Теоретическая информатика для биологов

А.А.Миронов

Оглавление

Предисловие

Предлагаемая Вашему вниманию книга основана на семестровом курсе лекций по теоретической информатике, который читается на факультете Биоинженерии и биоинформатики Московского Государственного университета им М. В. Ломоносова. Основной целью данного курса является стремление дать самые общие представления о теоретической информатике. Это необходимо, чтобы читать и понимать алгоритмические статьи по биоинформатике и не падать в обморок от пары формул, псевдокода или теорем. Поэтому в этой книге приведено достаточно много алгоритмов на псевдокоде. Есть леммы и теоремы — достаточно простые, поскольку доказательство, как правило, умещается в нескольких строчках. С другой стороны, иногда даются некоторые абстрактные понятия, такие, как полукольцо, язык, конечный автомат. Это представляется важным для того, чтобы у читателя возникла привычка если не оперировать абстракциями, то по крайней мере наблюдать за тем, как это делают другие.

При выборе тем и рассмотренных алгоритмов автор опирался на собственный опыт и представление о прекрасном. Возможно другой преподаватель при составлении плана лекций по теоретической информатике для биологов несколько изменил бы содержание, но, думаю, примерно 70% материала сохранилось бы, поскольку они являются базовыми и используются в биоинформатике.

В первой главе приведены некоторые основные понятия информатики и булевой алгебры, разбираются описываются способы записи алгоритмов. Замечу, что поскольку для записи псевдокода нет стандарта, то в книге использован синтаксис наиболее похожий на язык C или Java.

Вторая глава посвящена поиску, сортировке и некоторым основным структурам данных — списки, деревья, хеш-таблицы.

В третьей главе обсуждаются алгоритмы поиска образца в тексте. В конце главы даны основные представления о теории сложности А.Н.Колмогорова.

Четвертая глава посвящена алгоритмам на графах. Приведено несколько примеров того, как биологические задачи могут формулироваться на языке графов. Разобраны наиболее важные алгоритмы в этой области.

Наконец, пятая глава является наиболее абстрактной. В ней даются представления о теории вычислительной сложности. В книге есть некоторое количество задач, достаточно простых, чтобы не давать их решений.

Книга рассчитана на студентов и аспирантов, специализирующихся в биоинформатике и молекулярной биологии.

Автор приносит глубокую благодарность своим бывшим студентам Ольге Ганчаровой и Надежде Тухтумбаевой, которые взяли на себя труд записать и оцифровать конспект лекций. Именно эти конспекты легли в основу предлагаемой книги.

Глава 1

Некоторые предварительные сведения

Предлагаемая книга дает краткое и достаточно поверхностное введение в теоретическую информатику. Эта наука по-английски называется Computer Science. Зачем информатика биологу? Существенной частью практически любого современного исследования в области молекулярной биологии является биоинформатическое исследование, которое либо предваряет экспериментальную работу, либо ее завершает, хотя чаще всего биоинформатика применяется и до и после экспериментальной работы. Это связано в большой степени с тем, что современные работы в области молекулярной биологии опираются на большое количество разнообразных данных, таких как геномы, протеомы, экспрессионные данные и т.п. Анализ этих данных немыслим без современных методов компьютерного анализа. Разумеется, автор не предполагает, что биологи должны сами создавать алгоритмы анализа этих данных, писать программы и т.п. — для этого есть специалисты. Однако биологу придется взаимодействовать со специалистами в области информатики и программистами. Для этого необходимо иметь хотя бы общее представление о методах и подходах теоретической информатики. Кроме того, грамотный биолог читает научную литературу. А в ней часто встречаются работы, описывающие новые методы анализа. Можно привести ряд биологических журналов, в которых публикуются работы, посвященные методам компьютерного анализа данных, зачастую эти работы содержат также экспериментальную часть. Чтобы понимать эти статьи необходимо знакомство с языком информатики и основными идеями и методами.

С алгоритмами и/или программами вы встречаетесь постоянно, даже не замечая этого. Например, вы назначили встречу с Васей Пупкиным на факультете Биоинженерии и биоинформатики. Вы звоните по телефону и говорите примерно следующий текст. “Доедешь до метро Университет, выйдешь по указателям к Университету, перейдешь Ломоносовский проспект.

Если нет дождя, то пойдешь вдоль Ломоносовского проспекта. Пройдешь примерно два километра, далее мимо Биологического факультета, увидишь наш корпус. Если же идет дождь, то садись на любой автобус, кроме номеров 1, 47, 119, или на троллейбус 34. На транспорте проедешь 3 остановки и выйдешь на ул. Менделеева. На входе покажешь что-нибудь похожее на студ. билет. Если пропустят, то поднимайся на 4-й этаж, иначе звони мне, я проведу по своей карточке". В таком объяснении много неточностей и недоговорчнностей. Что здесь важно? Во-первых, вы предполагаете, что Вася Пупкин ориентируется в метро, умеет сесть в автобус, умеет считать до четырех, наконец, умеет ходить и читать. Теперь представьте, что вы объясняете, как доехать до факультета, вашей подруге из Швеции. Если она только что приехала в Москву, то ей надо еще объяснить, как дойти до метро, как туда войти, где, как и за сколько купить карточку, где и как сделать пересадку и т.п. Итак, при описании алгоритма предполагается, что существует исполнитель, и что исполнитель кое-что умеет. Определений для термина алгоритм существует довольно много. Вот лишь одно из них. Алгоритм — это точный однозначно понимаемый набор инструкций, описывающий порядок действий некоторого исполнителя для достижения конкретного результата за конечное время.

1.1 Компьютеры, программы и алгоритмы

Здесь мы будем рассматривать алгоритмы для компьютеров. Хотя компьютеры бывают разные (в мире есть не только пентиумы), все они обладают рядом общих свойств:

- Компьютеры имеют линейно организованную память, в которой можно запоминать промежуточные результаты. К элементам памяти можно обращаться по *адресам* — целым числам.
- Компьютеры имеют процессор (исполнитель), который умеет выполнять арифметические операции, операции сравнения, операции записи в память и извлечения данных из памяти.
- Процессор один. Вы слышали, конечно, о многоядерных процессорах и многопроцессорных кластерах. Наверняка у кого-то из вас есть дома двухядерный компьютер. Количество процессоров обычно заранее известно и ограничено, поэтому нет принципиальной разницы один процессор или несколько. Многопроцессорными вычислениями называются такие, что на каждый элемент задачи (атом в молекулярной динамике, вершина графа) выделен свой процессор. Многопроцессорные вычисления выходят за рамки этого курса.
- Процессор исполняет команды последовательно одну за другой.

Алгоритм должен за конечное время (пусть и достаточно большое) найти решение.

Следствие 1. *Для решения любой задачи используется конечная память (докажите).*

Хотя в любом реальном компьютере размер памяти вполне определен, мы будем считать, что память в компьютере не ограничена.

Важной особенностью алгоритма является то, что он имеет входные данные и производит результат. Для одного и того же алгоритма можно использовать разные входные данные, и он будет производить разные результаты. Даже алгоритм, вычисляющий значение числа π , имеет разные входные данные в зависимости от требуемой точности.

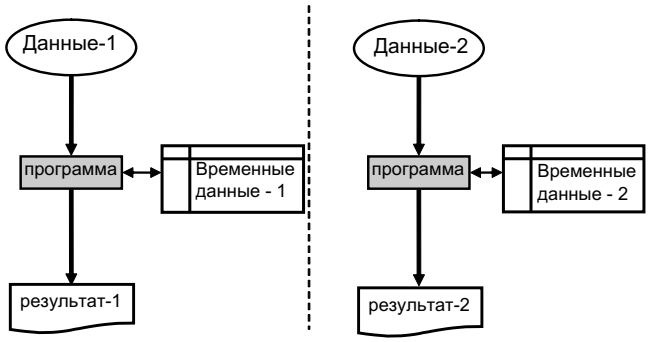


Рис. 1.1: Программа (алгоритм) принимает входные данные, в процессе обработки порождает временные данные и выдает результат.

Один и тот же алгоритм может быть реализован на компьютере в виде разных программ — это зависит от платформы компьютера, языка программирования, стиля программиста и т.п.

1.1.1 Время работы алгоритма

Важнейшим вопросом теоретической информатики является время работы того или иного алгоритма. Размер входной задачи (например, длина анализируемого слова или текста) может варьировать. Поскольку алгоритм принимает разные входные данные, то и время его работы может быть разным. Будем считать, что время работы алгоритма пропорционально числу выполняемых операций (сложений, сравнений и т.п.). На самом деле это не совсем правильно, поскольку в реальных компьютерах время исполнения разных команд может отличаться в десятки раз. Принято оценивать время работы алгоритма как функцию от размера задачи, причем важно его асимптотическое поведение (при больших размерах задачи). Поскольку реальное время зависит от деталей устройства процессора и от деталей

перевода кода в инструкции процессора, то время работы оценивается с точность до порядка. Вспомним курс математического анализа. Говорят, что величина T есть $O(f(L))$ (читается: О большое от $f(L)$), если существует такая константа C , что для любого допустимого L $T(L) \leq C \cdot f(L)$.

