x_n$ в лексикографически упорядоченном множестве S равен (нумерация начинается с 0)

$$N_S(x) = \sum_{i=1}^{n} \sum_{a<x_i} n_S(x_1, \mathbf{K}, x_{i-1}, a). \qquad (3)$$

Применим эту формулу к перестановкам. Перестановок из n элементов с фиксированными первыми i элементами имеется ровно $(n-i)!$, если все эти элементы различны, т.е. в этом случае $n(x_1, x_2, \mathbf{K}, x_i) = (n-i)!$ Если же среди $x_1, x_2, \mathbf{K}, x_i$ имеются одинаковые, то таких перестановок не существует и $n(x_1, x_2, \mathbf{K}, x_i) = 0$. Значит, число ненулевых слагаемых во внутренней сумме будет равно числу таких *a*, которые меньше x_i и отличны от $x_1, x_2, \mathbf{K}, x_{i-1}$, а каждое из этих слагаемых равно

$(n-i)!$. Определим для перестановки $p=(p_1,\mathbf{K}\,p_n)$ и для каждого $i=1,\mathbf{K},n$ число $b_i(p)$ – количество тех среди элементов $p_{i+1},p_{i+2},\mathbf{K},p_n$, которые меньше p_i. Тогда из (3) получается формула для номера перестановки:

$$N(p)=\sum_{i=1}^{n} b_i(p)(n-i)!.$$

Отметим, что эту формулу можно трактовать как представление числа $N(p)$ в позиционной системе счисления с переменным основанием.

Для решения обратной задачи – восстановления перестановки по номеру, сначала находим числа $b_1,\mathbf{K},b_n$ с помощью следующей процедуры:

$X = N(p)$

for $i=1$ **to** n **do**

$b_i = X$ **div** $(n-i)!$

$X = X$ **mod** $(n-i)!$

Теперь $p_1=b_1+1$, а для $i=2,\mathbf{K},n$ p_i – наименьшее число, большее b_i+1 и отличное от $p_1,\mathbf{K}\,p_{i-1}$.

Применим теперь формулу (3) к нумерации сочетаний. Сейчас будет удобнее предполагать, что в сочетании $a=(a_1,a_2,\mathbf{K},a_k)$ элементы располагаются в порядке убывания: $a_1>a_2>\mathbf{K}>a_k$. Тогда внутренняя сумма в формуле (3) равна числу сочетаний из элементов $1,\mathbf{K},a_i-1$ по $k-i+1$ и для номера сочетания получаем выражение

$$N(a)=\sum_{i=1}^{k}\binom{a_i-1}{k-i+1}.$$

Если перенумеровать элементы сочетания в обратном порядке и изменить порядок суммирования на противоположный, то формула примет более изящный вид:

$$N(a)=\sum_{i=1}^{k}\binom{a_i-1}{i}. \qquad (4)$$

Восстановить сочетание по номеру можно с помощью следующей процедуры.

$X = N(a)$

for $i=1$ **to** k **do**

найти наибольшее y, при котором $\binom{y}{i} \le X$

$a_i = y+1$

$X = X - \binom{y}{i}$

Универсальное кодирование

На алгоритмах нумерации дискретных объектов, подобных описанным выше, основан ряд методов кодирования с целью сжатия информации. Это направление в теории кодирования иногда так и называют нумерационным кодированием. Рассмотрим один из методов нумерационного кодирования, основанный на нумерации сочетаний.

Пусть кодируемый текст представляет собой последовательность символов 0,1. Выберем некоторое число n и разобьем эту последовательность на отрезки длины n (блоки). Пусть блок α содержит k единиц. Код $f(\alpha)$ этого блока будет состоять из двух частей: $f(\alpha)=\beta\gamma$, где β – двоичная запись числа k длины $\lceil \log(n+1) \rceil$ (логарифмы везде по основанию 2), а γ – номер слова α среди всех слов с k единицами, представленный двоичной записью длины $\left\lceil \log \binom{n}{k} \right\rceil$. Этот номер вычисляется по формуле (4), причем под $a_1, \mathbf{K}, a_k$ нужно понимать номера позиций, на которых расположены единицы в слове α.

Это кодирование называют универсальным по следующей причине. Одна из фундаментальных теорем теории информации утверждает, что для источника сообщений, порождающего независимые двоичные символы с вероятностями p_0 и p_1,

1) для любого дешифруемого кодирования коэффициент сжатия – математическое ожидание C отношения длины закодированного сообщения к длине кодируемого не может быть меньше энтропии источника – величины $H = -p_0 \log p_0 - p_1 \log p_1$;

2) для любого $e > 0$ существует такое дешифруемое кодирование, что $C < H + e$.

Известен алгоритм построения оптимального кода (минимизирующего коэффициент сжатия). Описанный же выше способ построения кода не зависит от каких-либо вероятностей – единственным параметром является длина блока n. В то же время можно доказать, что для любого $e > 0$ существует такое n, что для любого распределения вероятностей (p_0, p_1) коэффициент сжатия этого кодирования будет меньше $H + e$.

ГЛАВА 2. ОБХОДЫ ГРАФА

Анализ сложной системы может требовать исследования всех ее элементов и связей между ними. Процесс такого исследования иногда трактуется как систематический обход системы с посещением всех узлов и рассмотрением всех связей. В чистом виде методы обхода можно наблюдать на графах. Многие задачи структурного и метрического анализа графов решаются с помощью систематического обхода. Такой обход может быть эффективно выполнен многими способами, в действительности же широкое распространение получили две стратегии –поиск в ширину и поиск в глубину. Их популярность можно объяснить не только особой идейной простотой и ясностью этих методов, но в большей степени тем, что порядок обхода, обеспечиваемый каждым из них, обладает свойствами, делающими его полезным при решении определенного круга задач. В настоящей главе рассматриваются эти методы и примеры их применения.

Работа всякого алгоритма обхода состоит в последовательном посещении вершин и исследовании ребер. Какие именно действия выполняются при посещении вершины и исследовании ребра – зависит от конкретной задачи, для решения которой производится обход. В любом случае, однако, факт посещения вершины запоминается, так что с момента посещения и до конца работы алгоритма она считается посещенной. Вершину, которая еще не посещена, будем называть *новой*. В результате посещения вершина становится *открытой* и остается такой, пока не будут исследованы все инцидентные ей ребра. После этого она превращается в *закрытую*.

Обход начинается с некоторой заранее выбранной или указанной стартовой вершиной a. Она посещается первой и становится единственной открытой вершиной. В дальнейшем каждый очередной шаг начинается с выбора некоторой открытой вершины x. Эта вершина становится *активной*. Далее исследуются ребра, инцидентные активной вершине. Если имеется ребро, соединяющее вершину x с новой вершиной y, то вершина y посещается и превращается в открытую. Если все ребра, инцидентные активной вершине, исследованы, она становится закрытой. После этого выбирается новая активная вершина, и описанные действия повторяются. Процесс заканчивается, когда множество открытых вершин становится пустым.

Поиск в ширину

Идея поиска в ширину состоит в том, чтобы посещать вершины в порядке их удаленности от стартовой вершины a. Иначе говоря, сначала посещается сама вершина a, затем все вершины, смежные с a, то есть находящиеся от нее на расстоянии 1, затем вершины, находящиеся от a на расстоянии 2, и т. д.

Основная особенность поиска в ширину, отличающая его от других способов обхода графов, состоит в том, что в качестве активной вершины выбирается та из открытых, которая была посещена раньше других. Именно этим обеспечивается главное свойство поиска в ширину: чем ближе вершина к старту, тем раньше она будет посещена. Для реализации такого правила выбора активной вершины удобно использовать для хранения множества открытых вершин очередь – когда новая вершина становится открытой, она добавляется в конец очереди, а активная выбирается в ее начале.

Опишем процедуру поиска в ширину (BFS – от английского названия этого алгоритма – Breadth First Search) из стартовой вершины a. В этом описании $V(x)$ обозначает множество всех вершин, смежных с вершиной x, Q – очередь открытых вершин. Предполагается, что при посещении вершины она помечается как посещенная и эта пометка означает, что вершина уже не является новой.

Procedure BFS(a)

1 посетить вершину a
2 $a \Rightarrow Q$
3 **while** $Q \neq \varnothing$ **do**
4 $x \Leftarrow Q$
5 **for** $y \in V(x)$ **do**
6 исследовать ребро (x, y)
7 **if** вершина y новая
8 **then** посетить вершину y
9 $y \Rightarrow Q$

Процедура BFS(s) производит обход компоненты связности, содержащей вершину s. Пусть V – множество вершин графа. Следующий алгоритм осуществляет полный обход графа методом поиска в ширину.

Алгоритм 1. *Поиск в ширину*

1 пометить все вершины как новые
2 создать пустую очередь Q
3 **for** $s \in V$ **do if** s новая **then** BFS(s)

Проанализируем трудоемкость этого алгоритма. При этом будем предполагать, что время выполнения каждой из процедур посещения вершины и исследования ребра ограничено сверху константой. Пусть обрабатываемый граф имеет n вершин и m ребер. Отметим, что после инициализации множества новых вершин N все операции, которые

над ним производятся – это операции удаления элементов. Значит, всякая вершина, став посещенной (т.е. будучи удалена из N), останется такой до конца. Цикл по всем вершинам в строке 3 алгоритма гарантирует, что все вершины будут посещены. Так как каждая посещаемая вершина помещается в очередь, а цикл в строке 3 процедуры BFS повторяется, пока очередь не опустеет, то каждая вершина когда-нибудь становится активной, причем ровно один раз. Цикл в строке 5 гарантирует, что все ребра будут исследованы. Число повторений этого цикла зависит от того, как представлен граф. Если он задан матрицей смежности, то просмотр окрестности любой вершины требует просмотра строки матрицы смежности и общее число повторений цикла будет n^2. Если же граф задан списками смежности, то для вершины x число повторений цикла равно степени этой вершины $\deg(x)$, а общее число повторений будет равно $\sum_{x \in V} \deg(x) = 2m$. Учитывая, что цикл в строке 3 алгоритма повторяется n раз, получаем общую оценку трудоемкости $O(m+n)$.

BFS-дерево и вычисление расстояний

Одна из задач, для решения которых можно применить поиск в ширину, – построение каркаса. Каркасом связного графа называется остовное дерево, т.е. дерево, состоящее из всех вершин графа и некоторых его ребер. В общем случае каркасом графа называется остовный лес, у которого области связности совпадают с областями связности графа.

Ребра, исследуемые в процессе обхода графа, можно разделить на две категории: если ребро соединяет активную вершину x с новой вершиной y, то оно классифицируется как *прямое*, в противном случае – как *обратное* (если оно ранее не было объявлено прямым). В зависимости от решаемой задачи прямые и обратные ребра могут подвергаться различной обработке.

Предположим, что алгоритм поиска в ширину применяется к связному графу. Покажем, что в этом случае по окончании обхода множество всех прямых ребер образует дерево. Действительно, допустим, что на некотором шаге работы алгоритма обнаруживается новое прямое ребро (x, y), а множество прямых ребер, накопленных к этому шагу, образует дерево F. Тогда вершина x принадлежит дереву F, а вершина y не принадлежит ему. Поэтому при добавлении к дереву F ребра (x, y) связность сохранится, а циклов не появится.

Каркас, который будет построен описанным образом в результате поиска в ширину в связном графе, называется *BFS-деревом*. Его можно рассматривать как корневое дерево с корнем в стартовой вершине a. Всякое BFS-дерево обладает свойством, на котором и основаны наиболее важные применения поиска в ширину. Каркас T связного графа G с

корнем a назовем *геодезическим деревом*, если для любой вершины x путь из x в a в дереве T является кратчайшим путем между x и a в графе G.

Теорема 1. *Любое BFS-дерево является геодезическим деревом.*

Доказательство. Обозначим через $D(i)$ множество всех вершин графа, находящихся на расстоянии i от стартовой вершины a. Работа алгоритма начинается с посещения стартовой вершины, т.е. единственной вершины, составляющей множество $D(0)$. При первом выполнении цикла **while** будут посещены и помещены в очередь все вершины из множества $D(1)$. Затем эти вершины будут одна за другой извлекаться из очереди, становиться активными, и для каждой из них будут исследоваться все смежные вершины. Те из них, которые еще не посещались, будут посещены и помещены в очередь. Но это как раз все вершины из множества $D(2)$ (когда начинается исследование окрестностей вершин из $D(1)$, ни одна вершина из $D(2)$ еще не посещалась и каждая из них смежна хотя бы с одной вершиной из $D(1)$). Следовательно, каждая вершина из $D(2)$ будет посещена после всех вершин из $D(1)$. Рассуждая далее таким образом, приходим к следующему выводу.

(А) Все вершины из $D(i+1)$ будут посещены после всех вершин из $D(i)$, $i = 0,1,\mathbf{K}$.

Строгое доказательство легко провести индукцией по i. Отметим еще следующий факт.

(Б) Если активной является вершина из $D(i)$, то в этот момент все вершины из $D(i)$ уже посещены.

В самом деле, из (А) следует, что вершины из $D(i)$ попадут в очередь после вершин из $D(i-1)$. Поэтому, когда первая вершина из $D(i)$ становится активной, все вершины из $D(i-1)$ уже закрыты. Значит, к этому моменту окрестности всех вершин из $D(i-1)$ полностью исследованы, и, следовательно, все вершины из $D(i)$ посещены.

Рассмотрим теперь момент работы алгоритма, когда активной является вершина $x \in D(i)$ и обнаруживается смежная с ней новая вершина y. В BFS-дереве расстояние между y и a на 1 больше, чем расстояние между x и a. В графе расстояние между y и a не больше, чем $i+1$, так как x и y смежны. Ввиду (А) это расстояние не может быть меньше i, а ввиду (Б) оно не может быть равно i. Значит, $y \in D(i+1)$, т.е. в графе расстояние между y и a тоже на 1 больше, чем расстояние между x и a. Следовательно, если до какого-то момента работы алгоритма расстояния от каждой из посещенных вершин до стартовой вершины в графе и в дереве были равны, то это будет верно и для

вновь посещаемой вершины. Поскольку это верно вначале, когда имеется единственная посещенная вершина a (оба расстояния равны 0), то это останется верным и тогда, когда будут посещены все вершины. □

Итак, мы можем применить поиск в ширину для вычисления расстояний от стартовой вершины a до всех остальных вершин графа – нужно только в процессе обхода для каждой посещаемой вершины y определять расстояние от y до a в BFS-дереве. Это сделать легко: $d(a,y)=d(a,x)+1$, где x – активная вершина. Вначале устанавливаем $d(a,a)=0$.

Если граф несвязен, некоторые расстояния будут бесконечными. Чтобы учесть эту возможность, положим вначале $d(a,x)=\infty$ для всех $x\neq a$. Пока вершина x остается новой, для нее сохраняется значение $d(a,x)=\infty$, когда же она посещается, $d(a,x)$ становится равным расстоянию между a и x и больше не меняется. Таким образом, бесконечность расстояния можно использовать как признак того, что вершина новая. Если по окончании работы $d(a,x)=\infty$ для некоторой вершины x, это означает, что x не достижима из a, то есть принадлежит другой компоненте связности.

Для того чтобы не только определять расстояния, но и находить кратчайшие пути от a до остальных вершин, достаточно для каждой вершины y знать ее отца $F(y)$ в BFS-дереве. Очевидно, $F(y)=x$, где x – вершина, активная в момент посещения вершины y. Заполнение таблицы F фактически означает построение BFS-дерева.

Модифицируя процедуру BFS с учетом сделанных замечаний, получаем следующий алгоритм.

Алгоритм 2. *Построение BFS-дерева и вычисление расстояний от вершины a до всех остальных вершин*

```
1   for x∈V do d(a,x)=∞
2   d(a,a)=0
3   a⇒Q
4   while Q≠∅ do
5       x⇐Q
6       for y∈V(x) do
7           if d(a,y)=∞
8               then d(a,y)=d(a,x)+1
9                    F(y)=x
```

10 $y \Rightarrow Q$

Поиск в глубину

Поиск в глубину – вероятно, наиболее важная ввиду многочисленности приложений стратегия обхода графа. Идея этого метода: идти вперед в неисследованную область, пока это возможно; если же вокруг все исследовано, отступить на шаг назад и искать новые возможности для продвижения вперед. Метод поиска в глубину известен под разными названиями, например, «бэктрекинг», «поиск с возвращением»

Понятия не посещенной, открытой, закрытой и активной вершин для поиска в глубину имеют такой же смысл, как и для поиска в ширину. Главное отличие состоит в том, что в качестве активной выбирается та из открытых вершин, которая была открыта последней. Для реализации такого правила выбора наиболее удобной структурой хранения множества открытых вершин является стек: открываемые вершины складываются в стек в том порядке, в каком они открываются, а в качестве активной выбирается последняя.

Обозначим стек для открытых вершин через S, остальные обозначения сохраняют тот же смысл, что и для поиска в ширину. Через $top(S)$ обозначается верхний элемент стека (т.е. последний элемент, добавленный к стеку). Тогда процедура обхода одной компоненты связности методом поиска в глубину со стартовой вершиной a может быть записана следующим образом (DFS – от Depth First Search).

Procedure DFS(a)

```
1     посетить вершину a
2     a ⇒ S
3     while S ≠ ∅ do
4           x = top(S)
5           if имеется неисследованное ребро (x, y)
6                 then исследовать ребро (x, y)
7                       if вершина y новая
8                             then посетить вершину y
9                                   y ⇒ S
10          else удалить x из S
```

Обратим внимание на основное отличие этой процедуры от аналогичной процедуры поиска в ширину. При поиске в ширину вершина, став активной, остается ею, пока не будет полностью исследована ее окрестность, после чего она становится закрытой. При поиске в глубину, если в окрестности активной вершины x обнаруживается новая вершина y, то y помещается в стек и при следующем повторении цикла **while** станет активной. При этом x остается в стеке и через какое-то время снова станет активной. Иначе говоря, ребра, инцидентные вершине x, будут исследованы не подряд, а с перерывами.

Алгоритм обхода всего графа – тот же, что и в случае поиска в ширину, только нужно очередь заменить стеком, а процедуру BFS – процедурой DFS.

Оценка трудоемкости $O(m+n)$ верна и для поиска в глубину, но ее доказательство требует несколько иных рассуждений, так как каждая вершина теперь может становиться активной несколько раз. Однако каждое ребро рассматривается только два раза (один раз для каждой инцидентной ему вершины), поэтому в операторе **if** в строке 5 ветвь **then** (строки 6-9) повторяется $O(m)$ раз. В этом же операторе ветвь **else** (строка 10) повторяется $O(n)$ раз, так как каждая вершина может быть удалена из стека только один раз. В целом получается $O(m+n)$, причем остаются справедливыми сделанные в предыдущей лекции замечания об условиях, при которых имеет место эта оценка.

DFS-дерево

Как и при поиске в ширину, прямые ребра при поиске в глубину образуют каркас графа. Свойства этого каркаса лежат в основе многочисленных применений метода поиска в глубину. Предположим сейчас, что исходный граф G связен. Тогда множество F всех прямых ребер, которое может быть построено алгоритмом, образует остовное дерево с корнем s, называемое DFS-деревом.

Относительно любого корневого остовного дерева все ребра графа, не принадлежащие дереву, можно разделить на две категории. Ребро назовем *продольным*, если одна из его вершин является предком другой, в противном случае ребро назовем *поперечным*.

Теорема 2. *Пусть G – связный граф, T – DFS-дерево графа G. Тогда относительно T все обратные ребра являются продольными.*

Доказательство. Убедимся сначала, что, после того, как вершина s помещена в стек, на каждом последующем шаге работы алгоритма последовательность вершин, хранящаяся в стеке, образует путь с началом в вершине s, а все ребра этого пути принадлежат дереву. Вначале это, очевидно, так. В дальнейшем всякий раз, когда новая вершина y помещается в стек, к дереву в результате предыдущей операции уже добавлено ребро xy, причем вершина x находится в стеке перед вершиной y. Значит, если указанное свойство имело место до добавления вершины в стек, то оно сохранится и после добавления. Удаление же вершины из стека, конечно, не может нарушить этого свойства.

Пусть теперь ab – обратное ребро. Каждая из вершин a и b в ходе работы алгоритма когда-либо окажется в стеке. Допустим, a окажется там раньше, чем b. Рассмотрим шаг алгоритма, на котором b помещается в стек, то есть $y=b$. В этот момент a еще находится в стеке. Действительно, вершина исключается из стека только тогда, когда в ее

окрестности нет не посещенных вершин. Но непосредственно перед помещением в стек вершина b является не посещенной и она принадлежит окрестности вершины a. Таким образом, вершина a лежит на пути, принадлежащем дереву и соединяющем вершины s и b. Но это означает, что вершина a является предком вершины b в дереве T и, следовательно, ребро ab – продольное. □

Ввиду важности этого метода рассмотрим еще два варианта алгоритма поиска в глубину. Первый из них – рекурсивный, и, как обычно, рекурсия дает возможность представить алгоритм в наиболее компактной форме. Для того, чтобы алгоритм выполнял какую-то полезную работу, будем нумеровать вершины в том порядке, в каком они встречаются при обходе. Номер, получаемый вершиной x, обозначается через $Dnum(x)$ и называется ее *глубинным номером*. Вначале полагаем $Dnum(x) = 0$ для всех x. Это нулевое значение сохраняется до тех пор, пока вершина не становится открытой, в этот момент ей присваивается ее настоящий глубинный номер. Таким образом, нет необходимости в какой-либо специальной структуре для хранения множества новых вершин – они отличаются от всех других нулевым значением $Dnum$. Переменная c хранит текущий номер.

Алгоритм 3. *Поиск в глубину с вычислением номеров – рекурсивный вариант*

1 **for** $x \in V$ **do** $Dnum(x) = 0$

2 $c = 0$

3 **for** $x \in V$ **do if** $Dnum(x) = 0$ **then** DFSa(x)

Procedure DFSa(x)

1 $c = c + 1$

2 $Dnum(x) = c$

3 **for** $x \in V(x)$ **do if** $Dnum(x) = 0$ **then** DFSa(x)

Следующий вариант алгоритма поиска в глубину отличается тем, что не использует стека для хранения открытых вершин. Стек нужен, по существу, для того, чтобы в момент, когда окрестность активной вершины исследована и необходимо сделать “шаг назад”, можно было определить вершину, в которую нужно вернуться. Но это та вершина, которая является отцом активной в DFS-дереве, если его рассматривать как исходящее дерево с корнем в стартовой вершине. Поэтому, если решение задачи предусматривает построение DFS-дерева и его удобно представить как исходящее дерево, то его можно использовать и для организации “возвратных движений” в процессе обхода. В следующем

тексте алгоритма через $F(x)$ обозначается отец вершины x в строящемся DFS-дереве, при этом для стартовой вершины s полагаем $F(s) = s$. Как и в предыдущем варианте алгоритма, здесь вычисляются глубинные номера вершин и используются для распознавания новых вершин.

Алгоритм 4. *Поиск в глубину с построением каркаса*

```
1   пометить все вершины как новые
2   for a∈V do
3   if вершина a новая then DFSb(a)
```

Procedure DFSb(a)

```
1   F(a):= a
2   открыть вершину a
3   x = a
4   while x открытая do
5   if имеется неисследованное ребро (x, y)
6         then исследовать ребро (x, y)
7               if вершина y новая
8                     then F(y) = x
9                          открыть вершину y
10                         x = y
11  else закрыть вершину x
12        x = F(x)
```

Шарниры

Вершина графа называется *шарниром* (или *точкой сочленения*), если при ее удалении увеличивается число компонент связности. Основное свойство DFS-дерева – отсутствие поперечных ребер позволяет очень просто узнать, является ли корень этого дерева вершина a шарниром. Действительно, если степень вершины a в дереве больше 1, то она – шарнир, так как вершины из разных ветвей относительно a не могут быть смежными. Если же степень a в дереве равна 1, то в дереве имеется единственная вершина b, смежная с a, и каждая из остальных вершин графа соединена с b путем, не проходящим через a, поэтому удаление вершины a не нарушает связности и a не является шарниром.

Отмеченное свойство можно было бы использовать для выявления всех шарниров, просто выполнив n раз поиск в глубину, стартуя поочередно в каждой вершине. Оказывается, все шарниры можно выявить однократным поиском в глубину. Это основано на следующем свойстве.

Теорема 3. *Пусть T – DFS-дерево графа G с корнем a. Вершина $x \neq a$ является шарниром графа тогда и только тогда, когда у нее в дереве T имеется такой сын y, что ни один потомок вершины y не соединен ребром ни с одним собственным предком вершины x.*

Здесь “собственный предок” означает предка вершины, отличного от нее самой. Доказательство этого утверждения не намного сложнее приведенного выше рассуждения относительно корня дерева и мы его опускаем.

Для применения этого критерия к поиску шарниров введем на множестве вершин функцию *Low*, связанную с DFS-деревом: значением $Low(x)$ является наименьший из глубинных номеров вершин, смежных с потомками вершины x. Если вершина y является сыном вершины x, то $Low(y) \le Dnum(x)$ (так как вершина y является потомком самой себя и смежна с вершиной x). Из теоремы 3 следует, что вершина x, отличная от a, является шарниром тогда и только тогда, когда у нее имеется сын y такой, что $Low(y) = Dnum(x)$.

Функцию Low можно определить рекурсивно – если мы знаем ее значения для всех сыновей вершины x и глубинные номера всех вершин, смежных с x и не являющихся ее сыновьями, то $Low(x)$ есть минимум из всех этих величин, то есть

$$Low(x) = \min\left(\min_{y \in A} Low(y), \quad \min_{y \in B} Dnum(y) \right),$$

где A обозначает множество всех сыновей вершины x, а B – множество всех остальных вершин, смежных с x. Нетрудно видеть, что это определение эквивалентно первоначальному. Исходя из него, можно вычислять значения функции Low и выявлять шарниры в процессе поиска в глубину с помощью следующей рекурсивной процедуры. CP обозначает множество шарниров графа, отличных от корня DFS-дерева.