Время работы алгоритма может зависеть не только от размера задачи, но и от самих данных. Поэтому для времени работы алгоритма существуют оценки в худшем и в среднем. При оценке времени в худшем рассматривается такие данные (заданного размера), которые дадут самое большое количество операций. При оценке времени в среднем исходным данным заданного размера приписывается вероятность их реализации и оценивается математическое ожидание времени работы алгоритма. Говорят, что один алгоритм A_1 эффективнее алгоритма A_2, если $T_1(N)/T_2(N) \to 0$ при $N \to \infty$, где N — размер задачи . Алгоритмы используют память для хранения промежуточных данных. Количество используемой памяти также является важной характеристикой алгоритма.

1.2 Типы данных

В памяти компьютера хранятся данные, которые бывают разных типов.

1.2.1 Основные типы данных

Примитивные числовые типы данных — это байт, целое число со знаком, целое число без знака, число с плавающей точкой. Также к примитивным типам данных относятся boolean и char.

Байт (*byte*). Это целочисленный беззнаковый тип данных. Принимает значения от 128 до +127, всего 256 значений, которые можно записать с помощью двух шестнадцатеричных цифр (0123456789abcdef).

Целое число со знаком. Тип включает переменные, способные принимать целочисленные значения, положительные или отрицательные, и значение нуль. К этому типу относятся переменные вида *int* (занимает 4 байта, соответственно может принимать значения в пределах ± 2147483647), *long* (занимает 8 байтов, $\pm 9 \cdot 10^{18}$) и *short* (занимает 2 байта, принимает значения в пределах ± 32768).

Целое число без знака (*unsigned short*, *unsigned long*, *unsigned int*). переменные, аналогичные целочисленным переменным со знаком, но способные принимать только положительные значения и значение нуль. Беззнаковый int может принимать значения от 0 до $4 \cdot *10^9$.

Число с плавающей точкой (*double*, *float*) представление дробных чисел, кода число хранится в форме мантиссы и показателя степени. К примеру, число 0.02 представляется в виде двух чисел: мантиссы 2 и показателя степени 2. Как и целые числа, этот тип данных может

быть реализован с привлечением разных объемов памяти, что влияет на максимальное принимаемое значение. К примеру, в языках C и Java есть два типа чисел с плавающей точкой — *float* (занимает 4 байта) и *double* (занимает 8 байтов). *double* — 64-разрядный тип данных, при этом 54 разряда выделяется под мантиссу, 10 — под показатель степени. К точности чисел с плавающей точкой нужно относиться с осторожностью, особенно при сравнении двух чисел. Известно, что если поделить единицу на число 99999999999, а потом умножить на него же, и спросить Java, равен ли результат единице, получим ответ *false*, поскольку при округлении и переводе из двоичной в десятичную систему и обратно происходит снижение точности. В представлении *double* мантисса меняется в пределах от $+(2^{54})/2$ до $(2^{54})/2$, что соответствует примерно 15-ти десятичным знакам. Однако, существуют числа с длиной значащей части больше 15 знаков и даже иррациональные числа. Записав любое такое число в переменную типа *double*, мы неизбежно теряем точность, поскольку 16-ое и последующие значащие числа будут отброшены. Так что, если мы хотим сравнить два числа с плавающей точкой (a и b), лучше делать это с умом: взять какое-либо малое число (уровень значимости) и сравнивать разность ab с этим числом.

Булева переменная *boolean* может принимать значения "истина" или "ложь". Для реализации типа достаточно одного бита, и часто для экономии памяти удобно представлять данные в виде переменных типа *boolean*. Если у вас есть только два значения ($\{1,0\}$ или $\{+,-\}$) разумнее всего представлять их в виде *boolean*.

Символьный тип данных *char* хранит символы в формате Unicode и занимает до 16 битов (двух байтов).

1.2.2 Массивы

Часто используются массивы — это пронумерованные множества данных. К элементу массива можно обратиться с помощью индекса. Массивы лежат в памяти подряд. Следуя соглашениям в языке C и Java, мы будем считать, что первый элемент имеет индекс 0. Если у нас есть массив a, то элементом массива с номером i будет $a[i]$. Длину массива будем обозначать как a.length. Частным случаем массива является строка — массив из символов.

1.2.3 Указатели

Особое место в современном программировании занимают *указатели* — это адреса в памяти, которые указывают нам место, где хранятся данные. При этом сами данные могут быть размещены в самых разных местах памяти. На рис. 1.2 а) показан фрагмент памяти компьютера. Верхняя строка — адреса (они идут подряд). Нижняя строка — содержимое ячеек памяти (данные). Данные могут иметь самый разный характер — числа, сим-

волы (тоже представлены числами). В частности в данных могут лежать адреса — указатели на другие ячейки памяти. Использование указателей дает множество преимуществ. Например, массив строк (а каждая строка в свою очередь является массивом) может быть организован как массив указателей на строки. В этом случае перестановка двух строк не требует перетаскивания всех символов строки с места на место (что потребует большого количества операций). Достаточно только переставить два указателя. Кроме того, использование указателей позволяет не плодить сущностей. Например, полный телефонный номер содержит код страны, код места или сети и собственно телефонный номер. Вместо кода места или сети удобно хранить указатель на этот код. При изменении кода мы можем просто заменить содержание одной ячейки памяти — и все номера, содержащие этот код, автоматически его поменяют.

1.2.4 Структуры

Часто некоторые данные образуют комплекс. Например, фамилия, имя, дата рождения и номер зачетной книжки образуют неразделимую запись и эти данные должны жить вместе. В этом случае говорят о *структуре*. При этом отдельные элементы называются полями. Обратите внимание, что дата также является структурой, которая состоит из числа, месяца, года. Мы будем описывать структуру следующим образом:

```
1. Struct Date{year, month, day;}
2. Struct Stud{FamilyName, FirstName, BirthDay, Id}
```

К полям структуры мы будем обращаться через символ '.' (точка). Например, если у нас есть студент Pupkine, то номер его зачетки будем обозначать как `Pupkine.Id`. Поля структуры в памяти лежат подряд. Отметим важную деталь. При ссылке на другую структуру или строку мы всегда будем понимать, что в структуре лежит указатель на нее, а сама структура может при этом лежать в другом месте памяти. Наши соглашения следуют синтаксису языка Java. На языке C, например, нам следовало бы писать `Pupkine->FamilyName`.

Упражнение 1. *Приведите примеры структур.*

1.3 Способы описания алгоритмов

Для представления алгоритма обычно используются различные способы:

- в виде текстовых описаний действий (текстуальная форма); Эта форма не достаточно строгая, хотя часто она более понятна
- в виде блок схем (графическая форма); Эта форма достаточно прозрачна и на нее даже есть стандарт. Она наиболее удобна для представления потоков данных, но описание алгоритмов с помощью блок-схем громоздко, особенно, если в алгоритме есть циклы и ветвления

- в виде псевдокода (способ описания алгоритмов, использующий ключевые слова языков программирования, но опускающий подробности и специфический синтаксис). Этот способ весьма удобен для тех, кто знаком с программированием. На псевдокод нет стандарта — в разных источниках встречаются разные модификации. Мы будем использовать версию, наиболее похожую на язык C или Java. Текст на псевдокоде не есть непосредственно программа, поскольку обычно опускается целый ряд важных деталей

Пример. Дан массив из чисел. Надо его упорядочить, т.е. переставить числа так, что-бы они шли по возрастанию. Представим решение во всех трех формах.

Словесное описание (наивная сортировка). Ищем в массиве наименьший элемент. Меняем местами первый элемент и наименьший. Ищем наименьший элемент в части массива, начинающейся со второго элемента. Меняем местами второй элемент и найденный наименьший и т.д.

Блок-схема и пояснения к ней показаны на рис.1.3

Псевдокод:

```
1. NaiveSort (a[ ]){
2. for ( i=0; i < a.length ; i++){
3.      min = a[i];
4.      jmin = i;
5.      for (j=i+1; j < a.length; j++){
6.          if (min > a[j]){
7.              min = a[j];
8.              jmin = j;
9.          }
10.     }
11.     a[jmin] = a[i];
12.     a[i] = min;
13.     }
14. }
```

Несколько особенностей и встречающихся вариаций. В этом примере есть цикл, условный оператор, переменные, присваивания. В разных версиях псевдокода встречаются варианты для обозначения этих действий, например:

Цикл	`for i = 0 until n`
Условный оператор	`if min > a[i] then`
Присваивание	`a[i] := min` *или* `a[i]←min`

Упражнение 2. *Метод сортировки "всплывающий пузырек" заключается в следующем. Идем вдоль массива. Если встретим инверсию (ситуацию, когда два соседних элемента имеют неправильный порядок), меняем их местами и проверяем, не нарушился ли порядок для предыдущей пары элементов. Если порядок предыдущей пары неправильный, то также эти два элемента меняем местами. И так пока не исчезнет инверсия. После чего продолжаем поиск следующей инверсии. Представьте этот алгоритм в виде блок-схемы псевдокода.*

Упражнение 3. *Оцените время работы алгоритма наивной сортировки.*

1.4 Элементы булевой алгебры

Булева алгебра (названа в честь английского математика XIX века Джорджа Буля) рассматривает величины, принимающие только два значения — 1 (true; истина) или 0 (false; ложь). С такими величинами можно производить различные операции — так же, как мы оперируем с утверждениями при рассуждениях. Поэтому булева алгебра является основой формальной логики.

Итак, дано множество из двух элементов

$$B = \{0, 1\} \text{ или } B = \{true, false\}$$

В булевой алгебре рассматриваются функции в многомерном множестве B^n вида:

$$B^n \to B$$

Сколько всего разных булевых функций от n переменных? Например, для отображения $B \to B$ могут быть такие варианты:

$0 \to 0$	$0 \to 1$	$0 \to 1$	$0 \to 0$
$1 \to 0$	$1 \to 1$	$1 \to 0$	$1 \to 1$
$false$	$true$	$\neg x$	x

то есть существует 4 типа функции $B \to B$:

$$\{B \to B\} = \{true, false, \neg x, x\}$$

Если речь идет об отображении $B^2 \to B$ (n = 2; двухмерность: 2 значения на входе, 1 — на выходе), то число вариантов функции увеличивается и становится равным уже 2^4. В общем виде:

$$|\{B^n \to B\}| = 2^{2^n} \tag{1.1}$$

Упражнение 4. *Докажите равенство 1.1.*

1.4.1 Основные операции булевой алгебры

Выделяют несколько основных операций (функций) над булевыми переменными: логическое "или"(дизъюнкция, OR, логическое сложение, $\vee$, |), логическое "и"(конъюнкция, AND, логическое умножение, $\wedge$, &), логическое отрицание "не"(NOT, $\neg$, !).

В программировании вместо математических символов для операций используют обозначения:

текст	программа
$\vee$	|
$\wedge$	&
$\neg$	!

Значения функций двух переменных (например, OR или AND) или одной переменной (например, NOT) удобно записывать в виде матрицы:

OR ($\vee$) — логическое «или»:

or	0	1
0	0	1
1	1	1

AND ($\wedge$) — логическое «и»:

and	0	1
0	0	0
1	0	1

NOT ( ) — отрицание:

not	0
0	0
1	1

Можно записывать значение булевой функции (любого числа переменных) в виде таблицы. Например,

X,Y	**and**
0,0	0
0,1	0
1,0	0
1,1	1

Задача : чему равно значение $f(x,y) = (x \wedge y) \vee (x \wedge \neg y)$? Решение: запишем все возможные значения в виде таблицы:

X,Y	**and**
0,0	0
0,1	0
1,0	1
1,1	1

Отсюда $f(x,y) = x$.

1.4.2 Другие часто используемые функции

XOR ($\hat{}$) — "исключающее или":

$\hat{}$	0	1
0	0	1
1	1	0

EQ ($\equiv$) — совпадение:

$\equiv$	0	1
0	1	0
1	0	1

Следствие (импликация, $y \Rightarrow x$ — "из y следует x"):

$y \Rightarrow x$	0	1
0	1	1
1	0	1

Задача: Построим булеву функцию от n переменных $x_1, x_2, ..., x_n$ следующим образом: для начала напишем ВСЕ комбинации вида

$$[\neg]x_1 \wedge [\neg]x_2 \wedge ... \wedge [\neg]x_n,$$

где $[\neg]$ означает, что в каких-то комбинациях берем отрицание, а в каких-то — нет. Затем все полученные комбинации объединяем знаком $\vee$. Чему равно значение полученной функции?

Решение: Итак, мы рассматриваем функцию такого вида:

$$\begin{aligned} f(x_1, x_2, ..., x_n) = \quad & (x_1 \wedge x_2 \wedge ... \wedge x_n) \vee \\ & (\neg x_1 \wedge x_2 \wedge ... \wedge x_n) \vee \\ & (x_1 \wedge \neg x_2 \wedge ... \wedge x_n) \vee \\ & ... \\ & (\neg x_1 \wedge \neg x_2 \wedge ... \wedge \neg x_n) \end{aligned}$$

Идея решения состоит в том, что при любом наборе значений $x_1, x_2, ..., x_n$ хотя бы в одной скобке будет 1 (true) и, следовательно, $f(x_1, x_2, ..., x_n) = true$. Более формальное доказательство возможно по индукции.

Важные тождества

$$\begin{array}{lcl}
\neg(x \vee y) & = & \neg x \wedge \neg y \\
\neg(x \wedge y) & = & \neg x \vee \neg y \\
x \wedge true & = & x \\
x \vee true & = & true \\
x \wedge false & = & false \\
x \vee false & = & x \\
x \equiv y & = & (x \wedge y) \vee (\neg x \wedge \neg y) \\
x \hat{} y & = & (x \vee y) \wedge (\neg x \vee \neg y) \\
x \hat{} y & = & \neg(x \equiv y)
\end{array}$$

Теорема 1. *Любую логическую функцию n переменных можно представить с помощью основных операций ($\wedge$, $\vee$, $\neg$).*

Доказательство. Доказательство данного утверждения предлагается проводить по индукции:

- для $n = 1$ есть 4 типа функции от одной переменной: $true, false, x, \neg x,$. Они очевидно выражаются через базовые операции

$$\begin{array}{lcl}
true & = & x \vee \neg x; \\
false & = & x \wedge \neg x; \\
(x) & = & x; \\
(\neg x) & = & \neg x
\end{array}$$

- пусть с помощью основных операций можно представить функцию $n-1$ переменных. Тогда для любого n:

$$\begin{array}{rl}
f(x_1, ..., x_{n-1}, x_n) = & (f(x_1, ..., x_{n-1}, true) \wedge x_n) \vee \\
& (f(x_1, ..., x_{n-1}, false) \wedge \neg x_n)
\end{array}$$

  Заметим, что $f(x_1, ..., x_{n-1}, true)$ и $f(x_1, ..., x_{n-1}, false)$ на самом деле являются двумя функциями $n-1$ переменной, и по предположению индукции могут быть представлены в виде комбинации базовых операций. Следовательно, всякую булеву функцию любого числа переменных можно разложить на функции and, or, not. Индукция завершена.

□

Утверждение теоремы можно еще сузить. Для представления любой логической функции любого числа переменных достаточно операций $\vee$, $\neg$, так как $\wedge$ представить можно как: $(x \wedge y) = \neg(\neg x \vee \neg y)$

На самом деле булева алгебра может рассматриваться с одной стороны, как алгебра высказываний, а с другой, как алгебра множеств. Например утверждение {элемент x принадлежит множеству A} $\vee$ {элемент x принадлежит множеству B} эквивалентно утверждению {элемент x принадлежит множеству $A \cup B$}.

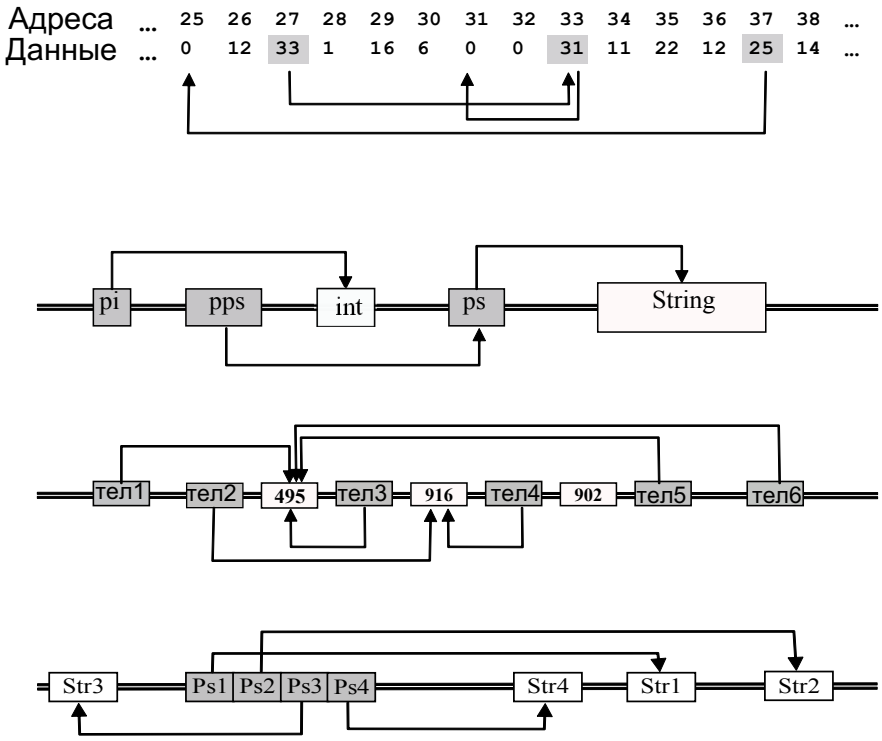


Рис. 1.2: Указатели (показаны серым) содержат адреса в памяти, по которым расположены сами объекты. а) фрагмент памяти. Верхняя строка — адреса (они идут подряд), нижняя строка — содержимое ячеек памяти. б) pps — указатель на указатель. в) иллюстрирует ситуацию с кодами сети. Четыре телефона ссылаются на код 495. При смене кода на 499 достаточно поменять значение в одной ячейке памяти — и сразу автоматически Тел1, Тел.3, Тел5 и Тел6 приобретут новые коды. г) Масив строк — указатели лежат в памяти подряд, Сами строки разбросаны по памяти.