Алгоритм 5. *Выявление шарниров*

1 **for** $x \in V$ **do** $Dnum(x) = 0$

2 $c = 0$

3 **for** $x \in V$ **do if** $Dnum(x) = 0$ **then** CutPoints(x)

Procedure CutPoints(x)

1 $c = c + 1$

2 $Dnum(x) = c$

3 $Low(x) = c$

4 **for** $y \in V(x)$ **do**

5 **if** $Dnum(y) = 0$

6 **then** CutPoints (y)

7 $Low(x) := \min(Low(x), Low(y))$

8 **if** $Low(y) = Dnum(x)$ **then** $CP = CP \cup \{x\}$

9 **else** $Low(x) = \min(Low(x), Dnum(y))$

База циклов

Суммой по модулю 2 (далее просто суммой) двух графов $G_1 = (V, E_1)$ и $G_2 = (V, E_2)$ называется граф $G_1 \oplus G_2 = (V, E_1 \otimes E_2)$, где $\otimes$ обозначает симметрическую разность множеств.

Граф называется *квазициклом*, если степени всех его вершин четны. Сумма по модулю 2 двух квазициклов всегда является квазициклом.

Пусть $G = (V, E)$ – некоторый граф. Множество всех его остовных квазициклов образует абелеву группу относительно сложения по модулю 2. Нулевым элементом этой группы является пустой граф $O = (V, \varnothing)$. Если определить еще операцию умножения графов на 0 и 1: $0 \cdot H = O$, $1 \cdot H = H$ для любого графа H, то это множество становится линейным векторным пространством над двухэлементным полем. Оно называется *пространством циклов* графа G. Базис пространства циклов называется *базой циклов*, а его размерность – *цикломатическим числом* графа. Цикломатическое число графа с n вершинами, m ребрами и k компонентами связности равно $m - n + k$. Иначе говоря, базу циклов графа G можно определить как такое множество остовных квазициклов $B = \{C_1, \mathbf{K}\, C_r\}$, что

1) ни один из этих квазициклов не является суммой других квазициклов из B;

2) любой остовный квазицикл графа G является суммой некоторых квазициклов из B.

Пусть T – некоторый каркас графа G. Если добавить к нему какое-либо ребро графа, не принадлежащее T, то получится подграф с единственным циклом. Циклы, которые можно получить таким способом, называют *фундаментальными циклами* (относительно данного каркаса). Каждый фундаментальный цикл состоит, таким образом, из нескольких

ребер каркаса T и одного ребра, не принадлежащего T. На самом деле, немного жертвуя точностью употребления терминов, мы будем понимать под фундаментальным циклом остовный подграф, состоящий из этого цикла и всех вершин графа, не принадлежащих циклу (они будут изолированными вершинами в этом подграфе. Ключ к построению базы циклов дает теорема, утверждающая, что множество всех фундаментальных циклов относительно любого фиксированного каркаса образует базу циклов графа. Поэтому любой алгоритм построения каркаса может быть использован для нахождения базы циклов.

Поиск в глубину особенно удобен благодаря основному свойству DFS-дерева (теорема 2) – каждое обратное ребро относительно этого дерева является продольным. Это означает, что из двух вершин такого ребра одна является предком другой в DFS-дереве. Каждое такое ребро в процессе поиска в глубину встретится дважды – один раз, когда активной вершиной будет предок, другой раз, когда ею будет потомок. В этом последнем случае искомый фундаментальный цикл состоит из рассматриваемого обратного ребра и участка пути в DFS-дереве, соединяющего эти две вершины. Но этот путь так или иначе запоминается в процессе обхода в глубину, так как он необходим для последующего возвращения. Если, например, для хранения открытых вершин используется стек, то вершины этого пути находятся в верхней части стека. В любом случае этот путь легко доступен и цикл находится без труда. Запишем процедуру построения фундаментальных циклов для одной компоненты на базе алгоритма поиска в глубину с построением DFS-дерева.

Алгоритм 6. *Построение базы циклов*

```
1    пометить все вершины как новые
2    k = 1
3    for x ∈ V do if x новая then CycleBase(x)
Procedure CycleBase(a)
1    открыть вершину a
2    F(a) = a
3    x = a
4    while x открытая do
5        if имеется неисследованное ребро (x, y)
6            then пометить ребро (x, y) как исследованное
7                if вершина y новая
```

8 **then** {открыть вершину y; $F(y)=x$; $x=y$}

9 **else** NewCycle

10 **else** закрыть вершину x

11 $x=F(x)$

Procedure NewCycle

1 $k=k+1$

2 создать список $C(k)$ из одного элемента x

3 $z=x$

4 **repeat** $z=F(z)$

5 добавить z к списку $C(k)$

6 **until** $z=y$

Хотя сам поиск в глубину выполняется за линейное время, решающее влияние на трудоемкость этого алгоритма оказывает необходимость запоминать встречающиеся циклы. Подсчитаем суммарную длину этих циклов для полного графа с n вершинами. DFS-дерево в этом случае является простым путем, относительно него будет $n-2$ цикла длины 3, $n-3$ цикла длины 4, ..., 1 цикл длины n. Сумма длин всех фундаментальных циклов будет равна

$$\sum_{i=1}^{n-2} i(n+1-i)=\frac{n^3+3n^2-8n}{6}.$$

Таким образом, на некоторых графах число операций этого алгоритма будет величиной порядка n^3.

Этот алгоритм нетрудно модифицировать так, что он будет строить базу циклов с суммарной длиной, ограниченной сверху величиной порядка n^2 (и такой же будет оценка трудоемкости алгоритма). Рассмотрим в графе произвольную вершину x и пусть $y_1, y_2, \mathbf{K}, y_k$ – все ее потомки в DFS-дереве, соединенные с x обратными ребрами. Положим также $y_{k+1}=x$. Обозначим через P_i для $i=1,\mathbf{K},k$ путь в DFS-дереве, соединяющий y_i и y_{i+1}. Описанный выше алгоритм выдает циклы вида $C_i=xP_iP_{I+1}\mathbf{K}P_k$, $i=1,\mathbf{K},k$. Рассмотрим циклы $C_i'=xP_ix$, $i=1,\mathbf{K},k$. Так как $C_i=C_i'\oplus C_{i+1}'\oplus\mathbf{K}\oplus C_k'$, то совокупность всех таких циклов также образует базу циклов графа. Назовем эту систему циклов *сокращенной*. Алгоритм легко

модифицировать так, чтобы вместо циклов C_i выдавались циклы C_i' – нужно только после обнаружения обратного ребра, ведущего от предка x к потомку y (строка 14) выписать вершины, содержащиеся в стеке, начиная с y и заканчивая следующей вершиной, смежной с x. Для эффективной проверки этой смежности целесообразно использовать матрицу смежности.

Оценим суммарную длину S циклов сокращенной системы. Предположим, что граф имеет n вершин и m ребер. Каждое обратное ребро принадлежит не более чем двум циклам сокращенной системы. Значит, общий вклад обратных ребер в S не превосходит $2m$.

Для каждого цикла из сокращенной системы назовем *верхушкой* этого цикла вершину цикла с наибольшим глубинным номером (это та вершина x, при исследовании окрестности которой был найден этот цикл). Очевидно, для каждого прямого ребра в сокращенной системе имеется не более одного цикла с данной верхушкой. Значит, число циклов, в которые входит данное прямое ребро, не превосходит числа вершин, лежащих в дереве выше этого ребра (т.е. являющихся потомками вершин этого ребра). Тем более это число не превосходит числа всех вершин графа. Так как имеется не более чем $n-1$ прямое ребро, то для суммарного вклада всех прямых ребер в S получаем верхнюю оценку n^2. Таким образом, $S < 2m + n^2 = O(n^2)$, т.е. на порядок меньше максимальной суммарной длины системы фундаментальных циклов.

ГЛАВА 3. ИСЧЕРПЫВАЮЩИЙ ПОИСК

Для многих задач дискретной математики перебор вариантов остается до сих пор единственным путем к их решению. Теория NP-полноты способствовала укоренению пессимистического взгляда на возможность существования алгоритмов полиномиальной трудоемкости для широкого круга таких задач. В то же время, как уже отмечалось, могут быть ситуации, когда перебор вполне приемлем, и тогда возникает проблема его рациональной организации. Удачно выбранная переборная стратегия может сочетаться с различными тактическими приемами сокращения перебора, что ведет к расширению множества значений входных параметров, для которых задача практически (т.е. за разумное время) решаема.

Здесь будут рассмотрены переборные стратегии для некоторых классических NP-полных задач на графах. В большинстве случаев эти стратегии можно трактовать как поиск в дереве вариантов (частичных решений или подзадач). Этот поиск может быть организован по разному, в частности, обходом дерева вариантов в глубину или в ширину. Представление множества рассматриваемых вариантов в виде дерева удобно для применения различных приемов рационализации, позволяющих отсекать (не исследовать) целые ветви в этом дереве.

Гамильтонов цикл

Гамильтоновым циклом называют простой цикл, содержащий все вершины графа. Гамильтонов цикл представляет собой, с комбинаторной точки зрения, просто перестановку вершин графа. При этом в качестве начальной вершины цикла можно выбрать любую вершину, так что можно рассматривать перестановки с фиксированным первым элементом. Самый бесхитростный план поиска гамильтонова цикла состоит в последовательном рассмотрении всех этих перестановок и проверке для каждой из них, представляет ли она цикл в данном графе. Такой способ действий уже при не очень большом числе вершин становится практически неосуществимым ввиду быстрого роста числа перестановок – имеется $(n-1)!$ перестановок из n элементов с фиксированным первым элементом.

Более рациональный подход состоит в рассмотрении всевозможных простых путей, начинающихся в произвольно выбранной стартовой вершине a до тех пор, пока не будет обнаружен гамильтонов цикл или все возможные пути не будут исследованы. По сути дела, речь тоже идет о переборе перестановок, но значительно сокращенном – если, например, вершина a несмежна с вершиной x, то все $(n-2)!$ перестановок, у которых на первом месте стоит a, а на втором x, попросту не рассматриваются.

Рассмотрим этот алгоритм подробнее. Будем считать, что граф задан списками смежности: $V(x)$ – список вершин, смежных с вершиной x. На каждом шаге алгоритма имеется уже построенный отрезок пути, он хранится в стеке $PATH$. Очередной шаг состоит в исследовании окрестности последней вершины пути. Если в ней имеется не вошедшая в путь вершина, она добавляется к пути. В противном случае последняя вершина пути исключается из стека. Когда путь содержит все вершины, графа, остается проверить, смежны ли первая и последняя вершины пути, и при утвердительном ответе выдать очередной гамильтонов цикл.

Алгоритм 1. *Поиск гамильтоновых циклов*

```
1   выбрать произвольно вершину a
2   a ⇒ PATH
3   N(a) = V(a)
4   while PATH ≠ ∅ do
5       x := top(PATH)
6       if N(x) ≠ ∅
7           then y ⇐ N(x)
8                if y не находится в PATH
9                   then y ⇒ PATH
10                       N(y) = V(y)
11                       if PATH содержит все вершины
12                          then if y смежна с a
13                                  then выдать цикл
14      else удалить вершину x из PATH
```

Этот алгоритм очень похож на алгоритм поиска в глубину и отличается от него по существу только тем, что открытая вершина, когда вся ее окрестность исследована, не закрывается, а опять становится новой (исключается из стека). В начале все вершины новые. Процесс заканчивается, когда все вершины опять станут новыми. На самом деле это и есть поиск в глубину, только не в самом графе, а в дереве путей. Вершинами этого дерева являются всевозможные простые пути, начинающиеся в вершине a, а ребро дерева соединяет два пути, один из которых получается из другого добавлением одной вершины в конце. На рисунке 1 показаны граф и его дерево путей из вершины 1.

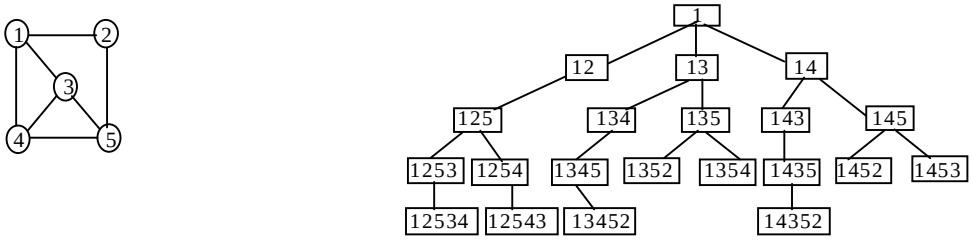

Рис. 1

В худшем случае время работы этого алгоритма тоже растет с факториальной скоростью. Например, для графа $K_{n-1} + K_1$ (граф с двумя компонентами связности, одна из которых – полный граф с $n-1$ вершиной, другая – изолированная вершина), если в качестве стартовой выбрана не изолированная вершина, то будут рассмотрены все $(n-2)!$ простых путей длины $n-2$ в большой компоненте. Вместе с тем, если перед поиском гамильтонова цикла исходный граф проверить на связность, то ответ будет получен быстро. Можно пойти дальше и при обходе дерева путей поверять на связность каждый встречающийся “остаточный граф”, т.е. граф, получающийся из исходного удалением всех вершин рассматриваемого пути. Если этот граф несвязен, то этот путь не может быть продолжен до гамильтонова цикла. Поэтому можно не исследовать соответствующую ветвь дерева, а вернуться к рассмотрению более короткого пути, удалив последнюю вершину (т.е. сделать “шаг назад” в поиске в глубину). Можно пойти еще дальше и заметить, что если некоторая вершина x DFS-дерева с корнем s является развилкой, т.е. имеет не менее двух сыновей, то в подграфе исходного графа, полученном удалением всех предков этой вершины, кроме нее самой, она будет шарниром. Поэтому путь от s до x в DFS-дереве не может быть продолжен до гамильтонова цикла. Эти соображения приводят к такой модификации алгоритма: обходим граф поиском в глубину с построением DFS-дерева, затем находим в этом дереве самую нижнюю развилку (развилку с наименьшим глубинным номером). Если ни одной развилки нет, то само DFS-дерево представляет собой гамильтонов путь и остается проверить наличие ребра, соединяющего начало и конец пути. Если же x – развилка, то возвращаемся из x в предшествующую вершину пути, помечаем все вершины, кроме собственных предков вершины x, как не посещенные и возобновляем поиск в глубину с этого места.

Изоморфное вложение

Рассмотрим обобщение задачи об изоморфизме графов – задачу об изоморфном вложении. Она состоит в том, чтобы для двух данных графов G_1 и G_2 определить,

существует ли во втором графе порожденный подграф, изоморфный первому. Известно, что эта задача NP-полна, в то время как для задачи об изоморфизме до настоящего времени ни доказательства NP-полноты, ни полиномиальных алгоритмов не найдено.

Простейший план решения задачи об изоморфном вложении состоит в переборе всех инъективных отображений множества вершин первого графа в множество второго (т.е. отображений, при которых образы различных вершин различны). Если в первом графе n_1 вершин, а во втором n_2, то существует $n_2(n_2-1)\mathbf{K}(n_2-n_1+1)$ таких отображений. Как и для задачи о гамильтоновом цикле этот план нетрудно усовершенствовать, если рассматривать частичные решения и анализировать возможности их продолжения.

Будем считать вершинами обоих графов натуральные числа: $VG_1=\{1,2,\mathbf{K},n_1\}$, $VG_2=\{1,2,\mathbf{K},n_2\}$. В качестве частичных решений будем рассматривать частичные отображения $f:\{1,\mathbf{K},p\}\to VG_2$, являющиеся изоморфными вложениями подграфа графа G_1, порожденного множеством вершин $\{1,\mathbf{K},p\}$, в граф G_2, $p<n_1$. Имея такое частичное решение, можно попытаться его продолжить, найдя подходящий образ для вершины $p+1$. Для этого пытаемся отобразить вершину $p+1$ последовательно в каждую свободную (т.е. не являющуюся образом какой-либо из вершин $1,\mathbf{K},p$) вершину графа G_2. Отображение может быть продолжено, если существует такая свободная вершина a, что равенство $A_1(k,p+1)=A_2(f(k),a)$ выполняется для всех $k=1,\mathbf{K},p$, где A_1 и A_2 – матрицы смежности графов G_1 и G_2. Если такая вершина a найдена, полагаем $f(p+1)=a$ и пытаемся продолжить это новое частичное решение. Если же такой вершины a не существует, отступаем на шаг назад и пытаемся подобрать другой подходящий образ для вершины p. В общем, это тоже поиск в глубину в дереве частичных решений, вершинами которого являются частичные отображения, а ребра связывают каждое отображения с его возможными одношаговыми продолжениями. Листьям дерева соответствуют изоморфные вложения G_1 в G_2.

Пусть $Emb(f,p)$ – функция со значением **true**, если отображение f, определенное на множестве $\{1,\mathbf{K},p\}$, является изоморфным вложением подграфа графа G_1, порожденного множеством вершин $\{1,\mathbf{K},p\}$, в граф G_2, и **false** в противном случае. Обозначим через N множество свободных вершин. Тогда намеченный выше алгоритм можно представить следующим образом.

Алгоритм 2. *Изоморфное вложение графов*

1 $p = 1$

2 $f(1) = 0$

3 $N = \{1, \mathbf{K}, n_2\}$

4 **while** $0 < p \le n_1$ **do**

5 **repeat** $f(p) = p + 1$

6 **until** $f(p) > n_2$ **or** ($f(p) \in N$ **and** $Emb(p)$)

7 **if** $f(p) \le n_2$

8 **then** $\{ N = N - \{f(p)\};\ p = p + 1;\ f(p) = 0 \}$

9 **else** $\{ p = p - 1;\ N = N \cup \{f(p)\} \}$

10 **if** $p > n_1$ **then** $IsEmb = 1$ **else** $IsEmb = 0$

Переменная *IsEmb* сигнализирует о том, найдено ли изоморфное вложение (значение 1), или его не существует (значение 0).

Как и в случае с гамильтоновыми циклами, имеются возможности для рационализации в рамках этой общей схемы. Одна из них состоит в том, чтобы “заглядывать вперед”: когда рассматривается частичное отображение f на множестве $\{1, \mathbf{K}, p\}$, можно исследовать возможность его продолжения не только на вершину $p+1$, но и по отдельности на каждую из остальных оставшихся вершин графа G_1, т.е. для каждой вершины $x \in \{p+1, \mathbf{K}, n_1\}$ выяснить, существует ли такая свободная вершина y, что отображение f, дополненное значением $f(x) = y$, будет изоморфным вложением подграфа графа G_1, порожденного множеством $\{1, \mathbf{K}, p\} \cup \{x\}$, в граф G_2. Если хотя бы для одной вершины x ответ отрицателен, то f не может быть продолжено до изоморфного вложения G_1 в G_2.

Независимые множества, клики, вершинные покрытия

Независимым множеством вершин графа называется любое множество попарно не смежных вершин, т.е. множество вершин, порождающее пустой подграф. Независимое множество называется *максимальным*, если оно не является собственным подмножеством другого независимого множества, и *наибольшим*, если содержит наибольшее количество вершин. Число вершин в наибольшем независимом множестве графа G обозначается через $a(G)$ и называется *числом независимости* графа. Задача о независимом множестве состоит в нахождении наибольшего независимого множества .

Кликой графа называется множество вершин, порождающее полный подграф, т.е. множество вершин, каждые две из которых смежны. Число вершин в клике наибольшего размера называется *кликовым числом* графа и обозначается через $k(G)$. Очевидно, задача о независимом множестве трансформируется в задачу о клике и наоборот простым переходом от данного графа G к дополнительному графу $\overline{G}$, так что $a(G) = k(\overline{G})$.

Вершинное покрытие графа – это такое множество вершин, что каждое ребро графа инцидентно хотя бы одной из этих вершин. Наименьшее число вершин в вершинном покрытии графа G обозначается через $b(G)$ и называется *числом вершинного покрытия* графа.

Между задачами о независимом множестве и о вершинном покрытии тоже имеется простая связь благодаря следующему легко устанавливаемому факту.

Теорема 1. *Подмножество U множества вершин графа G является вершинным покрытием тогда и только тогда, когда $VG - U$ – независимое множество.*

В частности, отсюда следует, что $a(G) + b(G) = n$ для любого графа G с n вершинами.

Таким образом, все три задачи тесно связаны друг с другом. Далее рассмотрим задачу о независимом множестве.

Пусть G – граф, в котором требуется найти наибольшее независимое множество. Выберем в нем произвольную вершину a и пусть A – множество всех вершин графа, смежных с a, B – множество всех вершин, не смежных с a. Обозначим через G_1 подграф, получающийся удалением из графа G вершины a, а через G_2 подграф, получающийся удалением из G всех вершин множества $A \cup \{a\}$. Иначе говоря, G_1 – подграф графа G, порожденный множеством $A \cup B$, а G_2 – подграф, порожденный множеством B.

Пусть X – какое-нибудь независимое множество графа G. Если оно не содержит вершину a, то оно является независимым множеством графа G_1, если же $a \in X$, то множество $X - \{a\}$ является независимым множеством графа G_2. Таким образом, задача о независимом множестве для графа G свелась к решению той же задачи для двух графов меньшего размера. Это приводит к рекуррентному уравнению для числа независимости:

$$a(G) = \max\{a(G_1), a(G_2) + 1\}$$

и к рекурсивному алгоритму для нахождения наибольшего независимого множества графа G: найдем наибольшее независимое множество X_1 графа G_1, затем наибольшее независимое множество X_2 графа G_2 и выберем большее из множеств X_1 и $X_2 \cup \{a\}$. В целом процесс решения задачи можно при этом интерпретировать как исчерпывающий поиск в возникающем дереве подзадач, т.е. подграфов. Каждой вершине этого дерева соответствует некоторый граф, сыновьям вершины – два подграфа этого графа, корню – исходный граф. Листьям дерева соответствуют графы, для которых задача решается непосредственно, без сведения к подзадачам. В простейшем случае это могут быть графы с пустым множеством вершин, но при этом количество листьев будет максимальным. Можно доводить разложение на подзадачи до графов с пустым множеством ребер, при этом листьев в дереве станет меньше, но потребуется для каждого встречающегося графа проверять наличие в нем ребер. Число листьев зависит от того, в каком порядке рассматриваются вершины графа, но оно во всяком случае не меньше, чем число максимальных независимых множеств у графа, так как каждое из этих множеств будет соответствовать некоторому листу. Так, для графа pK_2, т.е. графа, состоящего из p компонент связности, каждая из которых состоит из двух вершин, в дереве подзадач будет 2^p листьев.

Известны различные приемы сокращения перебора при использовании описанной стратегии исчерпывающего поиска. Один из них основан на следующем наблюдении. Допустим, в графе G, для которого нужно найти наибольшее независимое множество, имеются две смежные вершины a и b такие, что $V(a) - \{b\} \subseteq V(b) - \{a\}$. Тогда в этом графе существует наибольшее независимое множество, не содержащее вершину b. Это следует из того, что для любого независимого множества X, содержащего b, множество $(X - \{b\}) \cup \{a\}$ тоже будет независимым. Значит, если мы удалим из графа вершину b, то получим граф с тем же числом независимости. Этот прием называется “сжатием по включению”. Исследование применимости и применение операции сжатия по включению к каждому встречающемуся подграфу требует, конечно, дополнительных расходов времени ($O(n^3)$ на граф с n вершинами), но может привести к существенному сокращению дерева подзадач. Для некоторых графов дерево подзадач при применении этого приема вырождается в путь, т.е. задача о независимом множестве решается с помощью одних только сжатий по включению. Таков, например, граф pK_2, и вообще любой лес. Можно доказать, что это верно и для каждого из так называемых хордальных графов. Граф называется *хордальным* (или *триангулированным*), если в нем

нет порожденных простых циклов длины ≥ 4. Иначе говоря, в хордальном графе для каждого простого цикла длины 4 или больше имеется хотя бы одна хорда – ребро, соединяющее две вершины, принадлежащие циклу, но не являющимися в нем соседними.

Раскраски

Раскраской графа называется назначение цветов его вершинам. Обычно цвета – это числа $1,2,...,k$ и раскраску можно трактовать как разбиение множества вершин $V = V_1 \cup V_2 \cup \mathbf{K} \cup V_k$, где V_i – множество вершин цвета i. Раскраска называется *правильной*, если каждое V_i является независимым множеством. Иначе говоря, в правильной раскраске любые две смежные вершины должны иметь разные цвета. Задача о раскраске состоит в нахождении правильной раскраски данного графа G в наименьшее число цветов. Это число называется *хроматическим числом* графа и обозначается через $c(G)$.

Рассмотрим алгоритм решения задачи о раскраске, похожий на описанный выше алгоритм для задачи о независимом множестве в том смысле, что задача для данного графа сводится к той же задаче для двух других графов. Разница состоит в том, что теперь эти новые графы не будут подграфами исходного. Тем не менее снова возникает дерево вариантов, обход которого позволяет найти решение.

Выберем в данном графе G две несмежные вершины a и b и построим два новых графа: G_1, получающийся добавлением ребра (a,b) к графу G, и G_2, получающийся из G слиянием вершин a и b. Операция слияния состоит в удалении вершин a и b и добавлении новой вершины c и ребер, соединяющих ее с каждой вершиной, с которой была смежна хотя бы одна из вершин a, b. Если в правильной раскраске графа G вершины a и b имеют разные цвета, то она будет правильной и для графа G_1. Если же цвета вершин a и b в раскраске графа G одинаковы, то граф G_2 можно раскрасить в тоже число цветов: новая вершина c окрашивается в тот цвет, в который окрашены вершины a и b, а все остальные вершины сохраняют те цвета, которые они имели в графе G. Обратно, раскраска каждого из графов G_1, G_2, очевидно, дает раскраску графа G в то же число цветов. Поэтому

$$c(G) = \min(c(G_1), c(G_2)),$$

что дает возможность рекурсивного нахождения раскраски графа в минимальное число цветов. Заметим, что граф G_1 имеет столько же вершин, сколько исходный граф, но у

него больше ребер. Поэтому рекурсия в конечном счете приводит к полным графам, для которых задача о раскраске решается тривиально.