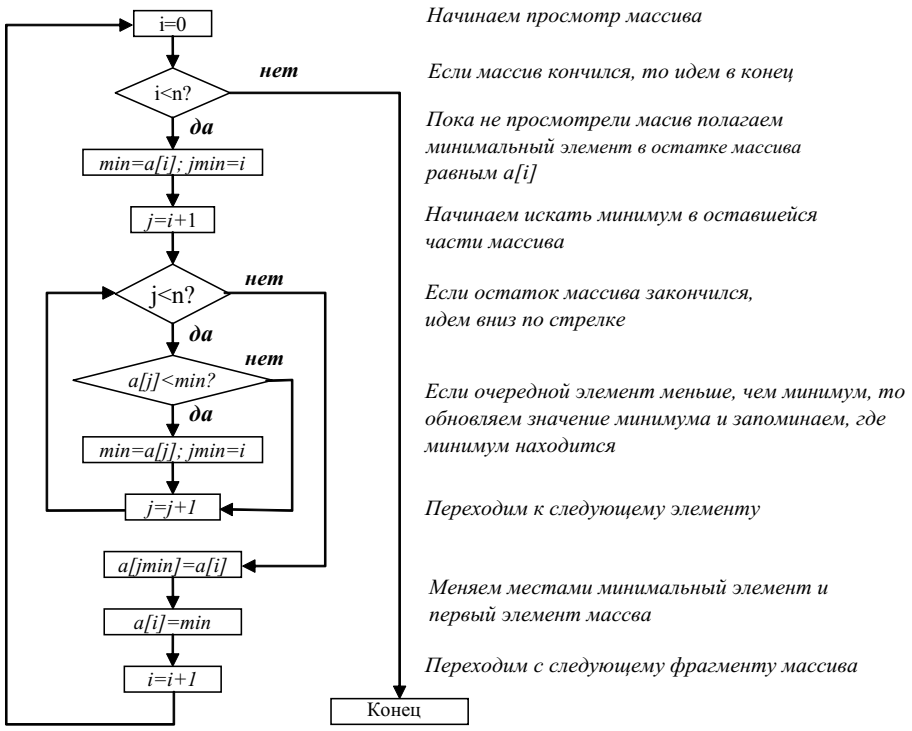


Рис. 1.3: Пример блок-схемы для наивной сортировки массива.

Глава 2

Поиск, сортировка и структуры данных

2.1 Поиск элемента в массиве

2.1.1 Поиск в несортированном массиве

Рассмотрим простейшую задачу — найти телефон в телефонной книге. Для этого разберем сначала, что такое телефонная книга. Телефонная книга состоит из записей (вспомним, что такое структура), а каждая запись состоит из имени и номера телефона. При поиске в телефонной книге, как правило, ищут по имени. То поле, по которому производят поиск, называется ключом, а остальные поля называются данными. В некоторых случаях ключ и данные совпадают (например, в простом массиве чисел).

Итак, ищем в книге Тютькина. Поиск устроен очень просто.

1. $i = 0$
2. Берем элемент номер i
3. Проверим, не Тютькин ли это
4. Если Тютькин, то **Ура!**
5. Иначе переходим к следующему элементу: $i = i + 1$
6. Если массив не кончился, то идем к 2
7. **Увы...**

Блок-схема алгоритма приведена на рис.2.1

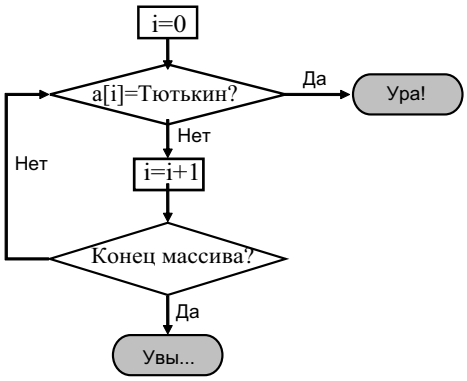


Рис. 2.1: блок-схема поиска в не сортированном массиве

Псевдокод:

```
1. SearchElement (Elm e, a[]){
2.   for (i =0; i<a.length; i++){
3.   if (a[i] == e)
4.   return "Ура!!!";
5.   }
6.   return (Увы...)
7. }
```

Время работы алгоритма в лучшем случае $T = O(1)$ — когда первый элемент содержит искомый ключ. Но это бывает крайне редко. Худшим случаем для алгоритма является отсутствие искомого элемента, или если искомый элемент находится в конце массива. Время работы в худшем случае есть $T = O(L)$. В среднем искомый элемент находится в середине массива. В первом случае нам в среднем надо просмотреть половину массива, во втором надо просмотреть весь массив. И в том и в другом случае время работы алгоритма $T = O(L)$, где L — длина массива. Так ли это?

Проведем более строгий анализ. Пусть вероятность совпадения ключа с запросом поиска равна p, а вероятность несовпадения $q = 1 - p$. Тогда вероятность того, что цикл на строке 2 пройдет один раз, равна p. Вероятность того, что цикл пройдет две итерации равна pq , поскольку на первой итерации было несовпадение, а на второй — совпадение. Вероятность того, что цикл остановится на итерации i, равна $q^i p$. Математическое ожидание числа оборотов цикла равно:

$$E = \begin{array}{l} p + 2pq + 3pq^2 + ... + (L-1)pq^{L-2} + Lq^{L-1} = \\ p(1 + 2q + 3q^2 + ... + (L-1)q^{L-2}) + Lq^{L-1} \end{array}$$

Последний член суммы не содержит множителя p, поскольку здесь нам уже все равно, совпал последний элемент или нет — цикл-то мы все равно

прокрутили. Вычислим сумму в скобках. Это вроде бы почти геометрическая прогрессия, только нам мешают множители $2, 3, ..., (L-1)$. Нетрудно заметить, что сумма в скобках есть производная по q от суммы:

$$1 + 2q + 3q^2 + ... + (L-1)q^{L-2} =$$

$$\frac{d(q + q^2 + ... + q^{L-1})}{dq} =$$

$$\frac{d}{dq}\left(\frac{q(1-q^{L-1})}{1-q}\right)$$

Вспоминая математический анализ, легко вычисляем эту производную:

$$S = \frac{1 - Lq^{L-1} + (L-1)q^L}{(1-q)^2}$$

Подставляя в формулу для математического ожидания и вспоминая, что $q = 1 - p$ получаем:

$$E = pS + Lq^{L-1} = \frac{1-q^L}{1-q}$$

Из того же матанализа мы знаем, что для $p \ll 1/L$ $(1-p)^L \approx 1 - Lp$. Поэтому получаем

$$E = \frac{1 - p - (1-p)^L}{p} \approx \frac{-p + Lp}{p} = L = O(L)$$

Таким образом, мы получили оценку времени работы алгоритма — время работы есть $O(L)$. Заметим, что обычно p достаточно мало. Однако, если p велико (т.е. вероятность встречи ключа высока), то время работы будет достаточно малым — при больших p оценка будет $O(p/q)$. Иными словами, если в нашей записной книжке есть только Тютькины и Пупкины, то поиск Тютькина быстро приведет к успеху.

Надо иметь возможность добавлять элементы в массив и удалять их из него. Добавление элемента происходит просто — надо в конец массива (если есть место) дописать элемент. Эта операция требует константного времени $T = O(1)$. Для вставки элемента в заданную позицию надо освободить место, для чего подвинуть все элемента массива. Это требует времени порядка $O(L)$. При удалении элемента надо часть массива справа от удаляемого элемента сдвинуть влево. Эта операция требует времени, пропорционального длине оставшейся части массива. В среднем — $O(L)$. Итак,

Простой массив	
Добавление	$O(1)$
Вставка	$O(L)$
Удаление	$O(L)$
Поиск	$O(L)$

2.1.2 Поиск в сортированном массиве

Где нужное слово найти проще — в случайном списке слов или в словаре? А чем хорош словарь? Какое его основное свойство? Слова в словаре *отсортированы, упорядочены.* Сортировать можно только такие массивы, в которых содержатся сравнимые элементы. Объекты называются сравнимыми, если Для $\forall x, y$ существует три ситуации: $x < y, x > y, x = y$, причем для операций $>, <$ выполняется условие транзитивности: если $a < b$ и $b < c$, то $a < c$. Заметим, что сравнивать можно не все элементы! Например, не существует естественного способа сравнивать точки на плоскости или в пространстве. Бывают случаи, когда между некоторыми элементами множества можно установить сравнение, а между другими — нельзя. Эти случаи мы рассмотрим при обсуждении алгоритмов на графах.