В описанную схему решения задачи о раскраске можно включить тот же прием сжатия по включению, что и для задачи о независимом множестве. Небольшое отличие состоит в том, что теперь вершины a и b должны быть несмежны. Итак, пусть в графе G имеются две несмежные вершины a и b такие, что $V(a) \subseteq V(b)$. Тогда хроматическое число этого графа равно хроматическому числу графа, полученного удалением вершины a, так как эта вершина в любом случае можно окрасить в тот же цвет, что и вершину b. Для графов, дополнительных к хордальным, раскраска в минимальное число цветов может быть найдена с помощью одних только сжатий по включению.

ГЛАВА 4. ЖАДНЫЕ АЛГОРИТМЫ

Термин «жадный алгоритм» употребляется по отношению к довольно разнообразным алгоритмам решения дискретных оптимизационных задач. Иногда для теоретических целей ему придают какое-либо точное значение, но в общем он имеет не очень четко очерченную область применения. Так называют алгоритмы, в которых на каждом шаге делается выбор, дающий наибольший эффект. Это не обязательно наибольшее приращение целевой функции, эффект может заключаться в чем-то другом, по видимости полезном для поиска наилучшего итогового решения. Иногда жадные алгоритмы называют градиентными, имея в виду идейную близость к известному методу решения непрерывных задач.

Распространенность жадных алгоритмов связана в первую очередь с их быстродействием. В большинстве случаев жадные алгоритмы не гарантируют оптимальности решения задачи в целом, т.е. являются эвристическими или приближенными. Но есть задачи, для которых решение, полученное жадным алгоритмом, всегда оптимально. Такие задачи давно привлекли к себе внимание исследователей. В настоящее время можно говорить о существовании теории жадных алгоритмов, одна из главных проблем которой – очертить круг задач, решаемых жадными алгоритмами.

В этой главе будут рассмотрены классические примеры жадных алгоритмов для задачи об оптимальном каркасе и дан набросок общей теории жадных алгоритмов. Затем будет рассмотрено применение жадного алгоритма в сочетании с методом увеличивающих цепей для решения задачи о паросочетании наибольшего веса в двудольном графе со взвешенными вершинами.

Оптимальные каркасы

Задача об оптимальном каркасе (стягивающем дереве) состоит в следующем. Дан обыкновенный граф $G = (V, E)$ и весовая функция на множестве ребер $w: V \to \mathbf{R}$. Вес множества $X \subseteq E$ определяется как сумма весов составляющих его ребер. Требуется в графе G найти каркас максимального веса. Обычно рассматривают задачу на минимум, но это не существенно – она преобразуется в задачу на максимум, если заменить функцию w на $-w$. В этом разделе будем предполагать, что граф G связен, так что решением задачи всегда будет дерево. Для решения задачи об оптимальном каркасе известно несколько жадных алгоритмов. Рассмотрим два из них.

В *алгоритме Прима* на каждом шаге рассматривается частичное решение задачи, представляющее дерево. Вначале это дерево состоит из единственной вершины, в

качестве этой вершины может быть выбрана любая вершина графа. Затем к дереву последовательно добавляются ребра и вершины, пока не получится остовное дерево, т.е. каркас. Для того, чтобы из текущего дерева при добавлении нового ребра опять получилось дерево, это новое ребро должно соединять вершину дерева с вершиной, еще не принадлежащей дереву. Такие ребра будем называть *подходящими* относительно рассматриваемого дерева. “Жадность” алгоритма состоит в том, что на каждом шаге из всех подходящих ребер выбирается ребро наименьшего веса. Это ребро вместе с одно новой вершиной добавляется к дереву. Если обозначить через U и F множества вершин и ребер строящегося дерева, а через W множество вершин, еще не вошедших в это дерево, то алгоритм Прима можно представить следующим образом.

Алгоритм 1. *Построение оптимального каркаса методом Прима*

1 $U = \{a\}$, где a – произвольная вершина графа

2 $F = \varnothing$

3 $W = V - \{a\}$

4 while $W \neq \varnothing$ **do**

5 найти ребро наибольшего веса $e = (x, y)$ среди всех таких ребер, у которых $x \in U$, $y \in W$

6 $F = F \cup \{e\}$

7 $U = U \cup \{y\}$

8 $W = W - \{y\}$

Следующая теорема дает обоснование алгоритма Прима. Дерево F назовем *фрагментом,* если оно является подграфом графа G и существует такой оптимальный каркас T_0 графа G, что F является подграфом дерева T_0.

Теорема 1. *Если F – фрагмент, e – подходящее ребро наибольшего веса относительно F, то $F \cup \{e\}$ – фрагмент.*

Доказательство. Пусть T_0 – оптимальный каркас, содержащий F в качестве подграфа. Если ребро e принадлежит T_0, то $F \cup \{e\}$ – подграф дерева T_0 и, следовательно, фрагмент. Допустим, e не принадлежит T_0. Если добавить ребро e к дереву T_0, то образуется цикл. В этом цикле есть еще хотя бы одно подходящее ребро относительно F (никакой цикл, очевидно, не может содержать единственное подходящее ребро). Пусть e' – такое ребро. Тогда подграф $T_0' = T_0 - e' + e$, получающийся из T_0

удалением ребра e' и добавлением ребра e, тоже будет деревом. Так как $w(e') \le w(e)$, то $w(T_0') \ge w(T_0)$. Но T_0 – оптимальный каркас, следовательно, $w(T_0') = w(T_0)$ и T_0' – тоже оптимальный каркас. Но $F \cup \{e\}$ является подграфом графа T_0' и, следовательно, фрагментом. □

Дерево, состоящее из единственной вершины, очевидно, является фрагментом. Из теоремы 1 следует, что если после некоторого количества шагов алгоритма Прима дерево F является фрагментом, то оно будет фрагментом и после следующего шага. Следовательно, и окончательное решение, полученное алгоритмом, будет фрагментом, т.е. оптимальным каркасом.

Другой жадный алгоритм для задачи об оптимальном каркасе известен как *алгоритм Крускала*. В нем тоже на каждом шаге рассматривается частичное решение; отличие от алгоритма Прима состоит в том, что в алгоритме Крускала частичное решение всегда представляет собой остовный лес F графа G, т.е. лес, состоящий из всех вершин графа G и некоторых его ребер. Вначале F не содержит ни одного ребра, т.е. состоит из изолированных вершин. Затем к нему последовательно добавляются ребра, пока не будет построен каркас графа G. Пусть F – лес, построенный к очередному шагу. Ребро графа, не принадлежащее F, назовем красным, если вершины этого ребра принадлежат одной компоненте связности леса F, и зеленым, если они принадлежат разным компонентам. Если к F добавить красное ребро, то образуется цикл. Если же к F добавить зеленое ребро, то получится новый лес, в котором будет на одну компоненту связности меньше, чем в F, так как в результате добавления ребра две компоненты сольются в одну. Таким образом, к F нельзя добавить никакое красное ребро и можно добавить любое зеленое. Для выбора добавляемого ребра применяется тот же жадный принцип, что и в алгоритме Прима – из всех зеленых ребер выбирается ребро наибольшего веса. Для того, чтобы облегчить поиск этого ребра, вначале все ребра графа упорядочиваются по убыванию весов: $w(e_1) \ge w(e_2) \ge \mathbf{K} \ge w(e_m)$. Теперь последовательность ребер $e_1, e_2, \mathbf{K}, e_m$ достаточно просмотреть один раз и для очередного рассматриваемого ребра нужно только уметь определять, является ли оно красным или зеленым относительно построенного к этому моменту леса F. Красные ребра просто пропускаются, а зеленые добавляются к F.

Для более формального описания алгоритма заметим, что текущий лес F определяет разбиение множества вершин графа на области связности этого леса: $V = P_1 \cup P_2 \cup \mathbf{K} \cup P_k$, и что красное ребро – это такое, у которого обе вершины принадлежат одной части разбиения. Пусть $Part(x)$ – функция, возвращающая для

каждой вершины x имя той части разбиения, которой принадлежит x, а $Unite(x, y)$ – процедура, которая по именам x и y двух частей разбиения строит новое разбиение, заменяя эти две части их объединением. Пусть $e_i = (a_i, b_i)$, $i = 1, \mathbf{K}\, m$. Тогда алгоритм Крускала (после упомянутого упорядочения ребер) можно записать следующим образом.

Алгоритм 2. *Построение оптимального каркаса методом Крускала*

1 **for** $i = 1$ **to** m **do**

2 $x = Part(a_i)$

3 $y = Part(b_i)$

4 **if** $x \neq y$ **then** $\{F = F \cup \{e_i\},\ Unite(x, y)\}$

Подробности реализации и оценки трудоемкости алгоритма Крускала будут рассмотрены в главе 6. Корректность этого алгоритма доказывается почти точно так же, как для алгоритма Прима. Впрочем, она является следствием более общей теоремы Радо-Эдмондса, которая будет рассмотрена далее.

Матроиды

Когда возник вопрос о том, каковы обстоятельства, при которых жадный алгоритм приводит к успеху, или иначе – что особенного в тех задачах, для которых он дает точное решение, – оказалось, что математическая теория, с помощью что-то в этом можно прояснить, уже существует. Это теория матроидов, основы которой были заложены Уитни в работе 1935 г. Целью создания теории матроидов было изучение комбинаторного аспекта линейной независимости, в дальнейшем обнаружились разнообразные применения понятия матроида, первоначально совсем не имевшиеся в виду.

Определение. *Матроидом* называется пара $M = (E, \Phi)$, где E – конечное непустое множество, Φ – семейство подмножеств множества E, удовлетворяющее условиям:

(1) если $X \in \Phi$ и $Y \subseteq X$, то $Y \in \Phi$;

(2) если $X \in \Phi$, $Y \in \Phi$ и $|X| < |Y|$, то существует такой элемент $a \in Y - X$, что $X \cup (a) \in \Phi$.

Элементы множества E называются *элементами матроида*, а множества из семейства Φ – *независимыми множествами* матроида. Максимальное по включению независимое множество называют *базой* матроида. Из аксиомы (2) следует, что все базы матроида состоят из одинакового количества элементов.

Если E – множество строк некоторой матрицы, а Φ состоит из всех линейно независимых множеств строк этой матрицы, то пара (E,Φ) образует матроид, называемый *матричным матроидом*.

Другим важным типом матроидов являются *графовые матроиды*. Пусть $G=(V,E)$ – обыкновенный граф. Подмножество множества E назовем *ациклическим*, если подграф, образованный ребрами этого подмножества, не содержит циклов, т.е. является лесом.

Теорема 2. *Если* $G=(V,E)$ *– обыкновенный граф,* Φ *– семейство всех ациклических подмножеств множества* E*, то пара* $M_G=(E,\Phi)$ *является матроидом.*

Доказательство. Аксиома (1) выполняется, так как всякое подмножество ациклического множества, очевидно, является ациклическим. Докажем, что выполняется и (2). Пусть X и Y – два ациклических множества и $|X|<|Y|$. Допустим, что не существует такого ребра $e\in Y$, что множество $X\cup\{e\}$ является ациклическим. Тогда при добавлении любого ребра из множества Y к множеству X образуется цикл. Это означает, что концы каждого такого ребра принадлежат одной компоненте связности остовного подграфа, образованного ребрами множества Y. Тогда каждая область связности подграфа X содержится в какой-нибудь области связности подграфа Y. Но компоненты связности каждого из этих подграфов – деревья, а дерево с k вершинами содержит ровно $k-1$ ребер, следовательно, в этом случае число ребер в Y не превосходило бы числа ребер в X, что противоречит условию $|X|<|Y|$. □

Базами графового матроида являются все каркасы графа. Для связного графа это будут все его остовные деревья.

Теорема Радо-Эдмондса

Рассмотрим общий тип оптимизационных задач, формулируемых следующим образом. Дано произвольное конечное множество E и некоторое семейство Φ его подмножеств. Для каждого элемента $x\in E$ задан его вес – положительное число $w(x)$. Вес множества $X\subseteq E$ определяется как сумма весов его элементов. Требуется найти множество наибольшего веса, принадлежащее Φ. В эту схему укладываются многие известные задачи, например, задача о независимом множестве графа (E – множество вершин графа, вес каждой вершины равен 1, а Φ состоит из всех независимых множеств) или задача об оптимальном каркасе (E – множество ребер графа, Φ состоит из всех ациклических множеств).

Сформулируем теперь жадный алгоритм для решения этой общей задачи. Чтобы отличать этот алгоритм от других жадных алгоритмов, назовем его СПО (Сортировка и Последовательный Отбор).

Алгоритм 3. *Алгоритм СПО*

1 Упорядочить элементы множества E по убыванию весов:
$E = \{e_1, e_{2,}\mathbf{K}, e_m\}$, $w(e_1) \geq w(e_2) \geq \mathbf{K} \geq w(e_m)$.

2 $A = \varnothing$

3 **for** $i = 1$ **to** m **do if** $A \cup \{e_i\} \in \Phi$ **then** $A = A \cup \{e_i\}$

Алгоритм Крускала является алгоритмом этого типа и он всегда дает оптимальное решение задачи. Уместен вопрос: каким условиям должно удовлетворять семейство Φ для того, чтобы при любой весовой функции w алгоритм СПО находил оптимальное решение? Исчерпывающий ответ дает следующая теорема Радо-Эдмондса.

Теорема 3. *Если* $M = (E, \Phi)$ *– матроид, то для любой весовой функции* $w: \to \mathbf{R}^+$ *множество A, найденное алгоритмом СПО, будет множеством наибольшего веса из* Φ. *Если же* $M = (E, \Phi)$ *не является матроидом, то найдется такая функция* $w: \to \mathbf{R}^+$, *что A не будет множеством наибольшего веса из* Φ.

Доказательство. Предположим, что пара $M = (E, \Phi)$ является матроидом и алгоритм СПО строит множество $A = \{a_1, a_2, \mathbf{K}, a_n\}$ причем $w(a_1) \geq w(a_2) \geq \mathbf{K} \geq w(a_n)$. Очевидно, A является базой матроида. Пусть $B = \{b_1, b_2, \mathbf{K}, b_k\}$ – любое другое независимое множество и $w(b_1) \geq w(b_2) \geq \mathbf{K} \geq w(b_k)$. Так как A – база, то $k \leq n$. Покажем, что $w(a_i) \geq w(b_i)$ для $i \in \{1, \mathbf{K}, k\}$. Действительно, положим $X = \{a_1, \mathbf{K}, a_{i-1}\}$, $Y = \{b_1, \mathbf{K}, b_{i-1}, b_i\}$ для некоторого i. Согласно условию (2) определения матроида, в множестве Y имеется такой элемент b_j, что $b_j \notin X$ и множество $X \cup \{b_j\}$ – независимое. В соответствии с алгоритмом, элементом наибольшего веса, который может быть добавлен к X так, чтобы получилось независимое множество, является a_i. Следовательно, $w(a_i) \geq w(b_j) \geq w(b_i)$.

Теперь предположим, что $M = (E, \Phi)$ не является матроидом. Допустим сначала, что нарушается условие (1), т.е. существуют такие подмножества X и Y множества E, что $X \in \Phi$, $Y \subset X$ и $Y \notin \Phi$. Определим функцию w следующим образом:

$$w(x)=\begin{cases}1, & \text{если} \quad x\in Y,\\ 0, & \text{если} \quad x\notin Y.\end{cases}$$

Алгоритм СПО сначала будет рассматривать все элементы множества Y. Так как $Y\notin\Phi$, то не все они войдут в построенное алгоритмом множество A. Следовательно, $w(A)<|Y|$. В то же время имеется множество $X\in\Phi$ такое, что $w(X)=|Y|$. Таким образом, в этом случае алгоритм СПО строит не оптимальное множество. Если же условие (1) выполнено, а не выполняется условие (2), то существуют такие подмножества X и Y множества E, что $X\in\Phi$, $Y\in\Phi$ и $X\cup\{x\}\notin\Phi$ для каждого $x\in Y$. Выберем такое e, что $0<e<\dfrac{|Y|}{|X|}-1$ и определим функцию w следующим образом:

$$w(x)=\begin{cases}1+e, & \text{если} \quad x\in X,\\ 1, & \text{если} \quad x\in Y-X,\\ 0, & \text{если} \quad x\notin X\cup Y.\end{cases}$$

Алгоритм СПО сначала выберет все элементы множества X, а затем отвергнет все элементы из $Y-X$. В результате будет построено множество A с весом $w(A)=(1+e)|X|<|Y|$, которое не является оптимальным, так как $W(Y)=|Y|$. □

Алгоритм Крускала – это алгоритм СПО, применяемый к семейству ациклических множеств ребер графа. Из теорем 2 и 3 следует, что он действительно решает задачу об оптимальном каркасе. В то же время существует много жадных алгоритмов, не являющихся алгоритмами типа СПО. Примером может служить алгоритм Прима. Эти алгоритмы не попадают под действие теоремы Радо-Эдмондса, для их обоснования нужна иная аргументация.

Если для некоторой конкретной задачи удалось установить применимость к ней алгоритма СПО, это не значит, что все проблемы позади. Этот алгоритм внешне очень прост, но он включает операцию проверки принадлежности множества семейству Φ, эффективное выполнение которой может потребовать серьезных дополнительных усилий. В алгоритме Крускала, например, для этого применяются специальные структуры данных. Ниже рассмотрим еще один пример, когда для успешного решения задачи алгоритм СПО комбинируется с известным методом чередующихся цепей для задачи о паросочетании. Заодно познакомимся с этим методом.

Паросочетания

Паросочетанием в графе называется множество ребер, попарно не имеющих общих вершин. Задача о паросочетании состоит в том, чтобы в данном графе найти

паросочетание с наибольшим числом ребер. Это число для графа G будем обозначать через $p(G)$. *Реберным покрытием* графа называется такое множество ребер, что всякая вершина графа инцидентна хотя бы одному из этих ребер. Наименьшее число ребер в реберном покрытии графа G обозначим через $r(G)$. Заметим, что реберное покрытие существует только для графов без изолированных вершин.

Определение паросочетания похоже на определение независимого множества вершин, паросочетание иногда так и называют – независимое множество ребер. Эта аналогия усиливается еще тесной связью между реберными покрытиями и паросочетаниями, подобно тому, как связаны между собой вершинные покрытия и независимые множества. Даже равенство, количественно выражающее эту связь, имеет точно такой же вид (напомним, что числа независимости $a(G)$ и вершинного покрытия $b(G)$ связаны равенством $a(G)+b(G)=n$). Приведем доказательство этого факта, так как оно имеет алгоритмическое значение – показывает, как каждая из двух задач сводится к другой.

Теорема 4. *Для любого графа G с n вершинами, не имеющего изолированных вершин, справедливо равенство* $p(G)+r(G)=n$.

Доказательство. Пусть M – наибольшее паросочетание в графе G. Обозначим через W множество всех вершин графа, не покрытых ребрами этого паросочетания. Тогда $|W|=n-2p(G)$. Очевидно, W – независимое множество (иначе M не было бы наибольшим). Выберем для каждой вершины из M какое-нибудь инцидентное ей ребро. Пусть F – множество всех выбранных ребер. Тогда $M\cup F$ – реберное покрытие и $|M\cup F|=n-p(G)$, следовательно, $r(G)\le n-p(G)$.

Обратно, пусть C – наименьшее реберное покрытие графа G. Рассмотрим подграф H графа G, образованный ребрами этого покрытия. В графе H один из концов каждого ребра является вершиной степени 1 (ребро, каждая вершина которого инцидентна по крайней мере еще одному ребру, можно было бы удалить из C, оставшиеся ребра по-прежнему покрывали бы все вершины). Отсюда следует, что каждая компонента связности графа H является звездой (звезда – это дерево, у которого не более одной вершины степени больше 1). Так как в любом лесе сумма количеств ребер и компонент связности равна числу вершин, то число компонент связности в графе H равно $n-r(G)$. Выбрав по одному ребру из каждой компоненты, получим паросочетание. Отсюда следует, что $p(G)\ge n-r(G)$. □

Несмотря на такое сходство между “вершинными” и “реберными” вариантами независимых множеств и покрытий, имеется кардинальное различие в сложности

соответствующих экстремальных задач. “Вершинные” задачи, как уже отмечалось, являются NP-полными. Для реберных же известны полиномиальные алгоритмы. Они основаны на методе чередующихся цепей, к рассмотрению которого мы теперь переходим.

Пусть G – граф, M – некоторое паросочетание в нем. Ребра паросочетания будем называть *сильными*, остальные ребра графа – *слабыми*. Вершину назовем *насыщенной*, если она инцидентна какому-либо ребру паросочетания, в противном случае *ненасыщенной*,. На рисунке 1 слева показан граф и в нем выделены ребра паросочетания $M = \{(2,3),(4,5),(7,8)\}$. Вершины 1 и 5 – ненасыщенные. Заметим, что к этому паросочетанию нельзя добавить ни одного ребра, т.е. оно максимальное. Однако оно не является наибольшим. В этом легко убедиться, если рассмотреть путь 5,6,8,9,10,7,3,4 (показан пунктиром). Он начинается и оканчивается в ненасыщенных вершинах, а вдоль пути чередуются сильные и слабые ребра. Если на этом пути превратить каждое сильное ребро в слабое, а каждое слабое – в сильное, то получится новое паросочетание, показанное на рисунке справа, в котором на одно ребро больше. Увеличение паросочетания с помощью подобных преобразований – в этом и состоит суть метода чередующихся цепей.

Рис. 1

Сформулируем необходимые понятия и докажем теорему, лежащую в основе этого метода чередующихся цепей. *Чередующейся цепью* относительно данного паросочетания называется простой путь, в котором чередуются сильные и слабые ребра (т.е. за сильным ребром следует слабое, за слабым – сильное). Чередующаяся цепь называется *увеличивающей*, если она соединяет две ненасыщенные вершины. Если M – паросочетание, P – увеличивающая цепь относительно M, то легко видеть, что $M \otimes P$ – тоже паросочетание и $|M \otimes P| = |M| + 1$.

Теорема 5. *Паросочетание является наибольшим тогда и только тогда, когда относительно него нет увеличивающих цепей.*

Доказательство. Если есть увеличивающая цепь, то, поступая так, как в рассмотренном примере, т.е. заменяя вдоль этой цепи сильные ребра на слабые и наоборот, мы, очевидно, получим большее паросочетание. Для доказательства обратного

утверждения рассмотрим паросочетание M в графе G и предположим, что M не наибольшее. Покажем, что тогда имеется увеличивающая цепь относительно M. Пусть M' – другое паросочетание и $|M'|>|M|$. Рассмотрим подграф H графа G, образованный теми ребрами, которые входят в одно и только в одно из паросочетаний M, M'. Иначе говоря, множеством ребер графа H является симметрическая разность $M \otimes M'$. В графе H каждая вершина инцидентна не более чем двум ребрам (одному из M и одному из M'), т.е. имеет степень не более двух. В таком графе каждая компонента связности – путь или цикл. В каждом из этих путей и циклов чередуются ребра из M и M'. Так как $|M'|>|M|$, то имеется компонента, в которой ребер из M' содержится больше, чем ребер из M. Это может быть только путь, у которого оба концевых ребра принадлежат M'. Легко видеть, что относительно M этот путь будет увеличивающей цепью. □

Для решения задачи о паросочетании остается научиться находить увеличивающие цепи или убеждаться, что таких нет. Тогда, начиная с любого паросочетания (можно и с пустого множества ребер), можем строить паросочетания со все увеличивающимся количеством ребер до тех пор, пока не получим такое, относительно которого нет увеличивающих цепей. Оно и будет наибольшим. Известны эффективные алгоритмы, которые ищут увеличивающие цепи для произвольных графов, но они достаточно сложны. Рассмотрим более простой алгоритм, решающий эту задачу для двудольных графов.

Пусть $G = (A, B, E)$ – двудольный граф, M – паросочетание в G, X и Y – множества ненасыщенных вершин соответственно в долях A и B. Всякая увеличивающая цепь, если такая имеется, соединяет вершину из A с вершиной из B. Ориентируем ребра графа G следующим образом: всякое слабое ребро в направлении от A к B, а всякое сильное – в направлении от B к A. Полученный орграф обозначим через $\vec{G}$. Очевидно, что увеличивающая цепь в графе G существует тогда и только тогда, когда в графе $\vec{G}$ имеется ориентированный путь из какой-либо вершины $x \in X$ в какую-либо вершину $y \in Y$. Такой путь можно найти (или убедиться в его отсутствии) простым поиском в ширину. По сравнению с ранее описанным алгоритмом поиска в ширину возникают два новых момента:

- Поиск в ширину проводится на ориентированном графе. На самом деле алгоритм остается тот же, только под $V(x)$ нужно понимать множество вершин, в которые ведут ориентированные ребра из вершины x.

- Обход производится не из одной вершины, а из множества вершин X. Все, что в связи с этим нужно изменить в алгоритме поиска в ширину – вначале заносить в очередь, используемую в этом алгоритме, не одну стартовую вершину, а все вершины множества X.

В целом алгоритм нахождения наибольшего паросочетания в двудольном графе можно теперь описать следующим образом.

Алгоритм 4. *Построение наибольшего паросочетания M в двудольном графе*

$G = (A, B, E)$

1 Построить граф $\vec{G}$, ориентировав все ребра графа G от A к B.

2 Положить X = множеству всех вершин из A, не являющихся концами ориентированных ребер; Y = множеству всех вершин из B, не являющихся началами ориентированных ребер;

3 Выполнить поиск в ширину в графе $\vec{G}$ из множества X. Если в процессе обхода встретилась вершина $y \in Y$, то

4 найти ориентированный путь P, ведущий в y из некоторой вершины $x \in X$;

5 изменить ориентацию всех ребер пути P на противоположную;

6 перейти к 2.

7 Положить M = множеству всех ребер, ориентированных от B к A.

Основной цикл (2–6) в этом алгоритме повторяется не более $n/2$ раз, поэтому общая оценка трудоемкости имеет вид $O(mn)$. В настоящее время известны и более быстрые алгоритмы для построения наибольших паросочетаний в двудольных графах.

Рассмотрим теперь следующую задачу. Дан двудольный граф $G = (A, B, E)$ и для каждой вершины $x \in A$ задан положительный вес $w(x)$. Требуется найти такое паросочетание в этом графе, чтобы сумма весов вершин из доли A, инцидентных ребрам паросочетания, была максимальной. Эту задачу иногда интерпретируют следующим образом. A – это множество работ, а B – множество работников. Ребро в графе G соединяет вершину $a \in A$ с вершиной $b \in B$, если квалификация работника b позволяет ему выполнить работу a. Каждая работа выполняется одним работником. Выполнение работы a принесет прибыль $w(a)$. Требуется так назначить работников на работы,

чтобы максимизировать общую прибыль. Покажем, что эта задача может быть решена алгоритмом СПО в сочетании с методом чередующихся цепей.

Множество $X \subseteq A$ назовем *отображаемым*, если в графе G существует такое паросочетание M, насыщающее все вершины из X. M в этом случае будем называть *отображением* для X. Пусть Φ – семейство всех отображаемых множеств.

Теорема 6. *Пара* (A, Φ) *является матроидом.*

Доказательство. Условие (1) определения матроида, очевидно, выполняется. Докажем, что выполняется и условие (2). Пусть $X \in \Phi$, $Y \in \Phi$, $|X|<|Y|$. Рассмотрим подграф H графа G, порожденный всеми вершинами из $X \cup Y$ и всеми смежными с ними вершинами из доли B. Пусть M_X – отображение для X, M_Y – для Y. Так как M_X не является наибольшим паросочетанием в графе H, то по теореме 5 относительно него в этом графе существует увеличивающая цепь. Одним из концов этой цепи является ненасыщенная относительно M_X вершина $a \in Y$. После увеличения паросочетания M_X с использованием этой цепи, как было описано выше, получим паросочетание M', отображающее множество $X \cup \{a\}$. Следовательно, $X \cup \{a\} \in \Phi$. □

Даже если бы в задаче требовалось только найти отображаемое множество наибольшего веса, проверка принадлежности множества семейству Φ требовала бы и нахождения соответствующего отображения, т.е. паросочетания. На самом же деле построение паросочетания входит в условие задачи. Комбинируя СПО с алгоритмом поиска увеличивающих цепей, получаем следующий алгоритм.

Алгоритм 5. *Построение паросочетания наибольшего веса в двудольном графе* $G = (A, B, E)$ *с заданными весами вершин доли* A.

1 Упорядочить элементы множества A по убыванию весов:
$A = \{a_1, a_2, \mathbf{K}, a_k\}$, $w(a_1) \geq w(a_2) \geq \mathbf{K} \geq w(a_k)$

2 $X = \varnothing$

3 $M = \varnothing$

4 **for** $i = 1$ **to** k **do**

5 **if** в G существует увеличивающая цепь P относительно M, начинающаяся в вершине a_i,

6 **then** $\{X = X \cup \{a_i\};\ M = M \otimes P\}$

Если для поиска увеличивающей цепи применить метод поиска в ширину, как описано выше, то время поиска будет пропорционально числу ребер. Общая трудоемкость алгоритма будет $O(mk)$, где k – число ребер в доле A.

ГЛАВА 5. ПРИОРИТЕТНЫЕ ОЧЕРЕДИ

Основные определения

Приоритетная очередь это абстрактный тип данных, значениями которого являются взвешенные множества. Множество называется взвешенным, если каждому его элементу однозначно соответствует вещественное число, называемое ключом или весом. Основными операциями над приоритетной очередью являются следующие:

1 *ВСТАВИТЬ* в очередь новый элемент со своим ключом.

2 *НАЙТИ* в очереди элемент с минимальным ключом. Если таких элементов несколько, то находится один из них. Найденный элемент не удаляется.

3 *УДАЛИТЬ* из очереди элемент с минимальным ключом. Если таких элементов несколько, то удаляется любой один из них.

4 *УМЕНЬШИТЬ* ключ указанного элемента на заданное положительное число.

5 *ОБЪЕДИНИТЬ* две очереди в одну.

Операция *ОБЪЕДИНИТЬ* иногда не включается в число основных операций, но если она используется, то очереди называются объединяемыми.

Приоритетная очередь естественным образом применяется в таких задачах, как сортировка элементов массива во внутренней памяти (пирамидальная сортировка), отыскание в графе минимального связывающего дерева (алгоритм Крускала), отыскание кратчайших путей от заданной вершины графа до его остальных вершин (алгоритм Дейкстры) и при алгоритмизации многих других задач.

Приоритетную очередь естественным (наивным) образом можно представить с помощью массива или списка элементов, но такие реализации не эффективны по времени выполнения основных операций. Так, например, поиск элемента с минимальным ключом в неупорядоченном массиве или списке связан с последовательным просмотром всех его элементов и поэтому требует O(n) единиц времени, где n - число элементов в очереди. Если поддерживать упорядоченность массива или списка по ключу, то "неудобной" окажется операция вставки нового элемента.

Чаще всего приоритетная очередь представляется с помощью так называемых кучеобразных структур. Кучеобразная структура строится на основе корневого дерева или набора корневых деревьев с определенными свойствами. Узлы дерева отождествляются с элементами рассматриваемого множества.

Соответствие между узлами дерева и элементами множества называется куче образным, если ключ элемента, приписанного узлу, не превосходит ключей элементов, приписанных его потомкам.

Такие представления взвешенных множеств называются кучами (*heap*). Вид дерева и способ его представления в памяти компьютера подбирается в зависимости от тех операций, которые предполагается выполнять над множеством и от того, насколько эти операции сказываются на суммарной трудоемкости алгоритма.

Представление приоритетной очереди с помощью *d*-кучи

Представления приоритетной очереди с помощью – *d*-кучи ($d \geq 2$), основано на использовании так называемых завершенных *d*-арных деревьев.

Дерево, состоящее из n узлов, называется завершенным d-арным деревом, если его узлы можно пронумеровать числами от 0 до n-1 так, что: корень получит номер 0, потомки узла с номером i получат номера из множества: $\{i \cdot d + 1,\ i \cdot d + 2,..., i \cdot d + d\}$, но не больше чем n-1.

Такая нумерация удобна тем, что позволяет разместить узлы дерева в массиве в порядке возрастания их номеров, при этом, как следует из определения, номера позиций потомков любого узла в массиве легко вычисляются по номеру позиции самого узла. Так же легко по номеру позиции узла вычисляется номер позиции его родителя. Так для узла из позиции с номером i, родительский узел будет расположен в позиции (i-1) div d, где div – операция деления нацело.

При изображении завершенного *d*-арного дерева узлы одинаковой глубины удобно располагать на одном уровне, при этом потомки одного узла располагаются слева направо в порядке установленных номеров. При таком рисовании нижний уровень заполняется, возможно, не полностью.

Очевидны следующие свойства завершенного *d*-арного дерева:

1 Каждый внутренний узел (то есть узел, не являющийся листом дерева), за исключением, быть может, только одного, имеет ровно d потомков. Один узел-исключение может иметь любое количество от 1 до $d - 1$ потомков.

2 Если k – глубина дерева, то для любого $i = 1, ..., k - 1$ такое дерево имеет ровно d^i узлов глубины i.

3 Количество узлов глубины k в дереве глубины k может варьироваться от 1 до d^k.

Отметим еще некоторые простые утверждения о завершенных *d*-арных деревьях, которые будут полезны при анализе трудоемкости основных операций.

Утверждение 1. Длина h пути от корня завершенного *d*-арного дерева, имеющего n узлов ($n > 0$), в любой лист удовлетворяет неравенствам: $\log_d n - 1 < h < \log_d n + 1$.

Доказательство. Минимальное количество узлов в d-куче высоты h (при $h > 0$), по свойствам 2 и 3 d-арного дерева, очевидно, равно $1 + d + d^2 + \ldots + d^{h-1} + 1$ (последний уровень содержит лишь одну вершину).

Максимальное количество узлов в такой d-куче равно $1 + d + d^2 + \ldots + d^h$ (все уровни, включая последний, заполнены). Отсюда имеем неравенства:

$$(1 + d + d^2 + \cdots + d^{h-1} + 1) \le n \le (1 + d + d^2 + \cdots + d^h).$$

Суммируя левую и правую части как геометрические прогрессии, получим

$$1+(d^h - 1) / (d - 1) \le n \le (d^{h+1} - 1) / (d - 1),$$

и после некоторых очевидных оценок с помощью логарифмирования получаем требуемые неравенства:

$$\log_d n - 1 < h < \log_d n + 1.$$

Утверждение 2. Количество узлов высоты h не превосходит n/d^h. Под высотой узла понимается расстояние от него до наиболее далекого потомка.

Кучу, содержащую n элементов, будем представлять двумя массивами $a[0 .. n\text{-}1]$ и $key[0 .. n\text{-}1]$, считая $a[i]$ ссылкой на элемент, соответствующий узлу i; $key[i]$ его ключ. Иногда под $a[i]$ удобно понимать вместо ссылки сам элемент исходного множества. В некоторых прикладных задачах нет необходимости помещать в приоритетную очередь ни сами элементы, ни их имена, в таких случаях при организации кучи используется лишь массив $key[0..(n\text{-}1)]$. На рисунке 1 приведен пример кучи для $d = 3$, $n = 18$. Кружочками изображены узлы дерева, в них записаны элементы массива, представляющие имена элементов кучи.

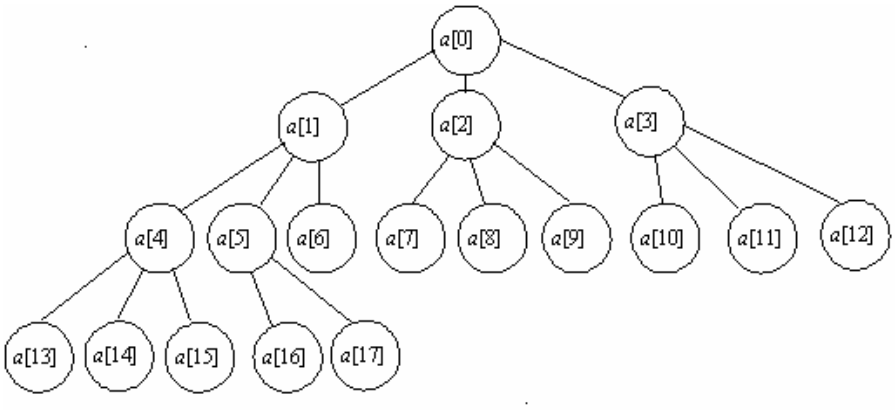

Рис. 1.

При реализации основных операций над кучами используются две вспомогательные операции – *ВСПЛЫТИЕ* и *ПОГРУЖЕНИЕ*, применяемые к некоторому узлу дерева. При реализации этих операций будем использовать еще одну вспомогательную операцию – транспонирование, меняющую местами элементы, расположенные в двух разных узлах i и j дерева. Ее реализация может быть представлена процедурой $tr(i, j)$:

```
procedure tr(i, j: integer); var tmp: integer;
begin
   tmp:= a[i]; a[i]:= a[j]; a[j]:= tmp;
   tmp:= key[i]; key[i]:= key[j]; key[j]:= tmp;
end;
```

Замечание. В этой процедуре тип элемента и его ключ считаются целочисленными. Если в кучу помещаются только ключи элементов, то процедура транспонирования модифицируется соответствующим образом.

Операция *ВСПЛЫТИЕ*

Эта операция может быть применена в случаях, когда в некотором узле кучи, расположен элемент *x*, ключ которого меньше ключа его родителя *y*. Операция меняет местами *x* и *y*. Если после этого элемент *x* снова не удовлетворяет условиям кучи, то еще раз проводится аналогичная перестановка. И так до тех пор, пока *x* не перестанет нарушать кучеобразный порядок.

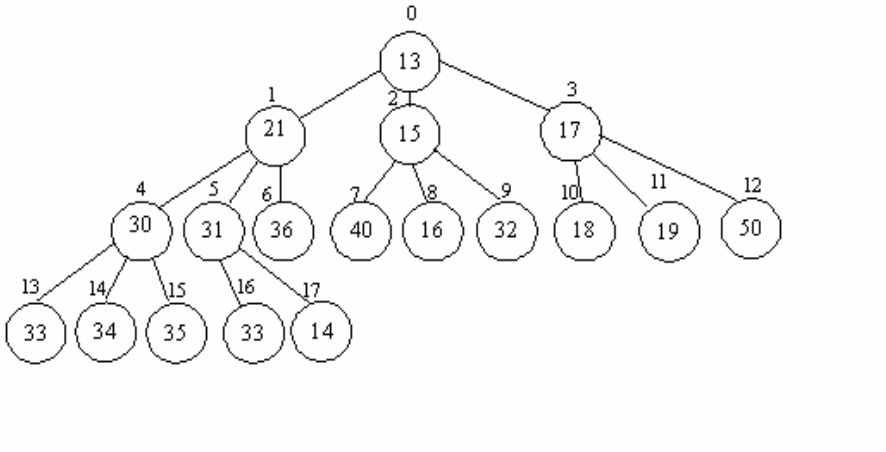

Рис. 2.

Рассмотрим дерево ($d = 3$), представленное на рисунке 2. В этом дереве кучеобразный порядок нарушает элемент с ключом 14, приписанный узлу с номером 17, так как его родительскому узлу приписан элемент с ключом 31 > 14.

Применим к узлу 17 операцию *ВСПЛЫТИЕ*. Транспонируются узлы 17 и 5, затем 5 и 1.

В результате получаем кучу, изображенную на рисунке 3. Кучеобразный порядок восстановлен, операция *ВСПЛЫТИЕ* завершена.

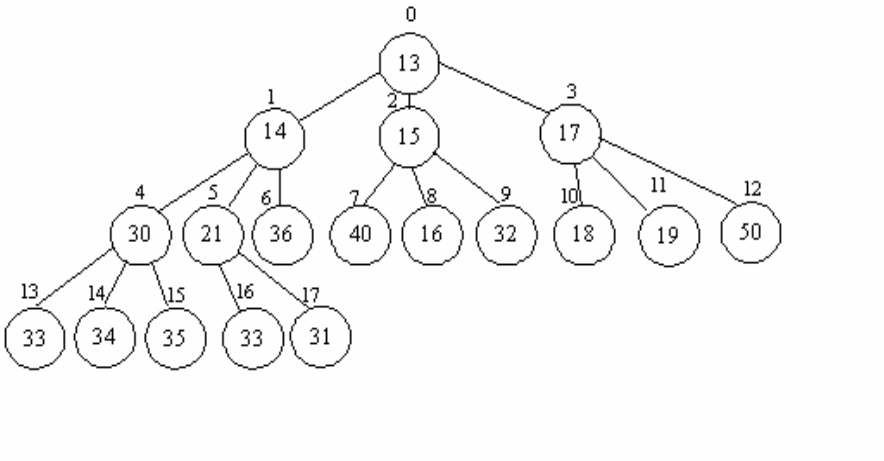

Рис. 3.

Вычислительная сложность этой операции пропорциональна числу сравнений элементов и их обменов. Это число, очевидно, не более чем удвоенное число узлов в пути от узла x до корня дерева. Длина такого пути в d-куче с n узлами не превосходит ее высоты, а именно $\log_d n + 1$, по доказанному выше утверждению 1. Значит, время выполнения данной операции – $O(\log_d n)$.

Операцию *ВСПЛЫТИЕ* можно реализовать с помощью представленной ниже процедуры ***Emersion*.** Входным параметром процедуры является номер i узла, который нарушает порядок описанным выше способом.

```
procedure Emersion(i: integer); var p: integer;
begin while i > 0 do
   begin p:= (i-1) div d;  if key[p] > key[i] then begin tr(i, p); i:= p end else exit end
end;
```

Замечание. Для более эффективного выполнения операции *ВСПЛЫТИЕ* можно поступить следующим образом. Запомнить элемент, находящийся в узле i, переместить элемент из его родительского узла $p = (i - 1)\ div\ d$ в узел i, затем из узла $(p - 1)\ div\ d$ в узел p и так до тех пор, пока не освободится узел для запомненного элемента. После этого поместить запомненный элемент на освободившееся место.

Операция *ПОГРУЖЕНИЕ*

Эта операция также применяется при восстановлении свойства кучеобразности. Пусть, например, в i-ом узле расположен элемент x, ключ которого больше ключа хотя бы одного из своих непосредственных потомков y. В этом случае среди непосредственных потомков узла i выбирается потомок y с наименьшим ключом, и элементы x и y меняются местами. Если после этого элемент x снова не удовлетворяет условиям кучи, то еще раз проводим аналогичную перестановку. И так до тех пор, пока x не встанет на свое место.

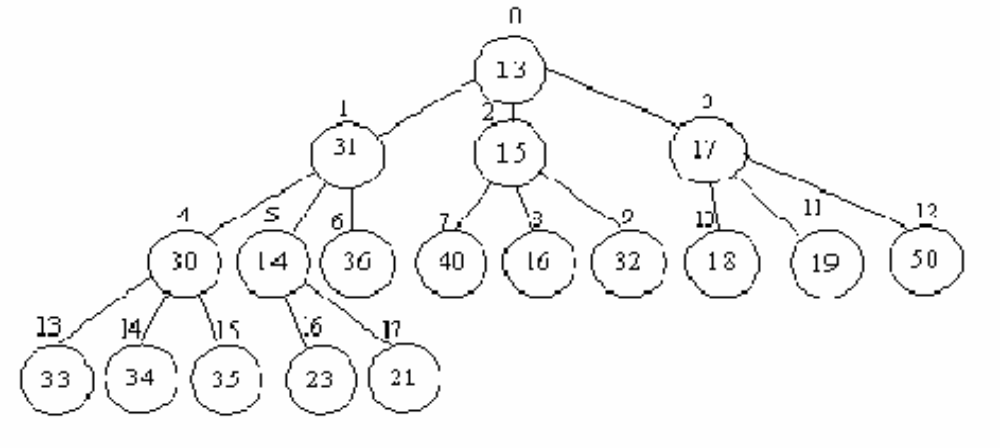

Рис. 4.

Рассмотрим представленное на рисунке 4 дерево. В узле 1 расположен элемент *x* с ключом 31, и этот узел имеет двух потомков с меньшими ключами, а именно 30 и 14. Применим к элементу *x* операцию *ПОГРУЖЕНИЕ*.

Среди непосредственных потомков узла 1 находим узел, которому приписан элемент *y* с наименьшим ключом, в нашем случае это узел 5, с ключом 14. Транспонируем узлы 1 и 5, затем 5 и 17. В результате кучеобразный порядок восстанавливается.

Вычислительная сложность этой операции пропорциональна числу сравнений элементов и их обменов. Для каждого узла в пути следования данной операции производится *d* сравнений при поиске потомка с минимальным ключом и один обмен. Длина этого пути в *d*-куче с *n* узлами по утверждению 1 не превосходит ее высоты, а именно $\log_d n$,. Следовательно, операция требует в худшем случае $O(d\log_d n)$ времени. Для ее реализации воспользуемся функцией *Minchild*(*i*), позволяющей для любого узла *i* находить его непосредственного потомка с минимальным ключом. Если у узла *i* нет потомков, то условно считаем *Minchild*(*i*) = 0.

```
procedure Diving(i: integer);var j: integer;
begin
  repeat j:=Minchild(i); if j = 0 then exit;
    if (key[i] > key[j])then begin tr(i, j); i:= j end else exit;
  until false
end;
```

Операцию *ПОГРУЖЕНИЕ* можно реализовать с помощью представленной процедуры ***Diving.*** Входным параметром процедуры является номер *i* узла, который нарушает порядок описанным выше способом.

В процедуре *Diving* для поиска потомка узла *j* с минимальным ключом используется процедура-функция *Minchild*(*j*).

```
function Minchild(i: integer): integer; var first, last, j, minkey: integer;
begin  first:=i*d+1;
  if first > n-1 then begin minchild:= 0; exit end;
  last:= i*d+d;
  if last > n-1 then last:= n-1;
  minkey:= maxint; minchild:= 0;
  for j:= first to last do if key[j] < minkey then begin minkey:= key[j]; minchild:= j end
end;
```

Операция *ВСТАВКА*

Эта операция используется для добавления к приоритетной очереди нового элемента с именем ***nameX*** и с ключом ***keyX***.

Если перед выполнением этой операции куча содержала n узлов (напомним, что они пронумерованы числами от 0 до $n - 1$), то добавляем к дереву $(n + 1)$-ый узел (его номер будет n) и приписываем ему элемент с именем ***nameX*** и ключом ***keyX***.

Вставка нового элемента производится посредством отведения для него места в n-ых позициях массивов a и ***key*** соответственно, после чего к добавленному узлу применяется операция *ВСПЛЫТИЕ* для восстановления кучеобразного порядка.

Вставим в кучу изображенную на рисунке 5 новый элемент с ключом 14.

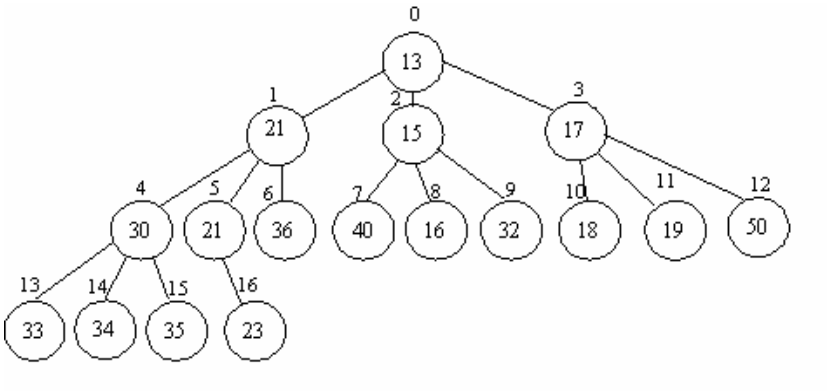

Рис. 5.

Сначала добавляем к дереву новый узел с номером 17 и приписываем ему элемент с ключом 14. Получим дерево, представленное на рисунке 6.

Затем применяем к узлу 17 операцию *ВСПЛЫТИЕ*. На этом операция завершается.

При описании этой операции (см. рисунки 2, 3, 4) использовался как раз этот пример. Операция *ВСТАВКА* завершена.

Вычислительная сложность операции определяется вычислительной сложностью операции *ВСПЛЫТИЕ*, то есть $O(\log_d n)$.

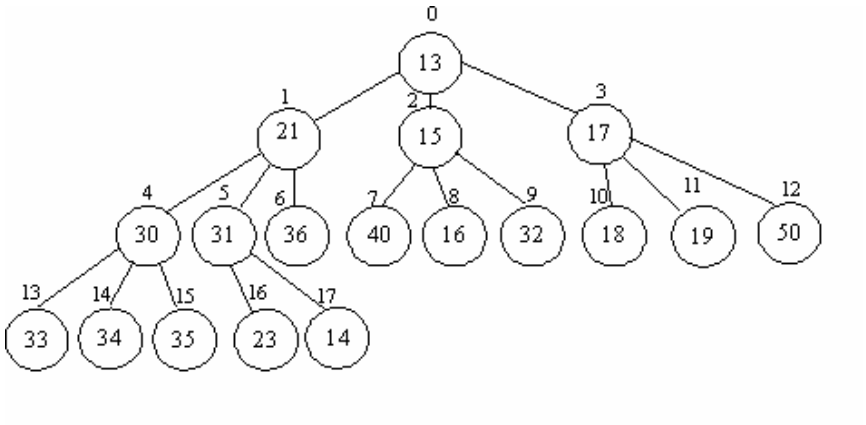

Рис. 6.

Реализовать операцию можно с помощью операторов

```
a[n]:= nameX; key[n]:= keyX; Emersion(n); n:= n + 1;
```

Операция *УДАЛЕНИЕ*

Используется для удаления элемента, приписанного узлу с заданным номером *i*. Сначала элемент, приписанный последнему узлу дерева, переносится на место удаляемого элемента, последний узел при этом становится ненужным и поэтому удаляется из дерева посредством операции *n*:= *n*-1. Далее, если узел *i*, в который помещен новый элемент, имеет родителя с большим ключом, то к узлу *i* применяется операция *ВСПЛЫТИЕ*. В противном случае к узлу *i* в случае необходимости применяется операция *ПОГРУЖЕНИЕ*. На этом операция *УДАЛЕНИЕ* заканчивается.

Вычислительная сложность операции удаления равна константе плюс вычислительная сложность худшего случая применения операций *ВСПЛЫТИЕ* или *ПОГРУЖЕНИЕ*, то есть в худшем случае получается $O(d^{\backslash}\log_d n)$.

Реализовать операцию можно с помощью операторов

```
a[i]:= a[n - 1]; key[i]:= key[n - 1]; n:= n – 1;
if (i ≠ 0) and (key[i] < key[(i - 1) div d]) then Emersion(i) else Diving(i);
```

Операция *НАЙТИ* минимум

Предназначена для нахождения элемента с минимальным ключом без удаления его из очереди. Поскольку искомый элемент всегда находится в корне дерева, трудоемкость операции, очевидно, равна O(1).