Отсортировать массив в соответствии с правилом сравнения — значит, выстроить его элементы в таком порядке, что

$$\begin{array}{lll} \forall i > 0 & a[i-1] \leq a[i] & \text{массив упорядочен по возрастанию} \\ \forall i > 0 & a[i-1] \geq a[i] & \text{массив упорядочен по убыванию} \end{array}$$

Слова сравнивают лексикографически. Этот способ сравнения используется, в частности, при построении словарей. Суть его сводится к следующему. Отбрасываем совпадающие части слов сначала, потом сравниваем первую пару букв, которые не совпадают. Какая буква ближе к началу алфавита, то слово и меньше. Если одно слово является началом другого слова, то более короткое слово меньше. Пример: ”пингвин” $<$ ”прорыв”;

Преимущество сортированного массива в быстроте поиска. Вспомним, как ищется слово в словаре. Можно листать словарь сначала, пока не наткнешься на нужное слово. Однако, это не самый эффективный способ. Можно поступить по-другому:

1. Делим массив пополам.

2. Проверяем, попало ли слово на середину, и если да, то Ура!

3. Смотрим, следует ли искать слово в левой или в правой половине.

4. Ищем в соответствующей половине таким же образом.

Покажем работу алгоритма на примере поиска слова "Добро"в словаре. После первого деления пополам находим слово ”Клюв”. Слово, которое ищем (ключ) меньше, чем ”Клюв”, поэтому продолжаем поиск в верхней половине словаря. Делим ее пополам и находим слово ”Викинг”. Это слово меньше ключа, следовательно ищем в части словаря между словами ”Викинг” и ”Клюв”.

Автор	Автор	Автор	Автор
Ангар	Ангар	Ангар	Ангар
Бензин	Бензин	Бензин	Бензин
Викинг	Викинг	**Викинг**	Викинг
Добро	Добро	Добро	**Добро**
Квадрат	Квадрат	Квадрат	Квадрат
Клюв	**Клюв**	Клюв	Клюв
Лень	Лень	Лень	Лень
Пингвин	Пингвин	Пингвин	Пингвин
Порыв	Порыв	Порыв	Порыв
Рассвет	Рассвет	Рассвет	Рассвет
Сказка	Сказка	Сказка	Сказка
Яхта	Яхта	Яхта	Яхта

Заметим, что на каждом этапе мы ищем в части словаря — полный словарь есть частный случай части словаря. Вот алгоритм поиска в части словаря:

```
1. B_Search (elm e, int from, int to){
2.   int i = (to-from)/2;
3.   if (e == a[i]) return(УРА!);
4.   if (to == from) return(УВЫ...);
5.   if (e>a[i])  return B_Search(e,i+1,to) ;  // рекурсия!!
6.   if (e<a[i])  return B_Search(e,from,i-1); // рекурсия!!
7. }
```

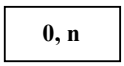
В строках 5, 6 процедура B_Search вызывает саму себя. Такой вызов называется *рекурсией*. Рекурсия — это аналог метода математической индукции, когда более длинная задача сводится к решению точно такой же, но более короткой. Рекурсивный вызов процедур широко используется в алгоритмах. Например, вычисление факториала можно представит в виде рекурсии:

```
1. factorial(n)={1, n==1; n*factorial(n-1)}
```

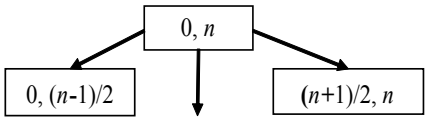
Упражнение 5. *Напишите алгоритм бинарного поиска без рекурсии в виде цикла.*

2.1.3 Анализ алгоритма бинарного поиска

Чтобы оценить время работы бинарного поиска элемента в отсортированном массиве, представим этапы работы алгоритма в виде узлов и листьев дерева (дерево поиска). Узлам соответствуют части ("половинки") массива. Корень дерева — сам массив (пусть $from$ будет 0 — первый элемент массива, to будет $n-1$ — последний элемент массива).



На первом шаге работы алгоритма листьями дерева становятся половинки (без среднего элемента) этого массива: до и после $n/2$:



В итоге возникает дерево, пройдя по которому, можно найти нужный нам элемент массива (рис.2.2) Понятно, что прохождение по этому дереву ана-

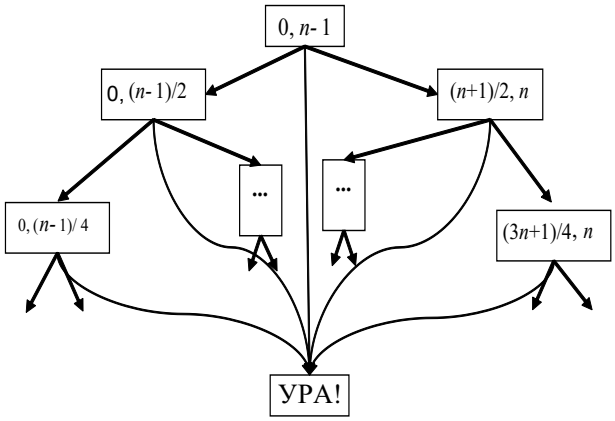


Рис. 2.2: дерево алгоритма поиска элемента в сортированном массиве

логично работе метода B_Search. Каждый следующий этап соответствует узлу, из которого выходят три ветви — три возможности:

1. закончить работу, найдя нужный элемент в середине соответствующей части массива (Ура!)
2. продолжить поиск в начальной половине данной части массива
3. продолжить поиск в конечной половине данной части массива

В худшем случае время работы B_Search равно количеству делений массива пополам, т.е. высоте дерева поиска. Какой случай является самым неблагоприятным? Когда мы начинаем из середины массива, а искомый элемент является самым первым или самым последним в этом массиве; в таком случае придется обойти все дерево, прежде чем мы найдем нужный элемент.

Высота дерева h зависит от числа листьев в нем. Обратите внимание: наше дерево — "красивое", "правильное" и симметричное, из каждой его

вершины выходят три дочерних. У этого дерева число листьев $\leq n$ (n — длина нашего массива). Для h и n имеем такую оценку:

$$2^{h-1} \leq n \leq 2^h$$

или

$$h - 1 \leq \log_2 n \leq h$$

Отсюда получаем, что время работы алгоритма $T(n) = O(h) = O(\log n)$. Время работы бинарного поиска — *логарифмическое*, тогда как время работы "наивного"поиска по массиву той же длины — *линейное*. Для массива длиной миллион элементов время работы алгоритма наивного поиска в худшем случае составит миллион операций, а бинарный поиск справится с этой задачей в худшем случае примерно за 20 операций. Однако для работы B_Search необходим отсортированный массив.

Сортированный массив — хорошая конструкция для поиска, но если нам надо добавить или удалить элемент — возникают проблемы. Для удаления элемента нам надо все элементы с бóльшими индексами подвинуть налево. Вот алгоритм удаления элемента номер k из массива:

```
1. DelArray(a[], k){
2.    for(i=k+1; i< a.length; i++){
3.        a[i-1]=a[i];
4.    }
5. }
```

Добавление элемента еще хуже. Нам надо, чтобы после добавления элемента массив сохранил свойство упорядоченности. Кроме того, массив имеет заранее определенный размер. Поэтому добавление элемента не должно приводить к переполнению. Если массив полностью заполнен, то вставить элемент уже не удастся (это одно из самых серьезных ограничений в использовании массивов). Если же место для нового элемента есть, то надо, во-первых, найти место куда вставлять элемент (можно использовать бинарный поиск), затем освободить место для вставки (подвинуть весь хвост массива направо), и только после этого вставить элемент.

Упражнение 6. *Напишите алгоритм вставки элемента в сортированный массив.*

Упражнение 7. *Модифицируйте алгоритм бинарного поиска так, чтобы он находил либо индекс элемента, если он есть, либо индекс самого большого элемента, который меньше заданного (т.е. тот элемент, после которого надо вставлять новый).*

2.2 Сортировка массивов

Допустим, есть два отсортированных массива. Можно ли их слить так, чтобы в результате получился сортированный массив? Можно. При этом время

работы такого слияния будет порядка суммарной длины входных массивов. Этот алгоритм был предложен Джоном фон Нейманом в 1945 году. Алгоритм называется сортировкой слиянием (Merge_Sort) В его основе лежат следующие соображения. Если:

- массив разбит на два отсортированных подмассива;
- и мы умеем сливать два отсортированных массива.

Тогда:

- разбиваем массив на две половинки;
- сортируем каждую половинку отдельно;
- сливаем половинки.

В основе этого алгоритма лежит парадигма "разделяй и властвуй", состоящая их 3-х этапов:

- *Разделение:* разделение задачи на несколько подзадач.
- *Покорение:* рекурсивное решение подзадач.
- *Комбинирование:* решение исходной задачи исходя из уже решенных подзадач.

Рекурсия достигает своего нижнего предела, когда длина сортируемого подмассива равна 1, а такой массив можно считать упорядоченным. После этого можно сливать массивы.

Рассмотрим этот алгоритм, реализованный на псевдокоде:

```
1. MergeSort (A, i1, i2){
2.   if (i1 == i2) return A;
3.   i = (i1+i2)/2;
4.   A1 = MergeSort (A, i1, i);
5.   A2 = MergeSort (A, i+1, i2);
6.   return Merge (A1, A2);
7. }
```

Теперь обратимся к алгоритму слияния массивов. Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть имеется две стопки карт, лежащих рубашками вниз так, что в любой момент видно только верхнюю карту в каждой из этих стопок. Пусть также, карты в каждой из этих стопок идут сверху вниз в неубывающем порядке (аналог отсортированных массивов). Как сделать из этих стопок одну? На каждом шаге мы берем меньшую из двух верхних карт и кладем ее (рубашкой вверх) в результирующую стопку. Когда одна из оставшихся стопок становится пустой, мы добавляем все оставшиеся карты второй стопки к результирующей стопке.

Псевдокод такого слияния можно увидеть ниже:

```
1.  Merge(a1[], a2[]) {
2.     a[]=new array[a1.length+a2.length];
3.     int i1=0;
4.     int i2=0;
5.     while ((i1+i2)< a.length ){
6.        if(i1 < a1.length && i2 < a2.length){
7.           if(a1[i1]< a2[i2]){
8.              a[i1+i2]=a1[i1];
9.              i1++;}
10.          else {
11.             a[i1+i2]=a2[i2];
12.             i2++;}
13.          }
14.       if(i1 = a1.length) {
15.          a[i1+i2]=a2[i2];
16.          i2++;}
17.       if(i2 = a2.length) {
18.          a[i1+i2]=a2[i1];
19.          i1++;}
20.       }
21.    return a;
22.    }
```

Поясним псевдокод. В строке 2 мы создаем вспомогательный массив, куда будем записывать результат слияния. Ясно, что длина этого массива равна сумме длин входных массивов. Строки 3 и 4 устанавливают указатели на начало массивов. В строке 6 мы поверяем, есть ли элементы в обоих массивах. Строка 6 — сравнение элементов и перенос соответствующих элементов (строки 11 и 15). При переносе элементов увеличиваются индексы массивов (строки 12,16). Если же один из массивов исчерпан (строки 14, 17 проверяют это), то переносим элементы оставшегося массива (строки 15, 18). Процедура возвращает новый массив (строка 21).

Упражнение 8. *Продемонстрируйте работу слияния на примере массивов:* `[3, 7, 8, 10]; [2, 4, 6, 9]`.

2.2.1 Анализ алгоритма Merge_Sort

Время работы алгоритма определяется временем работы процедуры Merge. Очевидно, что время работы этой процедуры есть $O(N)$. Действительно, время определяется временем работы цикла строки 5, а он прокручивается ровно N раз, где N — длина результирующего массива. Тогда время работы алгоритма можно представить в виде рекурсии:

$$T(N) = 2 \cdot T(N/2) + O(N) \tag{2.1}$$

Первое слагаемое — сортировка подмассивов (их два, длиной $N/2$), второе слагаемое — слияние

Теорема 1 (Теорема о рекурсии Merge_Sort)**.** *Если* $T(N) = 2T(N/2) + O(N) \leq 2T(N/2) + bN$*, где* b *— некоторая константа то существует* C *такое, что для любого* N

$$T(N) \leq C \cdot N \cdot \log N$$

Доказательство. Доказательство по индукции:

- База индукции: при $N = 3$ величина T равна чему-то, например, $a \cdot 3 \cdot \log 3$ и для любого $C \geq a$ не превышает оценку.
- Предположение индукции: Допустим, утверждение верно для $N/2$
- Тогда

$$\begin{aligned} T(N) \leq 2 \cdot T(N/2) + b \cdot N \leq & \quad \text{в силу рекурсии 2.1} \\ 2 \cdot C \cdot N/2 \cdot \log(N/2) + b \cdot N \leq & \quad \text{по предположению индукции} \\ C \cdot N \cdot \log N + N \cdot (b - C \cdot \log 2) & \end{aligned}$$

  полагая C достаточно большим ($C \geq \max\{T(3)/3 \log 3,\ b/\log 2\}$) делаем второй член отрицательным. Первый элемент в максимуме возникает из основания индукции. Второй — обусловлен необходимостью сделать отрицательным второй член. Таким образом, окончательно получаем:

$$T(N) \leq C \cdot N \cdot \log N = O(N \log N)$$

□

Время работы Merge_Sort $T(N) = O(N \log N)$, однако у сортировки слиянием имеется один недостаток: необходимо создание дополнительного массива, что занимает место в памяти.

Упражнение 9. *Попробуйте создать алгоритм сортировки Merge_Sort без использования дополнительных массивов и без рекурсии.*

2.2.2 Быстрая сортировка (QSort)

Алгоритм, предложенный Чарльзом Хоаром в 1962 году, был назван сортировкой разделением. Метод оказался настолько эффективным, что вскоре его стали называть алгоритмом быстрой сортировки (quick sort, QSort).

Сортировка методом QSort основана на одном из свойств упорядоченного массива: Массив отсортирован тогда и только тогда, когда для любого элемента q все элементы массива слева не больше его, а все элемента справа — не меньше:

$$\forall k : (\forall i < ka[i] \leq a[k]) \& (\forall j > ka[j] \geq a[k])$$

Быстрая сортировка, как и сортировка слиянием, основана на принципе "разделяй и властвуй". Для этого берем первый элемент (опорный элемент) в массиве и попробуем сделать перестановки так, чтобы слева от него были элементы меньше, а справа — больше его:

1. Ищем с правого конца первый элемент, который меньше опорного и меняем их местами

2. Ищем слева первый элемент, который больше опорного и меняем их местами

3. Повторяем процедуру до тех пор, пока массив не окажется разбитым на два подмассива

4. После разбиения можно применить ту же процедуру к левой и правой частям массива

После первой итерации справа оказывается часть массива, которая больше q, после второй — слева оказывается часть массива, которая не превышает q, а зона массива, для которой неизвестно отношение к q — сокращается. Процедуру повторяем до тех пор, пока эта зона не сожмется до одного элемента — q. После того, как мы разделили массив на две (вообще-то говоря не равные) части можно к этим частям применить ту же процедуру. Теперь рассмотрим реализацию алгоритма на псевдокоде:

```
1. QSort (A, from, to){
2.    int q;
3.    if (from < to){
4.        q = Partition (A, from, to);
5.        QSort (A, from, q-1);
6.        QSort (A, q+1, to);
7.   }
8. }
```

Пояснения. Если массив содержит больше одного элемента (строка 3), то его сортируем: претасовываем массив так, чтобы появилось две части так, чтобы все элементы в левой части были не больше любого элемента в правой части (строка 4). После этого сортируем по отдельности левую и правую части (строки 5,6). Отметим особенность — левая часть начинается с начала массива (`from`) и заканчивается предыдущим элементом ($q-1$), а правая начинается со следующего за q элементом, т.е. q больше не трогаем.

Метод Partition разбивает данный массив на два подмассива и возвращает позицию опорного элемента.

```
1. Partition (int[] a, int from, int to){
2.    int q=a[from];
3.    int i =from;
4.    int i1=from;
5.    int i2=to-1;
6.    while (i1!=i2){
7.        while (a[i2]>=q)
8.           i2--;
9.        a[i]=a[i2];
```

```
10.       a[i2]=q;
11.       i=i2;
12.       while(a[i1]<q)
13.    i1++;
14.       a[i]=a[i1];
15.       a[i1]=q;
16.       i=i1;
17.    }
18.  return i;  // процедура возвращает номер элемента, который
                // обладает нужным свойством: все элементы справа
                // больше, все элементы слева не больше
19. }
```

Пояснения к алгоритму Partition. На вход алгоритм получает массив и границы фрагмента массива, который мы будем делить на части. Строки 2-5 производят инициализацию. В переменных $i1$, $i2$ хранятся границы необработанной части массива; в части массива с индексами $< i1$ элементы меньше, чем q, а в части массива $> i2$ все элементы не меньше, чем q. В процессе работы мы сужаем необработанную часть массива. Сначала в правой части ищем элемент массива, нарушающий правило, т.е. элемент, который меньше или равен q. Когда его найдем, меняем местами его и первый элемент. Потом ищем "нарушителя"с начала массива и делаем перестановку. Потом продолжаем искать в конце массива и т.д., пока $i1$ и $i2$ не совпадут, что означает, что необработанная часть сжалась до одного элемента.

Точный анализ алгоритма быстрой сортировки достаточно сложен, приведем здесь лишь основные положения. Среднее время работы алгоритма $T(N) = O(N \log N)$. В худшем случае время работы алгоритма квадратичное ($T(N) = O(N^2)$). Худший случай — когда массив уже отсортирован в прямом или в обратном порядке. Если мы заранее знаем об этом, лучше "испортить" массив, перетасовав его элементы в случайном порядке (что занимает N операций), и затем применить QSort ($O(N \log N)$). Избежать этого можно предварительно перетасовав массив за время $O(N)$. И даже в этом случае быстрая сортировка будет работать быстрее наивной. Как узнать, хорош ли массив для работы QSort? Упорядоченность массива оценивается по числу инверсий. Случайный, неупорядоченный массив длины N содержит приблизительно $N/2$ инверсий (теория вероятностей!). Если число инверсий стремится к нулю — массив отсортирован в возрастающем порядке, если к N — в убывающем.

К достоинствам быстрой сортировки относится:

- Высокая скорость работы.
- Простота реализации.