Операция *УДАЛЕНИЕ_МИНИМУМА*

Эта операция предназначена для нахождения элемента с минимальным ключом и удаления его из очереди. Реализовать операцию можно с помощью операторов

```
a[0]:= a[n - 1]; key[0]:= key[n - 1]; n:= n - 1; Diving(0);
```

Операция *УМЕНЬШЕНИЕ_КЛЮЧА*

Предназначена для уменьшения ключа у элемента, приписанного узлу с заданным номером i, на заданную величину Δ. Это действие может нарушить кучеобразный порядок лишь таким образом, что уменьшенный ключ элемента в узле i будет меньше ключа его родителя. Для восстановления порядка в куче используется операция *ВСПЛЫТИЕ*.

Вычислительная сложность операции состоит из времени, затрачиваемого на уменьшение ключа (то есть константы), и времени выполнения операции *ВСПЛЫТИЕ* то есть $O(\log_d n)$. В итоге вычислительная сложность операции равна $O(\log_d n)$.

Реализовать операцию можно с помощью операторов

```
key[i] := key[i] – delta; Emersion(i);
```

Операция *ОКУЧИВАНИЕ*

Заметим, что если d-куча создается путем n-кратного применения операции *ВСТАВКА*, то суммарная трудоемкость ее создания будет равна $O(n \log_d n)$.

Если же ее элементы уже занимают в произвольном порядке массивы $a[0 .. n-1]$ и, соответственно, $key[0..(n-1)]$, то можно превратить их в d-кучу, применяя операцию *ПОГРУЖЕНИЕ* по очереди к узлам $(n-1)$, $(n-2)$, …, 0.

Такой процесс будем называть окучиванием массива. Для доказательства того, что в результате действительно устанавливается кучеобразный порядок достаточно заметить, что если поддеревья с корнями в узлах $i+1$, $i+2$, …, $n-1$ упорядочены по правилу кучи, то после применения процедуры *ПОГРУЖЕНИЕ* к узлу i поддерево с корнем в этом узле также станет упорядоченным по правилу кучи.

Итак, остановимся на следующей реализации с помощью процеуры ***MakeHeap***.

```
procedure MakeHeap; var i: integer;
begin for i:= n - 1 downto 0 do Diving(i) end;
```

Оказывается, вычислительная сложность операции *ОКУЧИВАНИЕ* оценивается величиной $O(n)$.

Для доказательства заметим, что трудоемкость погружения с высоты h есть величина $O(h)$, а количество узлов высоты h не превосходит n/d^h. Осталось оценить сумму

$$\sum_{h=1}^{H} h \frac{n}{d^h},$$

где H – высота кучи, и убедиться, что полученная сумма есть $O(n)$. Для суммирования можно воспользоваться известной формулой

$$\sum_{i=1}^{k} \frac{i}{x^i} = \frac{x^{k+1} - (k+1)x + k}{x^k (x-1)^2}.$$

Заметим также, что узлы с номерами большими чем (*n* - 1) **div** *d* потомков не имеют, поэтому в процедуре *MakeHeap* оператор

```
for i := n – 1 downto 0 do Diving(i);
```

можно заменить оператором

```
for i:= (n - 1) div d downto 0 do Diving(i);
```

Операция *СОЗДАТЬ_СПИСОК_МИНИМАЛЬНЫХ*

Применяется для получения списка элементов, которые имеют ключи меньшие заданного значения *key*0. Этой операции нет в списке обязательных операций, но ее легко реализовать следующим образом. Если ключ элемента, находящегося в корне больше, чем *key*0, то это дерево не имеет искомых элементов. В противном случае включаем его в выходной список *S*, а затем применяем ту же процедуру ко всем потомкам узла, включенного в список.

Пусть куча содержит *k* элементов с ключами меньшими, чем *key*0. По свойству кучи, они все расположены на ее "верхушке". Данная процедура обходит эту верхушку за время, пропорциональное *k*, и для каждого из этих *k* элементов просматривает все его *d* (или меньше) непосредственных потомков. Получаем, что время выполнения операции является величиной O(*d*·*k*).

Реализовать операцию можно с помощью операторов

```
Инициализировать пустой список S;
Инициализировать пустой стек;
while стек не пуст do
    стек ⇒ i;
    if (key[i] < key0) then
        Добавить a[i] к списку S;
        j:= d·i + 1;
        while j ≤ max(n – 1, d·i + d) do {j ⇒ стек; j:= j + 1};
```

Сводные данные о трудоемкости операций с *d*-кучами

ВСПЛЫТИЕ(*i*)	$O(\log_d n)$
ПОГРУЖЕНИЕ(*i*)	$O(d \log_d n)$
ВСТАВКА(*nameX*, *keyX*)	$O(\log_d n)$
УДАЛЕНИЕ(*i*)	$O(d \log_d n)$
УДАЛЕНИЕ_МИН(*nameX*, *keyX*)	$O(d \log_d n)$

НАЙТИ_МИНИМУМ	O(1)
УМЕНЬШЕНИЕ_КЛЮЧА(i, Δ)	O($\log_d n$)
ОБРАЗОВАТЬ_ОЧЕРЕДЬ	O(n)
СПИСОК_МИН(x, h)	O($d \cdot k$)

Замечание. Представление приоритетной очереди с помощью *d*-кучи может быть "неудобным" если в прикладной задаче требуется операция слияния двух очередей в одну.

Применение приоритетных очередей в задаче сортировки массивов

Под задачей сортировки в простейшем случае понимают следующее: дана последовательность (*key*[1], *key*[2], ..., *key*[*n*]) из *n* элементов некоторого линейно упорядоченного множества, например, целых или вещественных чисел, записанных в массив *key*. Требуется переставить элементы массива так, чтобы после перестановки выполнялись неравенства:

$$key[1] \le key[2] \le ... \le key[n].$$

Уточнения этой задачи связаны с теми средствами, с помощью которых предполагается ее решение. Нас интересуют алгоритмы с точки зрения их компьютерной реализации. Оценивая качество различных алгоритмов, обычно интересуются тем, как зависит время работы алгоритма от длины *n* сортируемой последовательности и требуется ли для этого дополнительная память, размер которой зависит от *n*. Существенную роль при этом играет метод доступа к элементам памяти. При сортировке во внутренней (оперативной) памяти обычно используется прямой доступ, а во внешней – последовательный.

Бесхитростная сортировка

Бесхитростный алгоритм сортировки в памяти с прямым доступом может заключаться в выполнении следующих операторов

```
for k:=1 to n - 1 do for i:= k + 1 to n do if key[i] < key[k] then tr(i, k);
```

где *tr*(*i*, *k*) – процедура, транспонирующая элементы *key*[*i*], *key*[*k*]. Заметим, что число сравнений "*key*[*i*] < *key*[*k*]" при реализации такого алгоритма равно $n\,(n-1)/2$. Это означает, что его трудоемкость оценивается величиной $O(n^2)$.

Сортировка с помощью *d*-кучи

Используя структуру *d*-кучи, сортировку массива *key* можно провести в два этапа.

1 Произвести *ОКУЧИВАНИЕ* массива *key*.

2 Осуществить окончательную сортировку следующим образом. Первый (минимальный) элемент кучи и последний поменять местами, уменьшить размер кучи на 1. Минимальный элемент остается в последней позиции массива *key*, не

являясь уже элементом кучи. Применить операцию *ПОГРУЖЕНИЕ* к корню. Повторять аналогичные действия, пока размер кучи не станет равным 1.

Эти два этапа реализуются с помощью процедуры *SORT*, которая сортирует массив по убыванию ключей.

```
procedure SORT;
begin
  for i:= (n - 1) div d downto 0 do Diving(i);
  while n > 0 do begin tr(0, n-1); n:= n – 1; Diving(0) end
end;
```

Чтобы не загромождать процедуру *SORT* лишними описаниями, мы считаем массив *key* и использованные процедуры *tr* и *Diving* глобальными. Заметим, что процедура *SORT* не требует дополнительной памяти, размер которой зависел бы от длины массива *key*. Трудоемкость сортировки массива с помощью этой процедуры очевидно равна $O(n \cdot log\ n)$.

Сортировка с помощью d-кучи известна в литературе как «пирамидальная сортировка».

Еще одной задачей в которой естественным образом используется приоритетная очередь является задача о нахождении кратчайших путей в графе со взвешенными ребрами с помощью известного алгоритма Дейкстры.

Нахождение кратчайших путей в графе

Входные данные:

1. Граф *G* со взвешенными ребрами. Считаем, что вершины графа пронумерованы числами от 1 до *n* и отождествляются с их номерами. Под весами ребер понимаем некоторые числовые характеристики ребер, например, длины ребер, если речь идет о геометрическом графе. Под *L*(*i, j*) понимаем вес ребра (*i, j*), соединяющего вершину *i* с вершиной *j*.

2. Стартовая вершина *s* это вершина, от которой находятся кратчайшие пути до всех остальных вершин.

Выходные данные:

3. Массив *dist*[1..*n*], (*dist*[*i*] – кратчайшее расстояние от вершины *s* до вершины *i*).

4. Массив *up*[1..*n*], (*up*[*i*] – предпоследняя вершина в кратчайшем пути из *s* в *i*).

Алгоритм Дейкстры

1 $\forall i \in [1..n]$ {*up*[*i*]:= *i*; *dist*[*i*]:= *gz*}; *dist*[*s*]:= 0; .

2 Организовать приоритетную очередь из вершин графа, используя в качестве ключей элементы массива *dist* [1..*n*].

3 Пока очередь не пуста, выполнять пункт 4.

4 Выбрать (с удалением) из очереди вершину r_0 с минимальным ключом. Для каждой вершины r, смежной с r_0, вычислить $delta = dist[r] - (dist[r_0] + L(r_0, r))$ и если $delta > 0$, то $\{dist[r] := dist[r] - delta; up[r] := r_0\}$.

Заметим, что алгоритм Дейкстры корректно решает задачу для графов с неотрицательными весами вершин. Если же в графе есть ребра с отрицательными весами, но нет циклов с отрицательным суммарным весом, то для решения задачи можно использовать алгоритм Форда-Беллмана.

Сделаем еще несколько замечаний к реализации алгоритма.

- В качестве *gz* можно взять максимально возможное представимое в компьютере число, во всяком случае, оно должно быть больше чем длина наиболее длинного простого пути исходящего из вершины *s*. В процессе вычислений это число уменьшается и в итоге становится длиной кратчайшего пути из вершины *s* в соответствующую вершину *i*.

- Для реализации приоритетной очереди хорошо подходит реализация с помощью *d*-кучи, например, с параметром $d = 2$.

- Для того чтобы рационально в пункте 4 перебирать вершины графа, смежные с вершиной *r*0, можно граф представить массивом длины *n* в котором элемент с номером *i* является указателем на список вершин, смежных с *i*-ой вершиной. В свою очередь элементом списка может быть пара $[j, L(i, j)]$ состоящая из номера *j* очередной вершины смежной с вершиной *i* и длины $L(i, j)$ ребра (i, j).

- При организации очереди и при выполнении пункта 4 приходится по номеру вершины определять номер соответствующего узла в *d*-куче, для этого полезно иметь массив $map[1..n]$, в котором $map[i]$ – номер узла в *d*-куче, соответствующий вершине *i*. При выполнении операций с *d*-кучей массив *map* необходимо модифицировать во время выполнения транспонирования элементов в куче.

Представление приоритетной очереди левосторонними кучами

Мы уже отмечали, что реализация приоритетных очередей с помощью *d*-кучи не предусматривает возможности их объединения. Сейчас мы рассмотрим реализацию с помощью, так называемой левосторонней кучи.

Левосторонняя куча (*leftist heap*) – это представление приоритетной очереди с помощью **левостороннего бинарного дерева.**

Бинарным деревом называется корневое дерево, у которого каждый узел имеет не более двух непосредственных потомков. Один из них называется левым, другой – правым.

Узел называем неполным, если он имеет менее двух непосредственных потомков. В частности, листья дерева являются неполными узлами.

Рангом узла называем увеличенное на 1 расстояние (число ребер) от него до ближайшего неполного потомка.

Ранг узла можно определить также следующим образом. Расширить дерево, добавляя к каждому узлу, имеющему менее двух потомков, в том числе и к листьям, недостающее количество потомков. Затем каждому из листьев полученного дерева приписать ранг 0, а ранг каждого из остальных узлов определить как минимум из рангов его непосредственных потомков плюс 1. Очевидно, ранги вершин исходного дерева совпадут с рангами соответствующих вершин расширенного дерева.

Левостороннее дерево – это бинарное дерево, для каждого узла которого ранг его левого непосредственного потомка в расширенном дереве не меньше ранга его правого потомка.

Ветвью бинарного дерева называем последовательность его узлов, начинающуюся с корня, заканчивающуюся листом, и такую, что каждый следующий узел является непосредственным потомком предыдущего.

Правой ветвью дерева называем ветвь такую, что каждый следующий узел является непосредственным правым потомком предыдущего.

Из определения расширенного левостороннего дерева следует, что ранг узла в таком дереве равен длине исходящей из него правой ветви.

Пример левостороннего дерева (и его расширения) приведен на рисунке 1. Ребра исходного дерева изображены жирными линиями, а ребра, добавленные при расширении – пунктиром. Числа рядом с узлами – их ранги.

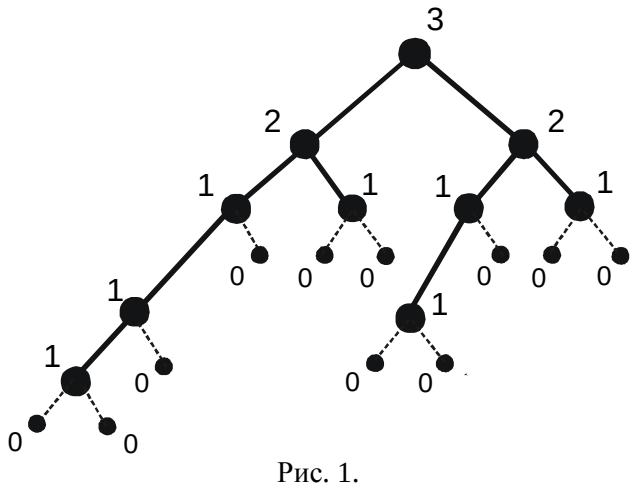

Рис. 1.

Свойства левостороннего дерева

1 Правая ветвь из любого узла дерева в лист имеет минимальную длину среди всех таких ветвей.

2 Длина h правой ветви, исходящей из корня левостороннего дерева, имеющего n узлов, есть величина O($\log n$).

Первое свойство непосредственно следует из определения левостороннего дерева. Для доказательства второго свойства рассмотрим левостороннее дерево T, у которого длина правой ветви равна h. Индукцией по числу h докажем, что число n узлов в таком дереве удовлетворяет неравенству $n \geq 2^h - 1$. Действительно, при $h = 1$ утверждение очевидно. При $h > 1$ левое и правое поддеревья дерева T будут левосторонними, а ранги их корней больше или равны $h - 1$. Следовательно, по предположению индукции число узлов в каждом из них больше или равно $2^{h-1} - 1$, а в дереве T – больше или равно

$$(2^{h-1} - 1) + (2^{h-1} - 1) + 1 = 2^h - 1.$$

Для реализации приоритетной очереди с помощью левосторонней кучи будем использовать узлы вида

node = (*element, key, rank, left, right, parent*),

содержащие следующую информацию:

11 *element* – элемент приоритетной очереди или ссылка на него,

12 *key* – его ключ (вес),

13 *rank* – ранг узла, которому приписан рассматриваемый элемент,

14 *left, right* – указатели на левое и правое поддеревья,

15 *parent* – указатель на родителя.

Куча представляется указателем на ее корень. Если h − указатель на корень кучи, то через h будем обозначать и саму кучу.

Операция *СЛИЯНИЕ*

Эта операция позволяет слить две левосторонние кучи $h1$ и $h2$ в одну кучу h. Реализуется она посредством слияния правых путей исходных куч в один правый путь, упорядоченный по правилам кучи, а левые поддеревья узлов сливаемых правых путей остаются левыми поддеревьями соответствующих узлов в результирующем пути. Очевидно в полученной куче свойство «левизны» может быть нарушено только у узлов исходящей из корня правой ветви, так как левые поддеревья этих узлов не изменились. Восстанавливается свойство левизны при помощи прохода правого пути снизу вверх (от листа к корню) с попутными транспонированиями в случае необходимости левых и правых поддеревьев и вычислением новых рангов проходимых узлов.

Рассмотрим процесс слияния двух левосторонних куч $h1$ и $h2$, изображенных на рисунке 2

Числа внутри кружочков являются ключами элементов, приписанных к соответствующим узлам. Правые ветви куч показаны жирными линиями. Числа рядом с узлами – их ранги.

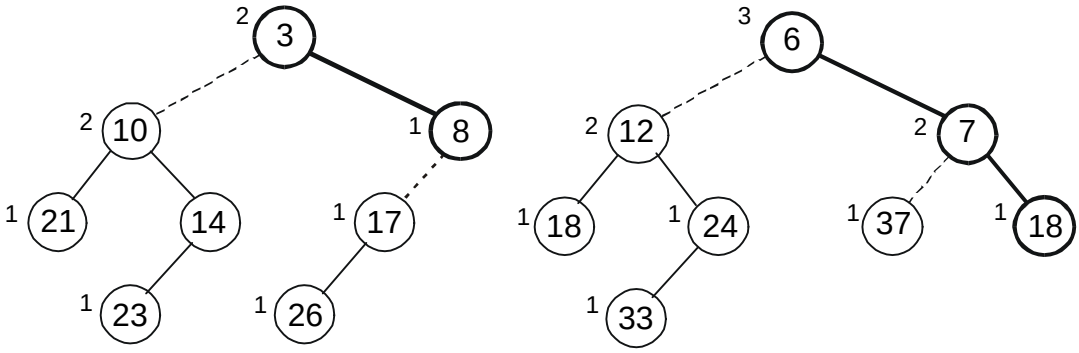

Рис. 2.

После объединения правых путей получим дерево, изображенное на рисунке 3. Оно не является левосторонним. В скобках указаны ранги узлов, какими они были в исходных кучах до слияния.

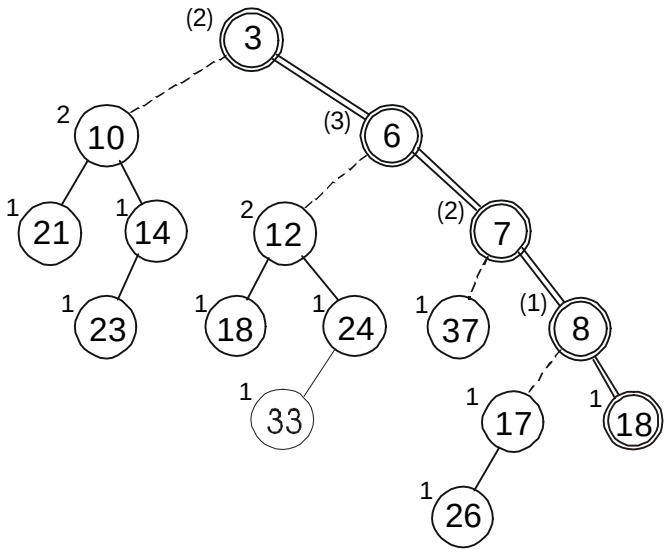

Рис. 3.

Восстановление свойства левизны кучи начинаем с последнего узла правой ветви. Это узел с ключом 18. Очевидно, он должен иметь ранг 1, совпадающий с его старым значением и поэтому не требующий обновления.

Следующий по направлению к корню узел правой ветви имеет ключ 8, ранг его левого потомка не меньше ранга правого потомка, следовательно, условие левизны выполняется и поэтому транспонирования его левого и правого поддеревьев не требуется. Однако, ранг этого узла необходимо обновить, так как его старое значение 1 не совпадает с увеличенным на 1 минимальным из рангов его потомков, то есть с числом 2. Обновив ранг, получим кучу, изображенную на рисунке 4.

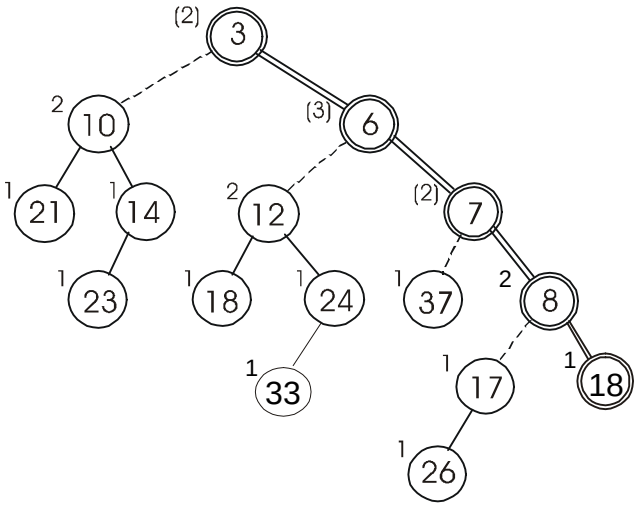

Рис. 4.

Теперь рассмотрим узел с ключом 7. Он имеет левого сына с ключом 37 и рангом 1, и правого сына с ключом 8 и рангом 2. Для восстановления свойства левизны в этом узле необходимо поменять местами его левое и правое поддеревья и обновить ранг. Его новым значением будет минимум из рангов его потомков (это ранг нового правого сына) плюс 1, то есть 2. В результате получаем дерево, изображенное на рисунке 5:

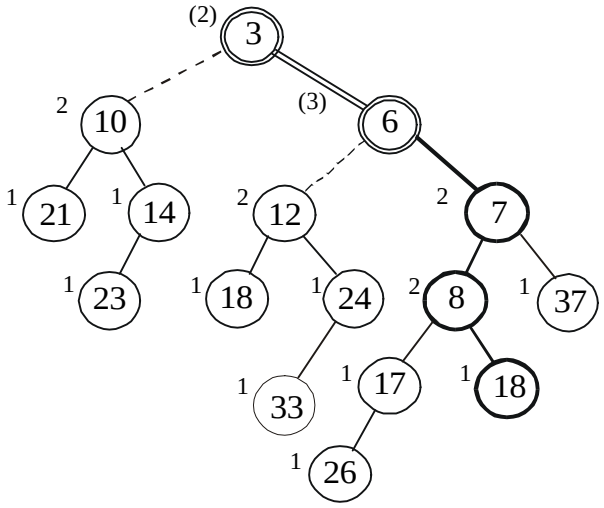

Рис. 5.

Далее рассматриваем узел с ключом 6. Оба его сына имеют одинаковый ранг 2, следовательно, менять их местами не требуется. Вычислим лишь новое значение ранга: оно равно минимальному из рангов его детей (рангу правого сына) плюс 1, то есть 3. Получаем дерево, изображенное на рисунке 6.

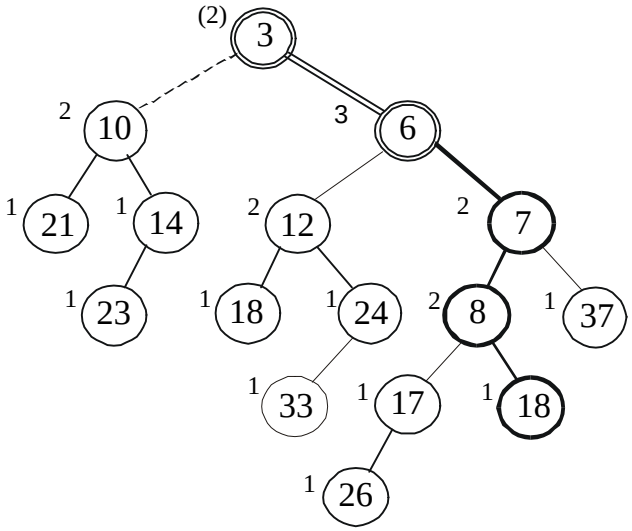

Рис. 6.

Наконец, рассматриваем узел с ключом 3, который является последним в правой ветви. Его потомков (узлы с ключами 10 и 6) необходимо поменять местами для восстановления свойства «левизны» и обновить ранг, который будет теперь равен 3. После выполнения этих операций получим левостороннюю кучу, изображенную на рисунке 7. На этом выполнение операции *СЛИЯНИЕ* заканчивается.

Очевидно, время выполнения операции пропорционально сумме длин правых путей сливаемых куч. По свойству левосторонней кучи оно не превосходит величины $log\, n_1 + log\, n_2 < log\, n + log\, n$, где n_1, n_2 – количества узлов в исходных кучах, а $n = n_1 + n_2$ – количество узлов в результирующей куче. Следовательно, вычислительная сложность операции *СЛИЯНИЕ* равна O($log\, n$).

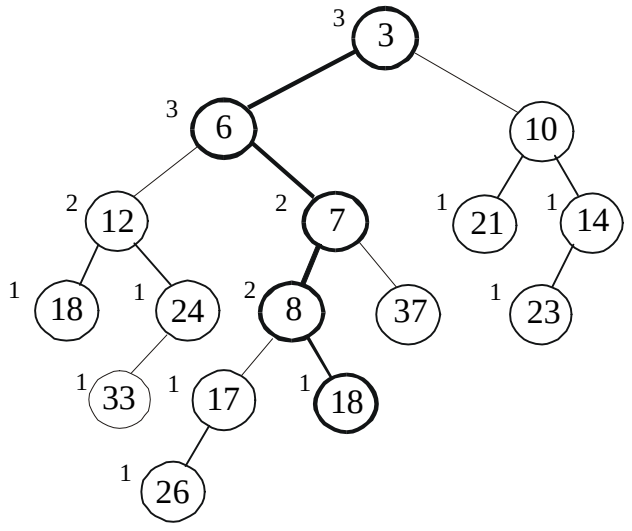

Рис. 7.

Реализация операции *СЛИЯНИЕ*

```
procedure СЛИЯНИЕ(h1, h2, h);
begin
    if h1 = nil then {h:= h2; exit}; if h2 = nil then {h:= h1; exit};
    if h1^.key > h2^.key then {h3:= h1; h1:= h2; h2:= h3;}
    h:= h1;
    СЛИЯНИЕ(h1^. right, h2, h3);
    h^ right:= h3;
    if h^left^.rank < h^right^rank then {h3:= h^left; h^.left:= h^right; h^right:= h3};
    h^rank:= min(h^right^rank, h^left^rank) + 1;
end;
```

Операция *ВСТАВКА*

Эта операция позволяет осуществить вставку в кучу *h* нового элемента *x* с ключом *k*. Она производится посредством образования левосторонней кучи, содержащей единственный элемент *x* с ключом *k*, и слияния ее с кучей *h*. Вычислительную сложность вставки нового элемента в приоритетную очередь можно оценить так же, как вычислительную сложность операции *СЛИЯНИЕ*, то есть величиной O(*log n*).

Реализация операции *ВСТАВКА*

```
procedure ВСТАВКА(x, k, h);
begin
    СОЗДАТЬ_УЗЕЛ h1: [element, key, rank, left, right, parent] = [x, k, 1, nil, nil, nil];
    СЛИЯНИЕ(h, h1, h2);  h:= h2
end;
```

Операция УДАЛЕНИЕ_МИНИМУМА

Эта операция позволяет из кучи *h* удалить элемент *Xmin* с минимальным ключом. Она производится посредством слияния его левой и правой подкуч. Трудоемкость операции - O(*log·n*).

Реализация операции УДАЛЕНИЕ_МИНИНИМУМА

```
procedure УДАЛЕНИЕ_МИНИМУМА(h, Xmin);
begin
  Xmin := h^.element; СЛИЯНИЕ(h^.left, h^.right, h3); h := h3
end;
```

Операция *МИНИМУМ*

Эта операция позволяет взять из кучи h элемент с минимальным ключом, не удаляя его из кучи. Поскольку элемент с минимальным ключом находится в корне кучи, то требуется лишь скопировать его в нужное место. Вычислительная сложность операции - O(1).

Реализация операции *МИНИМУМ*

```
function МИНИМУМ(h); begin МИНИМУМ:= h^.element end;
```

Операция УДАЛЕНИЕ

Эта операция позволяет удалить из кучи h элемент x, расположенный в узле, заданном позицией *pos*. Удаление может быть проведено в несколько этапов.

5 Если узел x является корнем кучи h, то применяется операция *УДАЛЕНИЕ_МИНИМУМА* из кучи h. Иначе выполняются следующие действия.

6 От исходной кучи h отрывается подкуча h_2 с корнем в удаляемом узле x. Оставшаяся куча, для которой сохраняем обозначение h, не обязательно является левосторонней.

7 Затем узел x удаляется из кучи h_2, а его левая и правая подкучи сливаются в одну кучу h_2' (время выполнения, как доказано выше – O ($log\ n$)).

8 Куча h_2' делается таким же сыном узла p (p – родитель удаляемого узла x), каким являлся для нее узел x (левым или правым).

9 Наконец, в куче h восстанавливается свойство «левизны». Фактически этому свойству могут не удовлетворять только узлы, находящиеся на пути от p к корню кучи h. Длина этого пути в худшем случае может линейно зависеть от n. Можно доказать, что на самом деле необходимо проверить только первые не более, чем $\lfloor log(n + 1) \rfloor$ узлов на этом пути.

Таким образом, время выполнения операции *УДАЛЕНИЕ* – O($log\ n$).

Рассмотрим пример. Пусть из кучи h, изображенной на рисунке 8, необходимо удалить элемент x с ключом 9.

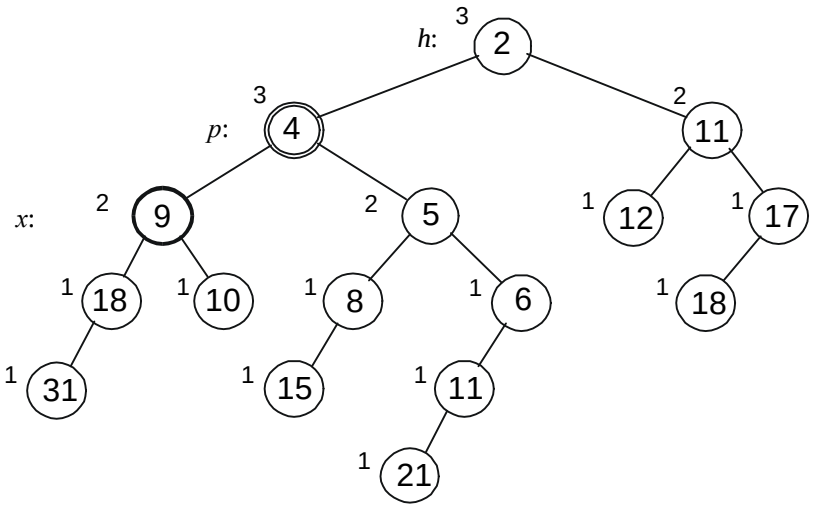

Рис. 8.

Сначала отрывается подкуча h_2 с корнем удаляемом узле x. Из кучи h получаются куча h_1 (не левосторонняя, так как свойству «левизны» не удовлетворяет узел p) и левосторонняя куча h_2, см. рисунок 9.

Затем удаляется узел x, а его левая и правая подкучи h_{2L} и h_{2R} сливаются в одну кучу h_2' с помощью операции *СЛИЯНИЕ*, см. рисунок 10.

Если бы удаляемый узел x был корнем в исходной куче h, то на этом операция была бы завершена, но поскольку в нашем случае узел x не являлся корнем кучи h, кучу $h2'$ делаем левым поддеревом узла p, так как узел x был его левым потомком, см. рисунок 11.

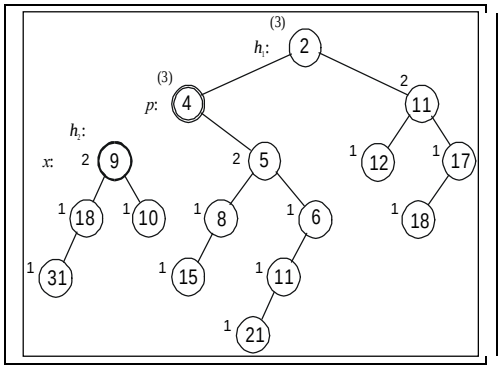
Рис. 9.

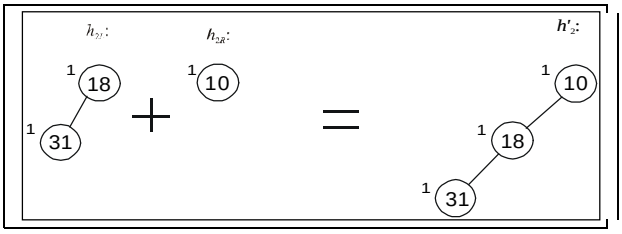
Рис. 10.

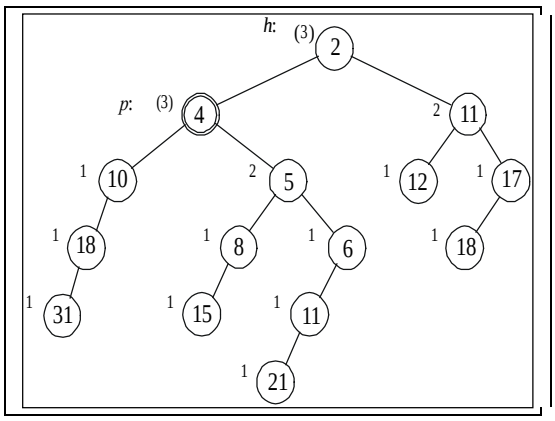

Рис. 11

Следуем от узла *p* к корню дерева и для каждого узла этого пути восстанавливаем свойство левизны и ранг, до тех пор не дойдем до узла, в котором ранг не потребует изменения или до корня,. Сначала проверяем узел *p*: его детей надо поменять местами, так как ранг узла с ключом 10 (он равен 1) меньше ранга узла с ключом 5 (он равен 2). После этого обновляется ранг узла *p*: он равен рангу правого сына плюс 1, то есть 2. Получилось дерево, изображенное на рисунке 12.

Следующий узел на пути к корню – это родитель узла *p* с ключом равным 2. Ранги его потомков равны, значит менять их местами не нужно. Однако его собственный ранг, возможно, требует обновления. Проверяем: новое значение равно рангу его правого потомка плюс 1, то есть старому: 3. Поскольку узел с ключом 2 является корнем дерева, операция *УДАЛЕНИЕ* завершена. В результате получаем искомое дерево, изображенное на рисунке 12.

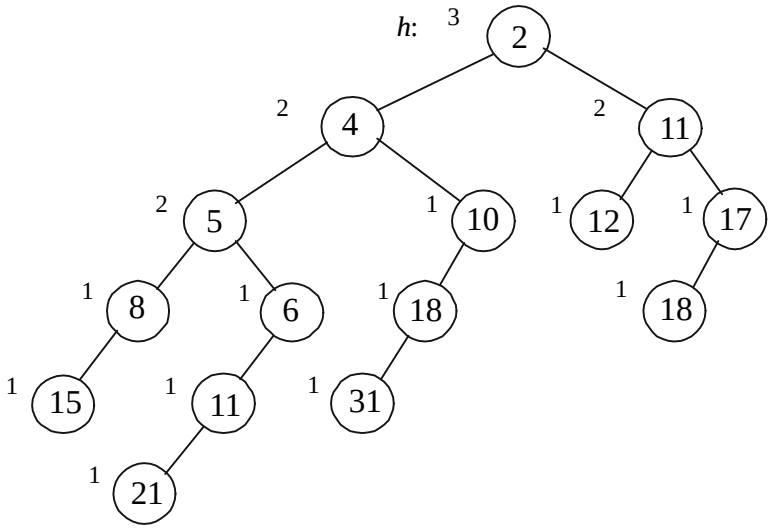

Рис. 12.

Операция УМЕНЬШИТЬ_КЛЮЧ

Чтобы уменьшить ключ указанного элемента, достаточно удалить его с помощью операции *УДАЛЕНИЕ*, а затем вставить его с уменьшенным ключом с помощью операции *ВСТАВКА*. Заметим, что если бы мы захотели уменьшить ключ и выполнить операцию *ВСПЛЫТИЕ*, то не получили бы логарифмическую оценку трудоемкости.

Операция ОБРАЗОВАТЬ_ОЧЕРЕДЬ

Из элементов списка S ($|S| = n$) образуется левосторонняя куча h. Способ формирования кучи из n элементов посредством n применений операции *ВСТАВКА* не эффективен. Читателю предоставляется возможность доказать, что в худшем случае формирование кучи таким способом может потребовать $O(n \log n)$ операций.

Более эффективным является следующий способ образования n-элементной левосторонней кучи. Заводится список Q, в который помещаются n одноэлементных куч. Пока длина списка Q больше 1, из его начала извлекаются две кучи, производится их слияние, а полученная куча вставляется в конец списка Q. Читателю предоставляется возможность доказать, что время образования приоритетной очереди таким способом - $O(n)$.

Сводные данные о трудоемкости операций с левосторонними кучами

1. *СЛИТЬ*(h_1, h_2, h)	$O(\log n)$
2. *ВСТАВИТЬ*(x, h)	$O(\log n)$
3. *УДАЛИТЬ_МИН*(h, x)	$O(\log n)$
4. *МИН*(x, h)	$O(1)$
5. *УДАЛИТЬ*(x, h)	$O(\log n)$
6. *УМЕНЬШИТЬ_КЛЮЧ*(x, Δ, h)	$O(\log n)$
7. *ОБРАЗОВАТЬ_ОЧЕРЕДЬ*(q, h)	$O(n)$

Представление приоритетной очереди с помощью биномиальных деревьев

Биномиальная очередь – это приоритетная очередь, организованная на базе так называемых биномиальных деревьев. При этой реализации также возможно эффективное слияние очередей. Биномиальные очереди интересны тем, что на их базе конструируются более эффективные реализации приоритетных очередей. В частности реализация с помощью фибоначчиевых куч, которая до недавнего времени считалась наиболее эффективной.

Для каждого $k = 0, 1, 2,\ldots$ биномиальное дерево B_k определяется следующим образом: B_0 – одноузловое дерево высоты 0; далее при $k > 0$ дерево B_k высоты k формируется из двух экземпляров дерева B_{k-1} посредством присоединения корня одного из них в качестве потомка к корню другого.

Биномиальный лес это набор биномиальных деревьев, в котором любые два дерева имеют разные высоты.

Свойства биномиальных деревьев.

1. Дерево B_k состоит из корня с присоединенными к нему корнями поддеревьев B_{k-1}, ... , B_1, B_0 в указанном порядке.
2. Дерево B_k имеет высоту k.
3. Дерево B_k имеет ровно 2^k узлов;
4. В дереве B_k на глубине i имеется ровно C_k^i узлов.
5. В дереве B_k корень имеет степень k, остальные узлы имеют меньшую степень.
6. Для каждого натурального числа n существует биномиальный лес, в котором суммарное количество узлов равно n.
7. Максимальная степень вершины в биномиальном лесе с n узлами равна $\log_2 n$
8. Биномиальный лес содержит не более $\log_2 n$ биномиальных поддеревьев

Чтобы убедиться в существовании биномиального леса из n узлов, представим n в двоичной системе счисления (разложим по степеням двойки) $n = a_0 \cdot 2^0 + a_1 \cdot 2^1 + ... + a_s \cdot 2^s$, где $a_k \in \{0, 1\}$. Для каждого $k = 0, 1, 2, \ldots, s$, такого, что $a_k = 1$, в искомый лес включаем дерево B_k

Биномиальная куча это набор биномиальных деревьев, узлам которых приписаны элементы взвешенного множества в соответствии с кучеобразным порядком. (Вес элемента приписанного узлу не превосходит весов элементов приписанных его потомкам.)

Поскольку количество детей у узлов варьируется в широких пределах, ссылка на детей осуществляется через левого потомка, а остальные потомки образуют односвязный список. Каждый узел в биномиальной куче представляется набором полей

[*key*, *parent*, *child*, *sibling*, *degree*]

4 *key* – ключ (вес) элемента, приписанного узлу,

5 *parent* – ссылка на родителя узла,

6 *child* – ссылка на левого потомка узла,

7 *sibling* – ссылка на правого брата узла,

8 *degree* – степень узла (число потомков узла)

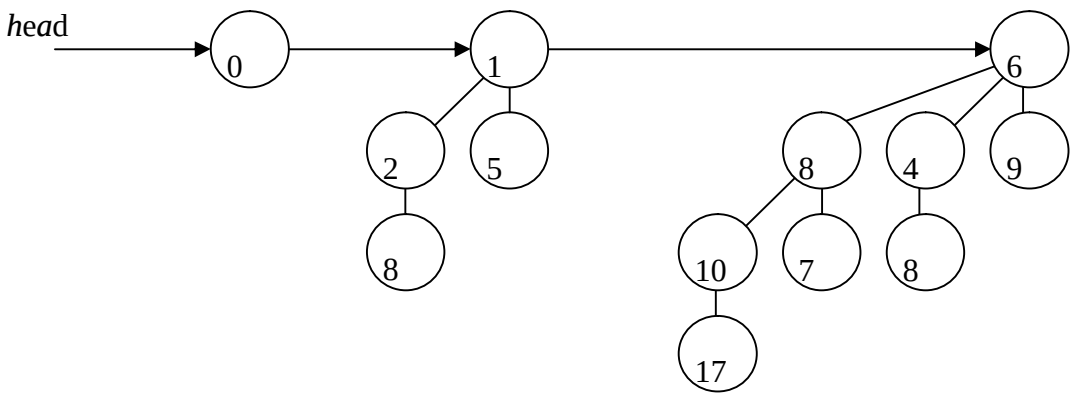

Рис. 13

Корни деревьев, из которых составлена куча, оказываются организованными с помощью поля *sibling* в, так называемый, корневой односвязный список, в порядке возрастания степеней. Доступ к куче осуществляется ссылкой на самое левое поддерево. На рисунке 13 изображена биномиальная куча, построенная на основе трех биномиальных деревьев B_0, B_2, B_3, с общим числом узлов равным 13. В кружочках записаны ключи соответствующих элементов.

Поиск элемента с минимальным ключом. Поскольку искомый элемент находится в корне одного из деревьев, а корневой список содержит не более $log_2 n$ элементов, то элемент с минимальным ключом находится просмотром корневого списка за время O($log\ n$).

Слияние двух очередей. Две очереди $H1$ и $H2$ объединяются в одну очередь H следующим образом. Последовательно вынимаются деревья из исходных очередей $H1$ и $H2$ в порядке возрастания их высот и вставляются в результирующую очередь H (вначале пустую).

Если дерево B_i очередной высоты i присутствует лишь в одной из исходных очередей, то перемещаем его в результирующую очередь.

Если оно присутствует в одной из исходных очередей и уже есть в результирующей очереди, то объединяем эти деревья в одно B_{i+1}, которое вставляем в H.

Если B_i присутствует во всех трех очередях, то одно из них перемещаем в H, а из двух других строим B_{i+1}, которое также вставляем в H.

Трудоемкость – O($log_2 n$).

Вставка нового элемента. Создается одноэлементная биномиальная очередь, которая объединяется с исходной. Трудоемкость – O($log_2 n$).

Удаление минимального элемента. Сначала в исходной куче H производится поиск дерева B_k, имеющего корень с минимальным ключом. Найденное дерево удаляется из H,

его корневые поддеревья B_{k-1}, ..., B_1, B_0 организуют в новую очередь $H1$, которая сливается с исходной очередью H. Возвращаем элемент – бывший корень дерева B_k. Трудоемкость – $O(log_2 n)$.

Уменьшение ключа. Осуществляется с помощью всплытия. Трудоемкость – $O(log_2 n)$.

Удаление элемента. Уменьшается ключ до минус бесконечности, производится всплытие и удаление как минимального элемента. Трудоемкость – $O(log_2 n)$.

ГЛАВА 6. РАЗДЕЛЕННЫЕ МНОЖЕСТВА

Основные определения

Разделенные множества (*disjoint sets*) – это абстрактный тип данных, предназначенный для представления коллекции, состоящей из некоторого числа k попарно непересекающихся (разделенных) подмножеств $U_1, U_2, ..., U_k$ заданного множества U. Для простоты в качестве U будем рассматривать множество $\{1, 2, \ldots, n\}$.

Этот тип данных применяется в таких задачах, как отыскание минимального остовного дерева для заданного взвешенного неориентированного графа (алгоритм Краскала), построение компонент связности графа, минимизация конечного автомата и многих других, требующих динамического поддержания некоторого отношения эквивалентности. Примеры таких задач будут рассмотрены ниже.

Как правило, в таких задачах вычисления начинаются с пустой коллекции подмножеств ($k = 0$). Затем по мере вычислений формируются новые подмножества, включаемые в коллекцию. Формирование новых подмножеств осуществляется либо путем создания одноэлементного подмножества, либо путем объединения уже существующих в коллекции подмножеств. Для осуществления таких действий вводятся в рассмотрение имена включенных в коллекцию подмножеств. В качестве имени подмножества мы будем использовать один из его элементов (канонический элемент), выбираемый по определенному правилу. Поскольку в коллекции всегда будут находиться попарно непересекающиеся подмножества множества U, такое имя будет однозначно определять требуемое подмножество.

Операции над разделенными множествами

***СОЗДАТЬ*(*x*).** Эта операция предназначена для введения в коллекцию нового подмножества, состоящего из одного элемента x, при этом предполагается, что x не входит ни в одно из подмножеств коллекции, созданной к моменту выполнения этой операции. Элемент x указывается в качестве параметра. Именем созданного подмножества будет считаться сам элемент x.

***ОБЪЕДИНИТЬ*(*x, y*).** С помощью этой операции можно объединить два подмножества коллекции, имеющие, соответственно, имена x и y, в одно новое подмножество, при этом оба объединяемые подмножества удаляются из коллекции, а вновь построенное подмножество получает некоторое имя. Во всех рассматриваемых нами случаях именем нового полученного в результате этой операции подмножества будет одно из имен x или y. Имена объединяемых подмножеств указываются в качестве параметров.

***НАЙТИ*(*x*, *y*).** Эта операция позволяет определить имя *y* того подмножества коллекции, которому принадлежит элемент *x*. Другими словами, с помощью этой операции находится канонический элемент *y* того подмножества коллекции, которому принадлежит элемент *x*. Если элемент *x* до выполнения операции не входил ни в одно из подмножеств коллекции, то в качестве *y* берется 0.

Последовательность s, составленную из операций типа *СОЗДАТЬ*, *ОБЪЕДИНИТЬ*, *НАЙТИ*, назовем *корректной*, если перед выполнением каждой операции из последовательности s выполнены условия ее применения. Например, перед выполнением очередной операции вида *ОБЪЕДИНИТЬ*(x, y) подмножества с именами x и y должны быть уже созданы. Перед выполнением операции *СОЗДАТЬ*(x), элемент x не должен принадлежать ни одному из подмножеств коллекции. Операция *НАЙТИ*(x, y) применима при любом значении аргумента $x \in U$. Следует только помнить, что если x не принадлежит ни одному из подмножеств коллекции, то получим $y = 0$.

Мы рассмотрим несколько способов представления коллекции разделенных множеств в памяти компьютера и алгоритмической реализации перечисленных операций. А именно, будут рассмотрены представления

1 с помощью массива,

2 с помощью древовидной структуры,

3 с помощью древовидной структуры с использованием рангов вершин,

4 с помощью древовидной структуры с использованием рангов вершин и сжатия путей.

Последний из перечисленных способов является наиболее эффективным по времени выполнения произвольных корректных последовательностей операций типа *СОЗДАТЬ*, *ОБЪЕДИНИТЬ*, *НАЙТИ*. Строго говоря, во всех перечисленных случаях будут использоваться массивы, но интерпретации их содержимого будут различными. Каждый раз при описании очередной реализации мы будем обсуждать трудоемкость рассматриваемых операций.

Примеры использования разделенных множеств

Пример 1. Рассмотрим задачу выделения компонент связности неориентированного графа. Напомним, что компонентой связности называется максимальное по включению подмножество вершин графа такое, что любые две его вершины связаны цепью. Полагаем, что вершины графа пронумерованы числами 1, 2,..., n и каждое ребро представлено парой (i, j) номеров вершин. Предполагаем также, что множество ребер не пусто.

Алгоритм выделения компонент связности неориентированного графа

7 Создать коллекцию из n одноэлементных подмножеств множества $\{1, 2, ..., n\}$;

8 Прочитать очередное ребро (i, j);

9 Найти имя a подмножества коллекции, содержащего элемент i;

10 Найти имя b подмножества коллекции, содержащего элемент j;

11 Если $a \neq b$, то объединить подмножества коллекции с именами a и b;

12 Если есть еще непрочитанные ребра, перейти к пункту 2, в противном случае закончить вычисления.

Очевидно, построенные подмножества коллекции будут представлять искомые компоненты связности исходного графа.

Используя введенные выше названия основных операций над коллекцией разделенных множеств, представленный выше алгоритм можно записать в следующем виде (операции, не относящиеся непосредственно к работе с коллекцией, записываем так же, как они были записаны выше).

4. **For** i:= 1 **to** n **do** *СОЗДАТЬ*(i);
5. Прочитать очередное ребро (i, j);
6. *НАЙТИ*(i, a);
7. *НАЙТИ*(j, b);
8. **if** $a \neq b$ **then** *ОБЪЕДИНИТЬ*(a, b);
9. Если есть еще непрочитанные ребра, перейти к пункту 2, в противном случае закончить вычисления.

Пример 2. Рассмотрим неориентированный связный граф без петель, ребрам которого приписаны в качестве весов вещественные числа. Требуется построить остовное дерево, накрывающее все вершины графа и имеющее минимальный суммарный вес входящих в него ребер. Итак, пусть заданный граф G имеет множество V вершин, пронумерованных числами 1, 2,..., n, и множество E ребер. Каждому ребру e из множества E поставлена в соответствие пара $(N(e), K(e))$ его концевых вершин и число $C(e)$ – его вес. Для решения этой задачи были предложены различные алгоритмы. Мы рассмотрим алгоритм Крускала.

Алгоритм Крускала

1 Создать коллекцию из n одноэлементных подмножеств множества {1, 2 ..., n};

2 Создать пустое множество T;

3 В множестве E найти ребро e с минимальным весом и удалить его из множества E;

4 Найти имя a подмножества коллекции, содержащего элемент $N(e)$;

5 Найти имя b подмножества коллекции, содержащего элемент $K(e)$;

6 Если $a \neq b$, то объединить подмножества коллекции с именами a и b, а ребро e добавить к множеству T;

7 Если множество E не пусто и $|T| < n$-1, перейти к пункту 3, в противном случае закончить вычисления.

Заметим, что в процессе работы алгоритма в множестве T будут находиться ребра, составляющие ациклический подграф исходного графа, являющийся лесом, состоящим из некоторого числа деревьев. Отсутствие циклов гарантируется проверкой "*Если* $a \neq b$" в пункте 6 описанного алгоритма. Фактически при $a \neq b$ происходит объединение двух поддеревьев в одно дерево с помощью ребра e, найденного на шаге 3 описанного алгоритма и имеющего концы, находящиеся в разных поддеревьях.

Если исходный граф связен, как это сказано в постановке задачи, то построенное с помощью такого алгоритма множество T будет, очевидно, представлять дерево, накрывающее все вершины исходного графа. Доказательство того факта, что суммарный вес входящих в него ребер будет минимальным, можно найти в работе [1].