В отличии от сортировки слиянием (Merge_Sort) алгоритм быстрой сортировки (QSort) одним недостатком. Это — неустойчивость (то есть ячейки с одинаковыми данными могут изменить положение друг относительно друга

после сортировки). Этот недостаток не так важен при сортировке массивов чисел, но может оказаться важным при сортировке ключей, которые связаны с некоторыми данными.

Упорядоченный массив тоже имеет свои недостатки. Добавление элементов в него требует усилий:

1. Найти место куда вставить ($T(N) = O(\log N)$)
2. Освободить место для вставки ($T(N) = O(N)$)
3. Вставить элемент

Кроме того, нет гарантий, что размер массива позволит вставку элемента.

Мы уже поняли, что сортированный массив это хорошо. Но возникает вопрос, что лучше по мере добавления элементов вставлять их на правильное место или сначала собрать массив, а потом отсортировать? Оценим время, требуемое на сборку массива. На каждом шаге надо: (1) найти место для вставки ($T(N) = O(\log N)$) ; (2) сделать вставку ($T(N) = O(N)$). Итого для сборки упорядоченного массива необходимо:

$$\sum_i (\alpha \cdot i + \beta \cdot \log i) = \alpha \cdot \frac{N(N+1)}{2} + \beta \cdot \sum_i \log i = O(N^2)$$

Свойства сортированного массива:

Упорядоченный массив	
Вставка	$O(L)$
Удаление	$O(L)$
Поиск	$O(\log L)$

Есть структуры данных лишенные этих недостатков, о них читайте ниже.

2.3 Списки

2.3.1 Односвязный список

Всем знаком принцип построения очереди в поликлинике: каждому следующему человеку достаточно знать, кто стоит перед ним, при этом люди не обязательно стоят строго один за другим, они могут, к примеру, отходить. На этом принципе построена такая структура данных, как связный список (односвязный список). Связный список состоит из элементов, каждый из которых несет какие-то данные (*data*). Кроме того, любой элемент имеет ссылку на следующий за ним элемент (*next*). Последний элемент списка ссылается на пустой элемент — *null*. В отличие от элементов массива, части связного списка могут лежать в разных местах памяти, не обязательно один за другим.

Элемент связного списка можно описать так:

```
1. Struct LinkedListItem {
2.   data d;
3.   LinkedListItem next;
4. }
```

Чтобы определить связный список, достаточно указать первый элемент списка (*root*). На него не ссылается ни один другой элемент. Полезно знать последний элемент (который ссылается на *null*) и размер списка (рис.2.3). Итак, структура связного списка может быть описана:

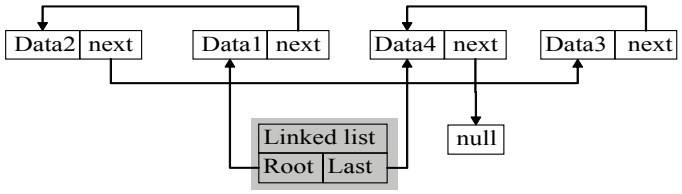


Рис. 2.3: Односвязный список

```
1. Struct LinkedList{
2.   LinkedListItem root;
3.   LinkedListItem last;
4.   int size;
5. }
```

Рассмотрим основные операции для связного списка.

Поиск. Поиск по связному списку определенных данных (data d) ведется от корня. В ходе поиска просматривается по очереди каждый элемент: есть в этом элементе нужные данные? — нет — идем дальше, есть — ура; так до последнего элемента (ссылающегося на null). Следовательно, алгоритм поиска не отличается от такового для массива. Понятно, что и работает он за то же время $T(N) = O(N)$, где N — величина списка

Упражнение 10. *Напишите алгоритм поиска элемента в связном списке.*

Вставка элемента. Плюс связного списка в том, что новый элемент в него можно добавлять всего за одну операцию, независимо от длины списка. Пример из жизни. Вы стоите в очереди в буфет (вы знаете за кем вы стоите). К вам подходит знакомый и он, естественно, встает вместе с вами. Теперь он стоит за тем, за кем стояли вы, а вы стоите за ним.

Если мы знаем элемент, после которого нужно вставить новый, то необходимо просто переключить ссылки.

```
1. insertAfter (LinkedListItem elm, LinkedListItem after) {
2.   if(after==null){
```

```
3.   elm.next=root;
4.   root= elm;
5.   if(size==0) last=elm;
6.   size++;
7.   return;
8.   }
9. elm.next = after.next;
10.   after.next = elm.next;
11. if(elm.next==null) last= elm;
12. size++;
13. }
```

Комментарий к коду. Если мы указываем в качестве параметра $after\ null$, то это означает, что мы вставляем новый корневой элемент (строки 2 - 7). В частности, так можно создавать новый список. Очевидно, что вставка элемента требует константного количества операций: $T(N) = O(1)$.

Удаление. Не получится удалить элемент из связного списка так же просто и быстро, как вставить, поскольку каждый элемент "знает"только следующего соседа, но не предыдущего. Поэтому для удаления элемента придется пройти весь список от root до элемента, за которым следует удаляемый элемент:

```
1. remove(LinkedListItem r){
2.   for(LinkedListItem l=root; l!=null; l=l.next){
3.   if (l.next == r){
4.   l.next=r.next;
5.   if(l==root) root=l.next;
6.   if(l.next==last) last=l;
7.   size --;
8.   return;
9.   }
10.   }
11.   return ERROR!!!
12. }
```

Связный список, в отличие от массива, является структурой данных с не фиксированной длиной. Создавая новый список, мы задаем его *root*, к которому в дальнейшем можно добавлять сколько угодно новых элементов один за другим. Обсуждая массивы, мы упоминали, что фиксированная длина, в зависимости от ситуации, может быть как плюсом, так и минусом.

Когда фиксированная длина — плюс? В случае, если не хочется засорять память лишними пустыми элементами, предпочтительнее списки. Пример из Java: известная всем и часто используемая структура данных с

не фиксированной длиной Vector, или массив в языке Perl. При создании нового экземпляра класса Vector занимается определенное количество памяти — около десяти элементов. Даже если в нашем векторе будет храниться только два элемента, остальные восемь останутся недоступными для других программ и объектов. В такой ситуации лучше было бы использовать массив длиной два элемента. Следовательно, "отъедание" памяти "про запас" — возможный минус структур данных с фиксированной длиной. Связный список таким свойством не обладает, он не занимает лишней памяти, поэтому его не фиксированная длина — только плюс. Но надо иметь в виду, что мы к каждому элементу добавили ссылку. Если поле data занимает один байт, то мы увеличили требование к памяти в 5 раз (вдобавок к одному байту данных мы храним четыре байта ссылки). Успокаивает одно — обычно поле данных достаточно большое. С другой стороны часто встречаются случаи, когда размер множества заранее известен, например, если вам надо создать какие-то данные, привязанные к дню недели. Тогда очевидно, что надо иметь массив размером 7.

Связный список	
Вставка	$O(1)$
Удаление	$O(L)$
Поиск	$O(L)$

2.3.2 Двусвязный список

Односвязный список хорош тем, что в него можно быстро добавлять новые элементы, но, к сожалению, по времени поиска и удаления элемента он не отличается от массива. Быстро удалять элементы не получается, поскольку каждый элемент "знает"только следующего соседа, но не предыдущего.

Этого недостатка лишен двусвязный список — структура, аналогичная списку односвязному и отличающаяся только более сложной системой связей. Каждый элемент двусвязного списка хранит данные, а также две ссылки: на следующий элемент и на предыдущий элемент списка.

Элемент двусвязного списка можно описать так:

```
1.  BLListItem{
2.     data d;
3.     BLListItem next;
4.     BLListItem prev;
5.  }
```

Чтобы описать двусвязный список достаточно указать первый элемент root. Здесь также полезно хранить последний элемент и размер. Структура двусвязного списка

```
1.  Struct BiLinkedList{
2.     BLListItem root;
3.     BLListItem last;
```

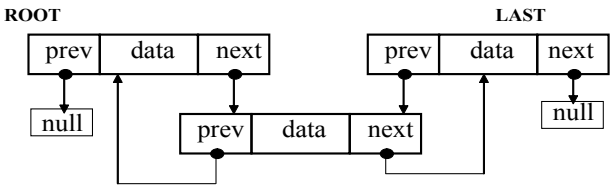


Рис. 2.4: Двусвязный список

```
4.     int size;
5.  }
```

Заметим, что и односвязный, и двусвязный список могут быть циклическими. Для односвязного цикла должно выполняться условие: $last.next = root$: Для двусвязного: $root.prev = last$; $last.next = root$. В подобных циклах теряется всякий смысл понятий *root* и *last*, но обход списка можно начинать с любого элемента.

Поиск элемента в двусвязном списке не отличается ни по алгоритму, ни по затрачиваемому времени от поиска в односвязном списке либо в массиве.

Двусвязный список можно быстро сортировать методом QSort. Процедура Partition требует возможности быстро менять два элемента местами. В случае списков можно менять не элементы целиком (что дольше и не всегда возможно), только их данные, или просто переключить ссылки. Кроме QSort существуют и другие алгоритмы сортировки двусвязного списка за время порядка $T(N) = O(N \log N)$.