Как видим из описания этого алгоритма, в нем естественным образом используется структура разделенных множеств. Обратим внимание на операцию поиска во множестве E ребра e с минимальным весом. Эффективность этой операции существенно зависит от выбора структуры данных для хранения множества E. Приемы эффективного выполнения таких операций будут рассмотрены в разделе "Приоритетная очередь".

Представление разделенных множеств с помощью массива

Пусть $U = \{1, 2, \ldots, n\}$ – множество, из элементов которого будем строить коллекцию разделенных подмножеств.

Одним из простых способов представления коллекции является представление ее с помощью массива. Для каждого элемента i в соответствующей (i-ой) ячейке массива будет

находиться имя (канонический элемент) того подмножества, которому принадлежит элемент *i*. Если элемент *i* не принадлежит ни одному из подмножеств коллекции, то в *i*-ой ячейке массива будет находиться нуль.

Обозначим через *f* массив, с помощью которого будем представлять подмножества коллекции. Если в самом начале вычислений в коллекцию не входит ни одно множество, то для ее представления необходим массив длины *n*, заполненный нулями. Для его достаточно выполнить следующий оператор цикла

```
for i:= 1 to n do f[i]:= 0;
```

Операции над множествами, реализованными с помощью массива

***СОЗДАТЬ*(x).**

В ячейку массива *f*, имеющую номер *x*, записываем *x*. Таким образом, время выполнения данной операции есть константа.

```
procedure СОЗДАТЬ(x); begin f[x]:= x end;
```

***ОБЪЕДИНИТЬ*(x, y)**

Просматриваем элементы массива *f* и в те ячейки, в которых было записано имя *x*, заносим новое имя – *y*. Таким образом, именем вновь образованного подмножества будет *y*, а *x* перестанет быть именем какого-либо подмножества. Очевидно, время выполнения этой операции ограничено сверху величиной пропорциональной длине *n* массива *f*.

```
Procedure ОБЪЕДИНИТЬ(x, y); begin for z:= 1 to n do if f[z] = x then f[z]:= y end;
```

***НАЙТИ*(x, y)**

В качестве *y* берем содержимое элемента с номером *x* в массиве *f*. Очевидно, время выполнения данной операции есть константа.

```
Procedure НАЙТИ(x, y); begin y:= f[x] end;
```

При такой реализации время выполнения *m* произвольных операций есть величина $O(m \cdot n)$.

Действительно, время выполнения *m* операций *СОЗДАТЬ*, так же как и *НАЙТИ*, очевидно, есть $O(m)$, так как время выполнения одной такой операции есть константа.

Время выполнения *m* операций *ОБЪЕДИНИТЬ* есть $O(m \cdot n)$, так как время выполнения одной такой операции есть $O(n)$. Итак, время выполнения *m* произвольных операций есть $O(m \cdot n)$.

Представление разделенных множеств с помощью древовидной структуры

Пусть, по-прежнему, $U = \{1, 2,\ldots, n\}$ – множество, из элементов которого будет строиться коллекция.

Каждое подмножество коллекции представляется корневым деревом, узлы которого являются элементами этого подмножества, то есть отождествляются с номерами из множества $1, 2,\ldots, n$. Корень дерева используется в качестве имени соответствующего подмножества (канонический элемент). Для каждого узла дерева определяется узел $p(x)$, являющийся его родителем в дереве; если x – корень, то полагаем $p(x) = x$.

Фактически в памяти компьютера это дерево будем представлять массивом $p[1..n]$ так, что $p(x)$ будет предком узла x, если x не является корнем, и $p(x) = x$ если x – корень. Если же x не входит ни в одно из подмножеств коллекции, то $p(x) = 0$.

Рассмотрим пример. Пусть $U = \{1, 2,\ldots, 7\}$ и коллекция состоит из двух подмножеств $\{1, 2, 3, 7\}$ и $\{4, 6\}$. Деревья, представляющие эти подмножества, могут быть такими как на рисунке 1.

Кружочки обозначают узлы дерева; указатели на родителей представлены при помощи стрелок. Именем одного из этих подмножеств является 3, другого – 6:

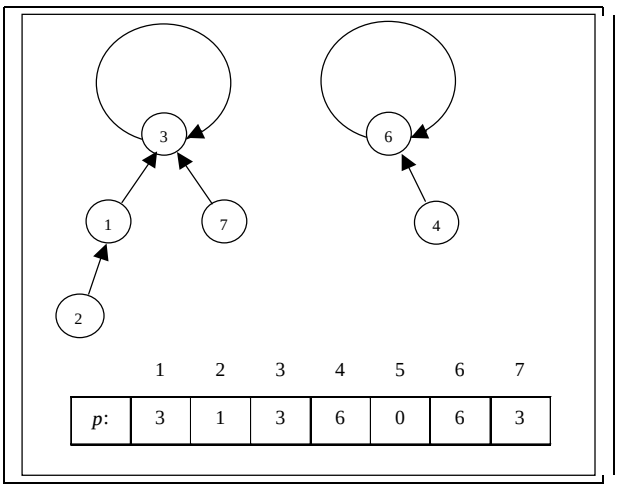

Рис. 1.

Операции над множествами, реализованными с помощью деревьев

***СОЗДАТЬ*(*x*)**

В качестве родителя узла x назначаем сам узел x. Таким образом, время выполнения данной операции есть константа. В результате выполнения операции СОЗДАТЬ(x) образуется новое одновершинное дерево с петлей в корне.

procedure *СОЗДАТЬ*(*x*); **begin** *p*[*x*]:= *x* **end**;

Если к коллекции подмножеств, изображенных на рисунке 1 применить операцию СОЗДАТЬ(5), то получим коллекцию, изображенную на рисунке 2.

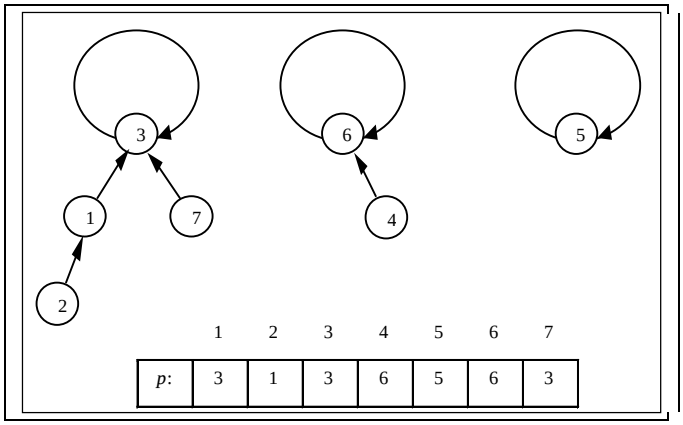

Рис. 2.

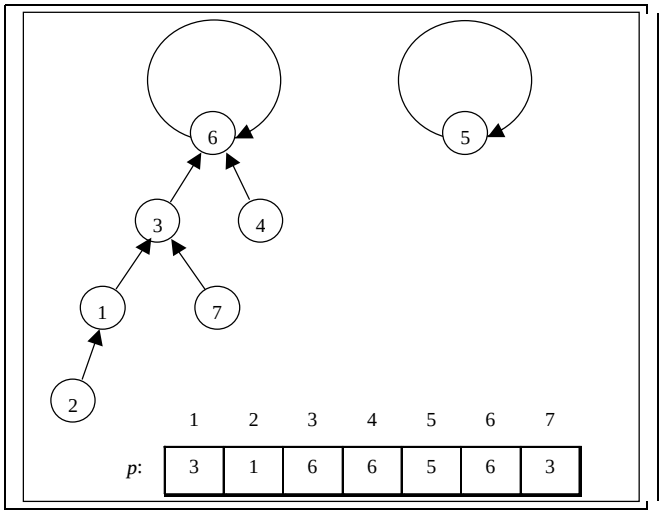

Рис. 3.

***ОБЪЕДИНИТЬ*(*x, y*)**

Делаем узел *y* родителем узла *x*. Заметим, что *x* и *y* до выполнения рассматриваемой операции должны быть корнями соответствующих деревьев. Именем вновь образованного подмножества будет y, а *x* перестанет быть именем какого-либо множества. Очевидно, время выполнения этой операции есть константа.

Если применить операцию *ОБЪЕДИНИТЬ*(3, 6) к коллекции, изображенной на рисунке 3, то получим коллекцию, состоящую из двух подмножеств {1, 2, 3, 4, 6, 7} и {5}, изображенную на рисунке 3. Именем первого из этих подмножеств будет 6, второго – 5).

procedure *ОБЪЕДИНИТЬ*(*x*, y); **begin** *p*[*x*]:= y **end**;

***НАЙТИ*(*x, y*)**

Продвигаемся по указателям на родителей от узла *x* до корня дерева и в качестве *y* берем этот корень.

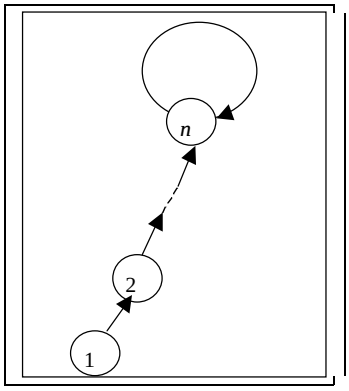

Рис. 4.

Очевидно, время выполнения операции пропорционально длине пути из узла x в корень соответствующего дерева. Из реализации операции *СОЗДАТЬ* и *ОБЪЕДИНИТЬ* видно, что возможно образование дерева в виде линейной цепочки узлов, изображенной на рисунке 4. Эта цепочка получилась в результате применения следующей последовательности операций:

```
СОЗДАТЬ(1);
СОЗДАТЬ(2);
…
СОЗДАТЬ(n);
ОБЪЕДИНИТЬ(1, 2);
ОБЪЕДИНИТЬ(2, 3);
…
ОБЪЕДИНИТЬ(n-1, n);
```

Худший случай применения операции *НАЙТИ* в данной ситуации – это *НАЙТИ*(1, y). В этом случае необходимо сделать $n - 1$ переход по ссылкам на родителей, чтобы дойти от узла 1 к корню дерева n, и один переход, чтобы узнать, что родитель узла n есть сам узел n. Таким образом, время выполнения операции *НАЙТИ* есть величина O(n).

```
procedure НАЙТИ(x, y); begin while p[x] ≠ x do x:= p[x]; y:= x end;
```

Если операция *СОЗДАТЬ* выполнялась n раз, то последовательность, из m операций *ОБЪЕДИНИТЬ* и/или *НАЙТИ* при рассматриваемой реализации разделенных множеств выполняется за время O($m \cdot n$). Действительно, время выполнения m операций *ОБЪЕДИНИТЬ*, очевидно, есть O(m), так как время выполнения одной такой операции есть константа. Время выполнения m операций *НАЙТИ* есть O($m \cdot n$), так как время выполнения одной такой операции есть O(n). Итак, время выполнения m произвольных операций есть O($m \cdot n$).

Представление множеств с помощью древовидной структуры с использованием рангов вершин

Предыдущую реализацию разделенных множеств можно усовершенствовать следующим образом. Операцию *ОБЪЕДИНИТЬ* можно выполнять так, чтобы высота результирующего дерева, была как можно меньше. А именно, корень большего по высоте дерева сделать родителем корня другого дерева.

Назовем такую реализацию операции *ОБЪЕДИНИТЬ* объединением по рангу. В качестве ранга в данном случае берется высота соответствующего дерева. Для такой реализации разделенных множеств необходимо хранить с каждым узлом x дополнительно еще одну величину – высоту поддерева, корнем которого является узел x. Будем называть ее высотой, или рангом, узла x. Остальные операции нужно настроить на корректную работу с этим полем. Будем хранить высоту каждого узла x в ячейке $h[x]$ массива h.

Описание операций над множествами, реализованными с помощью древовидной структуры с использованием рангов вершин

***СОЗДАТЬ*(*x*)**

В качестве родителя узла x берем тот же самый x, а высотой узла x считаем 0. Таким образом, время выполнения данной операции есть константа. В результате выполнения операции *СОЗДАТЬ*(x) образуется новое дерево, состоящее из одного узла.

```
procedure СОЗДАТЬ(x);   begin p[x] := x; h[x] := 0 end;
```

ОБЪЕДИНИТЬ(x, y)

Корень большего по высоте дерева становится родителем корня другого дерева. Если деревья имеют одинаковую высоту, то узел y становится родителем узла x, после чего увеличивается ранг узла y на единицу. Именем вновь образованного подмножества будет имя того из объединяемых подмножеств, у которого корень имел большую высоту, а имя другого из объединяемых подмножеств перестанет быть именем какого-либо из подмножеств. Очевидно, время выполнения этой операции есть константа.

```
procedure ОБЪЕДИНИТЬ(x, y);
begin
  if (h[x] < h[y]) then p[x]:= y
                   else if (h[x] > h[y]) then p[y]:= x
                   else {p[x]:= y; h[y]:= h[y] + 1}
end;
```

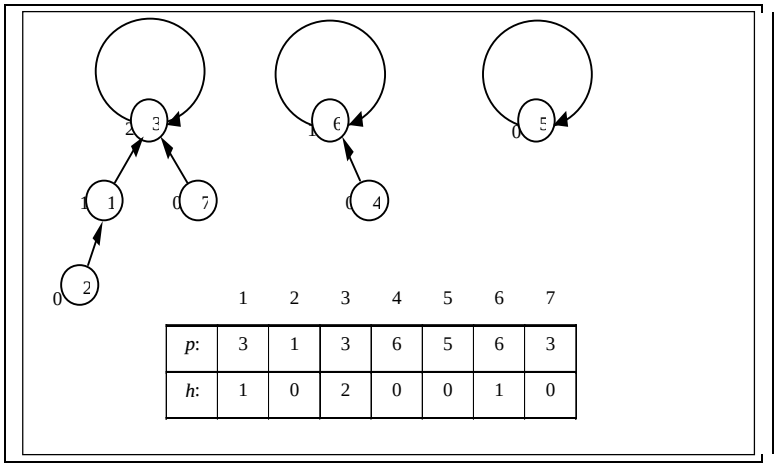

Рис. 5.

На рисунке 6 показан результат применения операции *ОБЪЕДИНИТЬ*(3, 6) к коллекции, изображенной на рисунке 5, с учетом высот объединяемых поддеревьев. Рядом с кружочками, изображающими узлы, показаны их высоты. Так как $h(3) = 2 > h(6) = 1$, то родителем узла 6 становится узел 3.

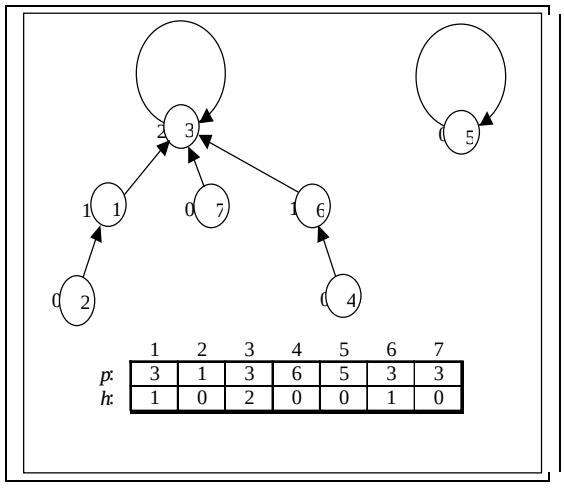

Рис. 6.

***НАЙТИ*(*x*, *y*)**

Продвигаемся по указателям на родителей от узла x до корня дерева и в качестве *y* берем найденный корень. Например, результатом применения операции *НАЙТИ*(4, *y*) к коллекции, изображенной на рисунке 8, будет $y = 3$.

Очевидно, время выполнения данной операции пропорционально длине пути из узла x в корень соответствующего дерева. Для оценки длины этого пути докажем следующие леммы.

Лемма 1. В результате выполнения любой последовательности операций типа *СОЗДАТЬ*, *ОБЪЕДИНИТЬ*, *НАЙТИ* над пустой коллекцией разделенных множеств для

любого узла x выполняется неравенство $n[x] \geq 2^{h(x)}$, где $n[x]$ – количество узлов в поддереве с корнем x, $h[x]$ – высота узла x.

Доказательство. Очевидно, перед первым применением операции *ОБЪЕДИНИТЬ* для любого узла x имели $n[x] = 1$, $h[x] = 0$ и, следовательно, $n[x] \geq 2^{h(x)}$. Операции СОЗДАТЬ, *НАЙТИ* не могут нарушить доказываемое неравенство, поэтому доказательство можно провести индукцией по количеству применений операции *ОБЪЕДИНИТЬ*.

Предположим, что перед очередным применением операции *ОБЪЕДИНИТЬ*(x, y) доказываемое неравенство выполнено, тогда если высота узла x меньше высоты узла y, то дерево, полученное с помощью *ОБЪЕДИНИТЬ*(x, y), имеет корень y, а высоты узлов x и y не изменились. Количество узлов в дереве с корнем x не изменилось, а количество узлов в дереве с корнем y увеличилось. Таким образом, как для узлов x, y, так и для всех остальных неравенство сохраняется. Случай, когда высота узла x больше высоты узла y, аналогичен предыдущему.

Если же высоты деревьев с корнями x и y до выполнения операции были одинаковы ($h[x] = h[y] = h$), то узел y становится родителем узла x, высота узла y увеличивается на 1, а высота узла x не изменяется. Пусть после выполнения операции величины $h[x]$, $h[y]$, $n[x]$, $n[y]$ становятся равными соответственно $h'[x]$, $h'[y]$, $n'[x]$, $n'[y]$, тогда имеем $h'[y] = h[y] + 1$, $h'[x] = h[x]$, $n'[x] = n[x]$, $n'[y] = n[y] + n[x]$. По предположению индукции, имеем $n[y] \geq 2^{h[y]}$, и $n[x] \geq 2^{h[x]}$. Следовательно, после выполнения рассматриваемой операции для узлов x и y имеем соотношения $n'[x] = n[x] \geq 2^{h[x]} = 2^{h'[x]}$ и $n'[y] = n[y] + n[x] \geq 2^{h[y]} + 2^{h[x]} = 2^{h+1} = 2^{h'[y]}$ Таким образом, утверждение леммы остается верным и в этом случае.

Лемма 2. Если за время работы, начавшейся с пустой коллекции, операция СОЗДАТЬ применялась n раз, то для любого $h \geq 0$ число k узлов высоты h удовлетворяет неравенству $k \leq n/2^h$.

Доказательство. Пусть x_1, x_2,..., x_k – все узлы высоты h, тогда по Лемме 1 при $i = 1, 2, \ldots, k$ справедливы неравенства $n[x_i] \geq 2^h$. Следовательно,

$$n \geq n[x_1] + n[x_2] + \ldots + n[x_k] \geq k \cdot 2^h,$$

откуда и следует требуемое неравенство $k \leq n / 2^h$.

Следствие. В результате выполнения любой последовательности операций типа *СОЗДАТЬ, ОБЪЕДИНИТЬ, НАЙТИ* над пустой коллекцией разделенных множеств, для любого узла x имеет место неравенство $h[x] \leq \log n$.

Доказательство. Дерево максимальной высоты образуется, очевидно, лишь тогда, когда все n элементов объединяются в одно множество. Для такого дерева количество k

узлов максимальной высоты h равно 1, по лемме 2 имеем $1 = k \le n / 2^h$, откуда $2^h \le n$ и, следовательно, $h \le \log n$.

Таким образом, время выполнения операции *НАЙТИ* есть O($\log n$).

```
Procedure НАЙТИ(x, y); begin while p[x] ≠ x do x := p[x]; y := x end;
```

При такой реализации разделенных множеств если количество узлов равно n, то выполнение m операций *ОБЪЕДИНИТЬ* и/или *НАЙТИ* займет время равное O($m \cdot \log n$).

Замечание. При объединении подмножеств в качестве ранга узла x можно использовать количество узлов в поддереве с корнем в узле x. В этом случае Лемма 1 для высот узлов остаетсясправедливой, а, значит, сохраняются и оценки времени выполнения операций.

Представление множеств с помощью древовидной структуры с использованием рангов вершин и сжатия путей

Предыдущий способ реализации разделенных множеств можно еще улучшить за счет усовершенствования реализации операции *НАЙТИ*(x, y). Она теперь будет выполняться в два прохода. При первом проходе находится корень у того дерева, которому принадлежит x. При втором проходе из x в y все встреченные узлы делаются непосредственными потомками узла y. Этот прием, как увидим ниже, намного уменьшает среднее время выполнения последующих операций *НАЙТИ*.

Рассматриваемая реализация не требует новых полей данных по сравнению с предыдущим случаем. Как и прежде, для каждого узла i будем хранить указатель $p[i]$ на его родителя и ранг $r[i]$, который теперь не обязательно будет равен высоте дерева с корнем i. Он будет равен этой высоте, если из последовательности применяемых операций удалить все операции *НАЙТИ*.

Описание операций над множествами с использованием рангов вершин и сжатия путей

***СОЗДАТЬ*(*x*)**

В качестве родителя узла x берем тот же самый x, а его высотой считаем 0. Таким образом, время выполнения данной операции есть константа.

```
procedure СОЗДАТЬ(x); begin p[x] := x; r[x] := 0 end;
```

***ОБЪЕДИНИТЬ*(*x*, *y*)**

При выполнении этой операции действия аналогичны действиям из предыдущей реализации, разница лишь в том, что вместо массива *h* используется массив *r*. Время выполнения операции – константа.

```
procedure ОБЪЕДИНИТЬ(x, y);
begin
  if (r[x] < r[y])   then p[x]:= y
                     else if (r[x] > r[y]) then p[y]:= x
                     else {p[x]:= y; r[y]:= r[y] + 1}
end;
```

***НАЙТИ*(*x*, *y*)**

Операция, как уже говорилось, выполняется в два прохода. При первом проходе мы идем от узла *x* к его родителю, потом к родителю его родителя и так далее, пока не достигнем корня *y* дерева, содержащего узел *x*. При втором проходе из *x* в *y* все встреченные на этом пути узлы делаются непосредственными потомками узла *y*. Будем называть это "сжатием путей". Очевидно, как и раньше, время выполнения одной такой операции есть O(*log n*). Но ниже будет доказано, что время выполнения *m* таких операций на самом деле меньше, чем O(*m*·*log n*).

```
procedure НАЙТИ(x, y);
begin
  z := x;
  while (p[x] ≠ x) do x:= p[x];
  y := x;
  while (p[z] ≠ z) do { z′ := z; z:= p[z]; p[z′]:= y}
end;
```

Для анализа временных затрат на выполнение операций нам потребуются две функции. Одна из них, *b*(*n*), является суперэкспонентой и определяется следующим образом:

$$b(0) = 0,\ b(n) = 2^{b(n-1)} \text{ для } n > 0.$$

Вторая – суперлогарифм *log* k,* по основанию 2 определяемая соотношением:

$$log^* k = min\{\, n: n \geq 0,\ log^{(n)} k \leq 0\},$$

где $log^{(1)} k = log\ k$, $log^{(n+1)} k = log\ (log^{(n)} k)$ для $\forall n > 0$.

Супер логарифм является обратной функцией к суперэкспоненте. Значения функций *log*k* и *b*(*n*) при нескольких значениях аргументов приведены в следующих таблицах.

n	0	1	2	3	4	5	…	16	17	…	65536	65537	…	2^{65536}
log^*n	0	1	2	3	3	4	…	4	5	…	5	6	…	6

n	0	1	2	3	4	5	6
$b(n)$	0	1	2	4	16	65536	2^{65536}

Теорема. Время выполнения последовательности операций, состоящей из n операций *СОЗДАТЬ*, $u \le n - 1$ операций *ОБЪЕДИНИТЬ* и f операций *НАЙТИ*, является величиной $O((f+n)\ log^*(u + 1))$.

Сводные данные о сложности операций с разделенными множествами

Реализация с помощью массива

СОЗДАТЬ(x)	$O(1)$
ОБЪЕДИНИТЬ(x, y)	$O(n)$
НАЙТИ(x, y)	$O(1)$
m операций	$O(mn)$

РЕАЛИЗАЦИЯ С ПОМОЩЬЮ ДРЕВОВИДНОЙ СТРУКТУРЫ

СОЗДАТЬ(x)	$O(1)$
ОБЪЕДИНИТЬ(x, y)	$O(1)$
НАЙТИ(x, y)	$O(n)$
m операций	$O(mn)$

РЕАЛИЗАЦИЯ С ПОМОЩЬЮ ДРЕВОВИДНОЙ СТРУКТУРЫ С ИСПОЛЬЗОВАНИЕМ РАНГОВ ВЕРШИН

СОЗДАТЬ(x)	$O(1)$
ОБЪЕДИНИТЬ(x, y)	$O(1)$
НАЙТИ(x, y)	$O(\log n)$
m операций	$O(m \log n)$

РЕАЛИЗАЦИЯ С ПОМОЩЬЮ ДРЕВОВИДНОЙ СТРУКТУРЫ С ИСПОЛЬЗОВАНИЕМ РАНГОВ И СЖАТИЯ ПУТЕЙ

СОЗДАТЬ(x)	$O(1)$
ОБЪЕДИНИТЬ(x, y)	$O(1)$
НАЙТИ(x, y)	$O(\log n)$
m операций	$O(n \lg^*(u + 1))$ $O(m\ a(m, n))$

Примечание. Р.Е. Тарьян доказал, что время выполнения последовательности, состоящей из u операций *ОБЪЕДИНИТЬ* в перемешку с f операциями НАЙТИ, где $u \le n - 1$, $u + f = m$, является величиной $O(m \cdot \alpha(m, n))$. Также он показал, что эта оценка не может быть улучшена, а, значит, алгоритм может потребовать для своего выполнения $\Omega(m \cdot \alpha(m, n))$ времени.

Здесь $\alpha(f, n) = min\{i \ge 1 : A(i, \lfloor f / n \rfloor) > log\ n\}$, где $A(i, j)$ – функция Аккермана, задаваемая равенствами:

$A(1,\ j) = 2j$ при $j \ge 1$,

$A(i, 1) = A(i - 1, 2)$ при $i \geq 2$,

$A(i, j) = A(i - 1, A(i, j - 1))$ при $i, \ j \geq 2$.

ГЛАВА 7. НЕТРАДИЦИОННЫЕ СИСТЕМЫ СЧИСЛЕНИЯ

Системой счисления принято называть совокупность приемов для представления чисел и выполнения операций над ними. Основные требования, предъявляемые к системе счисления это

1) возможность представления любого числа из заранее заданного диапазона,

2) простота оперирования с числами.

Наиболее распространенными системами на данный момент являются так называемые позиционные системы счисления. В человеческом общении используется десятичная, а в технических системах принято считать, что, как правило, используется двоичная. Однако появляющиеся со временем более жесткие требования к представлению и обработке чисел приводят к разработке новых нетрадиционных способов работы с ними. Такими требованиями могут быть существенное увеличение диапазона представляемых чисел, увеличение точности приближенных вычислений, увеличение скорости выполнения операций. Могут быть и более специфические требования, связанные с ускорением работы конкретных алгоритмов, например, алгоритмов используемых в криптографии.

Реализация нетрадиционных систем может идти по двум направлениям. Первый путь это программный, связанный с разработкой структур данных и алгоритмов для использования в типовых компьютерах. Второй связан с разработкой физических устройств, как правило, электронных, предназначенных для новых способов обработки чисел.

Среди систем реализуемых программным путем можно отметить так называемую систему остаточных классов и фибоначчиеву систему. Первая из них имеет недостатки из-за отсутствия эффективного алгоритма деления, вторая отличается хитроумностью операций. Среди систем, которые реализуются в настоящее время, как говорят, в «железе», можно отметить так называемые избыточные позиционные системы счисления. Следует отметить, что на заре развития вычислительной техники избыточность была крайне нежелательным свойством, поскольку приводила к существенному увеличению размеров вычислительных, устройств и к их удорожанию. В связи с развитием микроэлектроники, появилась возможность использовать избыточность для распараллеливания алгоритмов. В следующем разделе мы рассмотрим так называемые избыточные знакоразрядные системы счисления, которые в настоящее время используются в современных компьютерах.

Знакоразрядные системы счисления

Это позиционные системы счисления, определяемые некоторым натуральным параметром r ($r \geq 2$) – основанием системы и набором цифр. Цифрами в такой системе являются символы со значениями

$$-a, -a+1, \ldots, -1, 0, 1, \ldots, a,$$

где $a=[r/2]$. В общем случае a можно выбирать из условия $=[r/2] \leq a \leq r-1$.

Используя параметр r и набор цифр, любое рациональное число x можно представить,

$$x = \sum_{i=n}^{-m} x_i \cdot r^i$$

вообще говоря, неоднозначно в виде

$$x = x_n \cdot r^n + x_{n-1} \cdot r^{n-1} + \ldots + x_1 \cdot r^1 + x_0 \cdot r^0 + x_{-1} \cdot r^{-1} + \ldots + x_{-m+1} \cdot r^{-m+1} + x_{-m} \cdot r^{-m},$$

где $x_i \in \{-a, -a+1, \ldots, -1, 0, 1, \ldots, a\}$, $(i = -m, -m+1, \ldots, -1, 0, 1, \ldots, n-1, n)$.

Обычно число x в такой системе представляется последовательностью цифр

$$x_n, x_{n-1}, \ldots, x_1, x_0, x_{-1}, \ldots, x_{-m+1}, x_{-m},$$

В этой последовательности запятые опускаются, кроме той, которая отделяет первые $n+1$ разрядов от остальных. Первые $n+1$ разрядов представляют целую часть числа x, а остальные дробную.

Пусть, например $r = 3$ и $a = 1$, тогда цифры будут иметь значения -1, 0, 1. Цифру -1 будем записывать как подчеркнутую единицу $\underline{1}$. Последовательность ($\underline{1}$ 1 0, 1 0 $\underline{1}$ $\underline{1}$) будет представлять число $(-1)\cdot 3^2+1\cdot 3^1+0\cdot 3^0+1\cdot 3^{-1}+0\cdot 3^{-2}+(-1)\cdot 3^{-3}+(-1)\cdot 3^{-4} = -463/81$.

Рассмотрим варианты знакоразрядных систем для работы с целыми числами. (Авизинис, 1961 г.) В обычной позиционной системе счисления алгоритм сложения двух n-разрядных чисел выполняется за $O(n)$ шагов из-за возможного распространения переносов вплоть до старшего разряда.

Основным достоинством рассматриваемой ниже системы счисления является то, что перенос при выполнении операции сложения распространяется не далее соседнего разряда и время выполнения операции при параллельном исполнении не зависит от количества разрядов.

Рассматриваемая ниже система определяется двумя параметрами: b – основание системы, a – максимальная цифра. Используются цифры $\{-a, -(a-1),\ldots,-1, 0, 1,\ldots, -(a-1), a\}$. Систему с параметрами b, a обозначаем $D(b, a)$.

Теорема. $(\forall b \geq 2)(\forall a \in [(b-1)/2, (b-1)])$ любое натуральное число d можно представить в виде

$$d = \sum_{i=0}^{n-1} d_i b^i$$

где $d_i \in [-a, a]$ – цифра. Такое представление неоднозначно.

Пример 1. В знакоразрядной системе счисления с параметрами $b = 10$, $a = 6$ запись $1\underline{5}31\underline{2}0$ представляет число 0-20+100+3000-50000+100000 =52080.

Замечание 1. Если $1 \le a < (b\text{-}1)/2$, то не все числа $x \in Z$ можно представить в такой системе счисления.

Пример 2. Пусть $b = 10$, $a = 4$. Пара (b, a) недопустима, ($a \notin [(b–1)/2, b\text{-}1]$). Число 5 не представимо.

Пример 3. Пусть $b = 10$, $a = 5$. Пара (b, a) допустима, так как $a \in [(b\text{-}1)/2\ ,\ b–1]$. Число 15725 представленное здесь в обычной десятичной системе может быть представлено в системе $D(b, a) = D(10, 5)$ последовательностями цифр $(2\,\underline{4}\,\underline{3}\,2\,5)$ и $(2\,\underline{4}\,\underline{3}\,3\,\underline{5})$, поскольку

$15725=2\cdot10^4+(-4)\cdot10^3+(-3)\cdot10^2+2\cdot10+5\cdot1 = 2\cdot10^4+(-4)\cdot10^3+(-3)\cdot10^2+3\cdot10+(-5)\cdot1$

Пример 4. , если $b = 2k$, $a = k$, то число k можно представить в виде (a) и $(1\ \underline{a})_s$.

Замечание 2. Если b – нечетно и $a = {}^{(b-1)}/_2$, то система $D(b, a)$ не избыточна.

Знакоразрядная система называется системой Авизиниса, если ее параметры удовлетворяют условию $b \ge 3$, $a \in [(b+1)/2, (b\text{-}1)]$.

Алгоритм сложения чисел в системе Авизиниса.

Пусть x и y – два числа, представленные в системе Авизиниса соответственно записями

$$x_{n-1},x_{n-2},\ldots,x_0,$$

$$y_{n-1},y_{n-2},\ldots,y_0.$$

Алгоритм строит запись $z_n,z_{n-1},z_{n-2},\ldots,z_0$ числа $z = x+y$. В алгоритме используется переменная $t[0..n]$ для обозначения переносов в следующий разряд. Переносы принимают значения +1, 0 и –1.

1). t_0:= 0; z_n:= 0;
2). **for** i:= 0 **to** n-1**do** || z_i:= x_i+y_i;
3). **for** i:= 0 **to** n-1**do** || **if** $z_i > (a\text{-}1)$ t*hen* {z_i:= z_i-b; t_{i+1}:= 1} **else**
if $z_i < -(a\text{-}1)$ t*hen* {z_i:= z_i+b; t_{i+1}:= 1} **else**
t_{i+1}:= 0;
4). **for** i:= 1 **to** n **do** || z_i:= z_i+ t_i

Символ || в записи алгоритма означает параллельное выполнение тела цикла. Время работы алгоритма при параллельном исполнении операторов в теле цикла t*cnm* O(1).

Пример. Выполним сложение $1\,\underline{5}\,3\,1\,\underline{2}\,0 + 1\,\underline{1}\,\underline{2}\,6\,1\,\underline{6}$.

$1\,\underline{5}\,3\,1\,\underline{2}\,0 – x_i$

$1\,\underline{1}\,\underline{2}\,6\,1\,\underline{6} – y_i$

$2\,\underline{6}\,1\,7\,\underline{1}\,\underline{6} – z_i$ – после выполнения оператора 2

0 1 0 1 0 1 0 – t_i - после выполнения оператора 3

0 2 4 1 3 1 4 – z_i – после выполнения оператора 3

0 1 4 2 3 2 4 – z_i – после выполнения оператора 4

Заметим, что если у всех цифр числа y поменять знак, то получится число $-y$. Поэтому алгоритм сложения чисел можно использовать и для вычитания.

Возможные значения параметра a в зависимости от r, приведены в таблице.

r	Возможные значения параметра $a \in \{\frac{r-1}{2}, r\text{-}1\}$ в знакоразрядной системе счисления	Значения параметра $a \in \{\frac{r+1}{2}, r\text{-}1\}$, для которых справедлива теорема Авизиниса
2	1	–
3	1, 2	2
4	2, 3	3
5	2, 3, 4	3, 4
8	4, 5, 6, 7	5, 6, 7
10	5, 6, 7, 8, 9	6, 7, 8, 9
16	8, 9, A, B, C, D, E, F	9, A, B, C, D, E, F

Система счисления, допускающая инкрементацию цифры за время O(1)

Определение. Избыточным b-арным представлением неотрицательного целого числа x считаем теперь последовательность $d = d_n, d_{n-1} \ldots, d_0$, где $d_i \in \{0, 1, .., b\}$, $i \in \{0, 1, \ldots, n\}$, такую, что $x = \sum_{i=0}^{n} d_i b^i$. Называем d_i цифрой стоящей в i-ом разряде.

В примерах запятые между цифрами будем опускать, а в случаях, когда $b = 10$, «цифру» 10 будем обозначать символом b. Заметим, что избыточное представление отличается от обычного b-арного представления использованием «лишней» цифры b, что приводит к неоднозначности представления чисел. Например, при $b = 3$, число 3, может быть представлено как 3 и как 10.

Определение. Назовем избыточное b-арное представление числа x регулярным, если в нем между любыми двумя цифрами, равными b, находится цифра, отличная от b–1.

Определение. Операция фиксации цифры b, стоящей в i-ом разряде b-арного регулярного представления числа заключается в обнулении цифры d_i и инкрементировании цифры d_{i+1}. Считаем, что операция фиксации, примененная к разряду не равному b равносильна пустой операции.

Очевидно, фиксация цифры $d_i = b$ в регулярном b-арном представлении числа, при $b > 2$, дает новое регулярное b-арное представление того же числа.

Инкрементирование цифры в регулярном представлении с трудоемкостью O(1).

Пусть $j[i]$ – номер разряда ближайшего слева к i-му, и не равного b-1.

Пример. Пусть избыточное регулярное представление имеет вид $d = b990b2$. Тогда $j[i]$ представлено в следующей таблице.

i	5	4	3	2	1	0
$D[i]$	b	9	9	0	b	2
$j[i]$		5	5	5	2	1

Чтобы обеспечить требуемую трудоемкость, представим избыточное регулярное представление двумя массивами D и L, в которых $D[i]$ – значение i-го разряда, а $L[i]$ – указатель, удовлетворяющий при любом i следующему условию (инварианту)

$$\{D[j[i]] = b \Rightarrow L[i] = j[i]\} \;\&\; \{D[j[i]] < b\text{-}1 \Rightarrow L[i] > i\}. \qquad (1)$$

Фиксация i-го разряда.

```
Procedure Fix(i);
begin if (D[i] ≠ b) then exit;
  D[i]:= 0; D[i+1]:= D[i+1] + 1;
  if (D[i+1] = b-1) then L[i]:= L[i+1] else L[i]:= i+1
end;
```

Нетрудно видеть, что фиксация разряда имеет трудоёмкость O(1) и не нарушает инвариант. Используя эту операцию, инкрементацию разряда можно осуществить следующей процедурой.

```
Procedure Inc(i);
begin
  Fix(i); Fix(L[i]); D[i]:= D[i]+1; Fix(i); Fix(L[i]);
end;
```

Заметим, что предварительная фиксация разрядов i и $L[i]$ в этой процедуре гарантирует возможность инкрементации i-го разряда и обеспечивает регулярность (после проведения операции) выражения на участке от i-го и до самого старшего разряда.

Заключительная фиксация разрядов i и $L[i]$ гарантирует после операции неравенство $D[i] \neq b$ и, следовательно, регулярность выражения на участке от i-го и до самого младшего разряда, при этом значение ближайшего допустимого слева от i-го разряда (после операции) не равно b. А это означает, что нет необходимости обновлять массив L для разрядов младше i-го.

Теорема. Операция инкрементирования i-го разряда регулярного b-арного избыточного представления d числа x, производит регулярное b-арное избыточное представление d' числа $x' = x+b^i$.

Доказательство оставляем читателю.

Избыточные системы счисления с основанием 2

Рассмотрим одну из таких систем, но сначала вспомним схему обычного двоичного сумматора. Для его конструирования можно использовать ячейки типа FA (Full-Adder), изображенные на рисунке 1. Каждая такая ячейка имеет 3 двоичных входа x, y, z и два выхода t и u, где x и y – биты очередных разрядов слагаемых, z – вход для приема переноса из предыдущего разряда, t - перенос в следующий разряд, u – очередной бит результата.

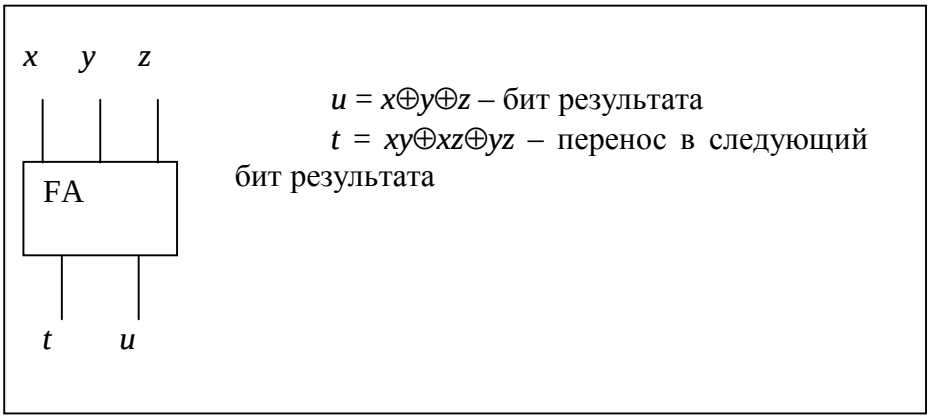

Рис. 1

Схема сумматора, построенного из ячеек такого вида, изображена на рисунке 2.

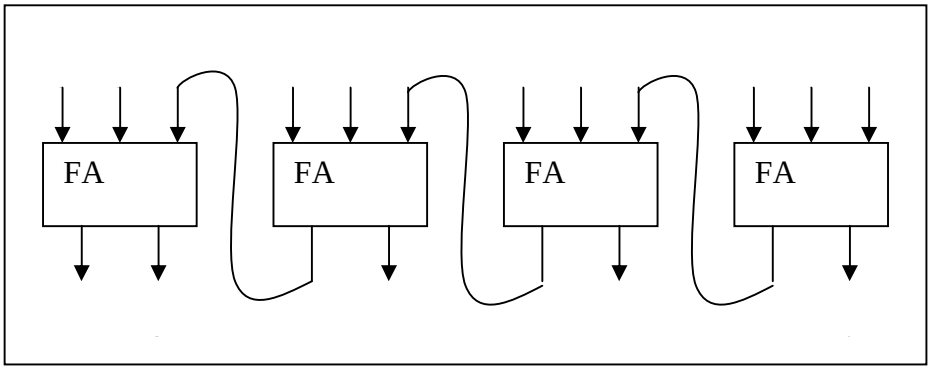

Рис. 2

Система счисления, сохраняющая переносы (CS-система, carry-save number system).

В этой системе используются три цифры $d \in \{0,1,2\}$. Каждая цифра d представляется двумя битами $(d(2), d(1))$, так, что $d = d(2)+d(1)$. Заметим, что цифра 1 может быть представлена неоднозначно: как парой (0,1), так и парой (1,0).

Схема сумматора для сложения CS-числа *a*, каждая цифра которого задана двумя битами с обычным двоичным числом *b*, каждая цифра которого задана одним битом, изображена на рисунке 3. Результатом сложения является число *s*, заданное двумя битами в CS-системе:

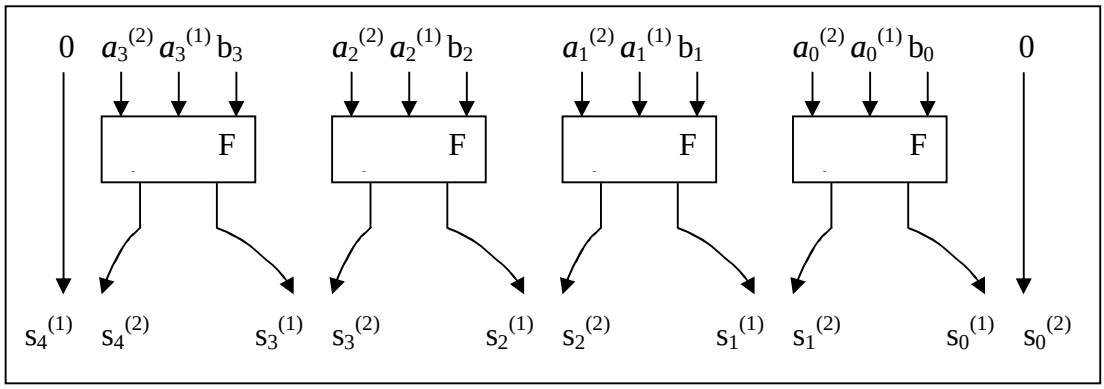

Рис. 3

В таком сумматоре перенос сразу прибавляется к следующему разряду. Чтобы таким сумматором за два такта сложить два CS-числа с двух битовыми цифрами можно сначала к *a* прибавить первые биты числа *b*, а потом к полученному результату прибавить его вторые биты.

Избыточные системы счисления используются внутри арифметических операторов, таких как умножители и делители. Входные и выходные данные таких операторов представляются в не избыточной системе счисления, а внутренние вычисления выполняются в избыточных системах. Большинство умножителей используют хотя бы условно системы счисления с сохранением переноса, в то время как делитель в *Pentium* использует две различные системы счисления. Операции деления выполняются в системе CS, а частное сначала генерируется в CS-системе с 4 цифрами между - 2 и +2, потом конвертируются в двоичную CS-систему.

ЛИТЕРАТУРА

1. В.Е. Алексеев, В.А. Таланов Графы и алгоритмы. Структуры данных. Модели вычислений. Интернет-Университет Информационных Технологий. Москва, 2006.

2. В.Е. Алексеев, В.А. Таланов Графы. Модели вычислений. Структуры данных. Нижний Новгород, Издательство ННГУ, 2005.

3. Т. Кормен, Ч. Лейзерсон, Р. Ривест Алгоритмы. Построение и анализ. МЦНМО, Москва, 1999.

4. В. Липский Комбинаторика для программистов. Москва, «Мир», 1988.

ПРИЛОЖЕНИЕ

Приводится текст программы на языке PASCAL, демонстрирующий использование приоритетной очереди реализованной с помощью d-кучи в алгоритме нахождения кратчайших путей в графе.

Процедура **GraphGenerator** с помощью датчика псевдослучайных чисел генерирует координаты вершин для будущего графа и запоминает их в массивах x, y: array[vertex] of integer. Для каждой пары вершин вычисляется евклидово расстояние между ними и если это расстояние меньше заданной величины delta_ro, то считается, что есть ребро между данными вершинами. Построенный граф запоминается в виде массива adj:array[vertex] of ptrAdjBox, в котором adj[i] это указатель на список элементов тепа

AdjBox = record v:vertex; w:weigh; next:ptrAdjBox **end**;

v – номер очередной вершины смежной с вершиной i, w – вес ребра (i, v), next – указатель на следующий AdjBox.

Процедура **GraphDraw** рисует граф на экране.

Процедура **Dijkstra** вычисляет кратчайшие расстояния от заданной стартовой вершины s и с помощью процедуры **TreeDraw** рисует дерево кратчайших путей от вершины s.

```
uses graphABC, VCL, crt, utils;
  const gz=maxint;  delta_ro=100; ngr=50; d=3;
  type  vertex = 1..ngr;
        node = 0..ngr;
        weigh = real;
        ptrAdjBox=^AdjBox;
        AdjBox = record v:vertex; w:Weigh; next:ptrAdjBox end;
  var   adj:array[vertex] of ptrAdjBox;
        up: array[vertex] of vertex;
        dist: array[vertex] of weigh;
        start: vertex;
        x, y: array[vertex] of integer;
//-------------------------------------------------------------------------------------------------------------
function ro(i, j: vertex): weigh; begin ro:= sqrt(sqr(x[i]-x[j]) + sqr(y[i]-y[j])) end;
//-------------------------------------------------------------------------------------------------------------
procedure GraphGenerator;  var i, j: vertex; tmp: ptrAdjBox; roij: weigh;
begin  SetWindowSize(600,400); start:= 1+random(ngr);
  for i:=1 to ngr do begin x[i]:= random(WindowWidth); y[i]:= random(WindowHeight) end;
  for i:=1 to ngr do adj[i]:= nil;
  for i:=1 to ngr-1 do  for j:= i+1 to ngr do
  begin roij:= ro(i,j);
    if (roij < delta_ro) then
      begin
        new(tmp); tmp^.v:= j; tmp^.w:= roij; tmp^.next:= adj[i]; adj[i]:= tmp;
        new(tmp); tmp^.v:= i; tmp^.w:= roij; tmp^.next:= adj[j]; adj[j]:= tmp;
      end
  end
end;
//-------------------------------------------------------------------------------------------------------------
procedure GraphDraw;  var i:vertex; tmp: ptrAdjBox;
begin  SetPenWidth(1);
  for i:=1 to ngr do
```

```
     begin tmp:= adj[i];
       while tmp <> nil do begin Line(x[i], y[i], x[tmp^.v], y[tmp^.v]); tmp:= tmp^.next end;
     end;
end;
//-------------------------------------------------------------------------------------------------------------------
procedure TreeDraw;  var i, j: vertex;
   begin SetPenWidth(2); SetPenColor(clBlue);
    for i:= 1 to ngr do if i <> start  then
    begin  if up[i]= i then continue; j:= i;
       while j <> start do begin Line(x[j], y[j], x[up[j]], y[up[j]]); j:= up[j] end;
    end;
    for i:=1 to ngr do begin circle(x[i], y[i], 3) end;
    SetPenColor(clRed); SetPenWidth(4); circle(x[start], y[start], 5);
   end;
//----------------------------Dijkstra: Global adj:array[1..maxver] of ptr_box;--------------------------
procedure Dijkstra;
var r, r0, v: vertex; delta: weigh; size, j: node;
     H: array[node] of vertex;  //куча
     map: array[vertex] of node;
     ptr: ptrAdjBox;
//-------------------------------------------------------------------------------------------------------------------
procedure TR(i,j:node);  var tmpV: vertex; tmpN: node;
   begin tmpV:=H[i]; H[i]:=H[j]; H[j]:=tmpV;
          tmpN:= map[H[i]]; map[H[i]]:= map[H[j]]; map[H[j]]:= tmpN
   end;
//-------------------------------------------------------------------------------------------------------------------
procedure EMERSION(i: node);var parent: node;
begin
    while i>0 do
    begin parent:= (i-1) div d;
       if dist[H[parent]] > dist[H[i]] then begin TR(i, parent); i:= parent end else exit
    end
end;
//-------------------------------------------------------------------------------------------------------------------
function MINCHILD(i: node): node; var first, last, j:integer; minkey: Weigh;
begin first:=i*d+1; if first > size then begin minchild:= 0; exit end;
   last:= i*d+d; if last > size then last:= size;
   minkey:= gz; minchild:= 0;
   for j:= first to last do if dist[H[j]] < minkey then begin minkey:= dist[H[j]]; minchild:= j end
end;
//-------------------------------------------------------------------------------------------------------------------
procedure DIVING(i:node); var j: node;
begin repeat j:=MINCHILD(i); if j=0 then exit;
       if (dist[H[i]] > dist[H[j]]) then begin TR(i,j); i:=j end else exit;
      until false
end;
//-------------------------------------------------------------------------------------------------------------------
procedure EXTRACT_MIN(var r0: vertex);
begin r0:= H[0]; TR(0,size); size:= size-1; DIVING(0) end;
//-------------------------------------------------------------------------------------------------------------------
BEGIN  {Dijkstra}
  for v:=1 to ngr do begin dist[v]:= gz; up[v]:= v; map[v]:= v-1 end;
```

```
  for j:= 0 to ngr-1 do begin H[j]:= j+1 end;
  size:= ngr-1; dist[start]:= 0;  TR(0, map[start]);
  while size > 0 do
    begin EXTRACT_MIN(r0); ptr:= adj[r0];
      while ptr <> nil do
      begin r:= ptr^.v; delta:= dist[r]-(dist[r0]+ptr^.w); ptr:= ptr^.next;
        if delta > 0 then begin dist[r]:= dist[r]-delta; EMERSION(map[r]); up[r]:= r0 end
      end
    end
END; {Dijkstra}
//-------------------------------------------------------------------------------------------------------------
BEGIN  GraphGenerator; GraphDraw; Dijkstra; TreeDraw END
//-------------------------------------------------------------------------------------------------------------
```