К сожалению, быстрый бинарный поиск по списку неосуществим, поскольку мы не можем найти середину списка. В массиве существовал индекс каждого элемента, и получение любого элемента, в том числе среднего, стоило $O(1)$ операций. В списке либо невозможно найти середину, либо, в лучшем случае, это будет стоить $O(N)$ операций, что сведет на нет всю пользу бинарного поиска.

Добавление элемента в двусвязный список, так же, как в односвязный, занимает $T(N) = O(1)$ операций:

```
1.  insertAfter(BLListItem elm, BLListItem after) {
2.     if(after==null){
3.        elm.next=root;
4.        root.prev=elm;
5.        root=elm;
6.        if(size==0) last=elm;
7.        size++;
8.        return;
9.     }
```

```
10.    elm.next= after.next;
11.    BLListItem l2 = after.next;
12.    if(l2!=null) prev=this;
13.    elm.next = after.next;
14.    if(elm.next==null) last=elm;
15.    elm.prev= after;
16.    size ++;
17. }
```

Удаление элемента из двусвязного списка занимает O(1) операций:

```
1.  remove(BLListItem r){
2.     BLListItem prevElm=r.prev;
3.     BLListItem nextElm=r.next;
4.     if(prevElm!=null)
5.        prevElm.next=this.next
6.     else
7.        root=this.next
8.     if(nextElm!=null)
9.     nextElm.prev=this.prev;
10.    if(last=r) last=r.prev;
11.    if(r==root) root=root.next;
12.    size--;
13.  }
```

Двусвязный список	
Вставка	$O(1)$
Удаление	$O(1)$
Поиск	$O(L)$

2.3.3 Пример применения связного списка. Организация данных в памяти компьютера

Память компьютера линейна, то есть переменные, указатели и т.п. расположены один за другим. Выше мы уже обсуждали организацию памяти с помощью указателей на примере массива.

Структура оперативной памяти в современных компьютерах очень сложна. Мы рассмотрим простейшую систему распределения памяти, на которой, к примеру, базировались ранние версии операционной системы UNIX и DOS. Такая память организована в виде ”кусков” разного размера. У каждого такого "куска"есть часть, несущая данные и так называемый контрольный блок (control block, CB). Контрольные блоки напоминают элементы связного списка — каждый из них несет информацию об объеме данных ”подопечного” участка памяти и его состоянии (занят/свободен), а также ссылку на следующий контрольный блок(рис.2.5).

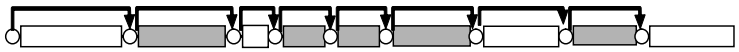


Рис. 2.5: Распределение памяти. Кружки обозначают контрольный блок, белые прямоугольники — свободную память, серые прямоугольники — занятую память.

Как "положить"в такую память, к примеру, массив объемом 500 байт? Идем к первому СВ спрашиваем: "участок памяти занят?". Пусть он свободен, тогда спрашиваем: "достаточно ли места для нашего массива?". Нет, этот участок памяти может хранить только 300 байт. Идем дальше, пока не найдем незанятый участок нужной длины. Если находим участок длины 1000 байт, делим его на два, вставляя новый СВ. Если два соседних участка памяти свободны, они объединяются под контролем одного СВ.

Память, построенная таким образом может привести к феномену "дырявой памяти". Бывают ситуации, когда объем свободной памяти большой, а положить в нее ничего нельзя — память разбита на очень маленькие кусочки. Современные системы используют улучшенными схемами распределения памяти и управления ею.

Упражнение 11. *Напишите алгоритм распределения памяти. Необходимо две процедуры — `alloc(size)` — запрос на занятие памяти размера `size`. процедура возвращает указатель на предоставленную память. Освобождения памяти — `free`(указатель на блок памяти). При этом имейте в виду, что `alloc` возвращает не СВ, а именно свободное место, а `free` получает указатель на занятую память, а не на СВ.*

2.3.4 Очередь и стек

Со списками связаны важные структуры данных — очереди. Они еще больше, чем списки, напоминают настоящую живую очередь. В поликлинике первый пришедший человек (первый добавленный в очередь элемент) первым заходит к врачу (выбывает из очереди). Следовательно, структура данных Очередь (`Queue`) построена по принципу `FIFO` — first input, first output. Раньше добавленный в очередь элемент раньше из нее и выйдет.

Очередь — это множество элементов, позволяющее добавлять один элемент и извлекать один элемент. При этом соблюдается правило: первым извлекается элемент, пришедший первым. Очередь строится на базе односвязного списка, только элементы помнят не следующий элемент, а предыдущий (как в жизни — Вы помните ЗА кем Вы стоите в очереди. А не того, кто за Вами). Естественно необходимо знать начало и конец очереди — начало для того, чтобы извлекать из очереди элементы, а конец — для того, чтобы добавлять элементы. Касательно очереди существует две главные процедуры — `void put(elm e)` — добавляет элемент `e` в конец очереди

и `elm get()` — берет элемент из начала очереди. В компьютере примером очереди служит очередь задач.

Стек (Stack) , как и очередь, — это множество элементов, позволяющее добавлять один элемент и извлекать один элемент. Однако стек построен по противоположному принципу: первым из него извлекается элемент, добавленный последним (`LIFO` — last input, first output, вспомните стопку тарелок). Для работы со стеком необходимы две основных процедуры: `void push (elm e)` — кладет элемент e в стек и `elm pop()` — извлекает элемент из стека.

Стек очень широко используется в организации работы компьютера. Приведем простой пример из жизни. Представьте, что Вы читаете книгу. Вдруг зазвонил телефон. Вы отложили книгу и начали телефонный разговор. При этом Вы вернетесь к книге, когда закончите телефонный разговор, иными словами Вы положили чтение книги в стек. Теперь во время телефонного разговора позвонили в дверь. Вы извинились и поставили телефонный разговор в стек. Вы идете открывать дверь. В этот момент ваш любимый кот решил съесть вашу колбасу. Посетитель тоже отправился в стек. После того, как Вы разобрались с котом Вы извлекаете из стека посетителя и разбираетесь с ним. После ухода посетителя Вы извлекаете из стека то, что там лежит — телефонный разговор. Поговорив, вы снова извлекаете из стека книгу и продолжаете читать. Подобных событий в компьютере происходит множество Можно привести два примера.

Разбор формулы (см. табл. 2.3.4). Пусть дана формула:

$$2 + 3 + (4 * (2 + (2 * 3))/3) - 1$$

В этой формуле мы предполагаем, что нет приоритета операций, а порядок вычислений определяется только скобками. Как все это будет вычислять компьютер. Пусть у него есть регистр, в котором будет накапливаться результат. Итак, считываем первое число и помещаем в регистр. Затем считываем операцию и второе число. Производим сложение. Теперь у нас в регистре лежит 5. Далее считываем операцию и видим, что дальше идет скобка. Это сигнал к тому, чтобы содержание регистра и текущую операцию отложить до тех пор, пока эта скобка не закроется, т.е. положить содержимое регистра и операцию в стек. Далее считываем очередное число и операцию. Снова наталкиваемся на скобку — кладем регистр и операцию в стек, и т.д. На каком-то этапе мы встретим закрывающую скобку — это сигнал к тому, чтобы снять со стека регистр и операцию.

Упражнение 12. *Напишите структуры и алгоритмы для очереди и стека.*

Упражнение 13. *Разберите формулу 1+2*(3+4*(5+6*(7+8*9)))*

Упражнение 14. *Напишите алгоритм разбора формулы, считая, что за одно считывание читается целиком число или символ операции или скобка и что стек уже реализован и имеет методы push и pop.*

Таблица 2.1: пример разбора формулы

прочитано	рег.	опер.	стек	Комментарий
	0			
2	2			
2+	2	+		
2+3	5			2+3=5
2+3+	5	+		
2+3+(	0		5+	Кладем регистр и операцию в стек
2+3+(4	4		5+	
2+3+(4*	4	*	5+	
2+3+(4*(	0		4*;5+	Кладем регистр и операцию в стек
2+3+(4*(2	2		4*;5+	
2+3+(4*(2+	2	+	4*;5+	
2+3+(4*(2+(	0		2+;4*;5+	Кладем регистр и операцию в стек
2+3+(4*(2+(2	2		2+;4*;5+	
2+3+(4*(2+(2*	2	*	2+;4*;5+	
2+3+(4*(2+(2*3	6		2+;4*;5+	2*3=6
2+3+(4*(2+(2*3)	8		4*;5+	снимаем со стека 2+; 2+6=8
2+3+(4*(2+(2*3))	32		5+	снимаем со стека 4*; 4*8=32
2+3+(4*(2+(2*3))/	32	/	5+	
2+3+(4*(2+(2*3))/3	10.67		5+	32/3=10.67
2+3+(4*(2+(2*3))/3)	15.67			снимаем со стека 5+; 5+10.67=15.67
2+3+(4*(2+(2*3))/3)–	15.67	–		
2+3+(4*(2+(2*3))/3)–1	14.67			15.67–1=14.67

Аналогичная картина происходит, когда вызывается подпрограмма. В этом случае в стеке запоминается точка возврата и состояние текущей программы (все ее внутренние переменные). По окончании работы подпрограммы со стека снимается точка возврата и восстанавливается контекст. В частности при рекурсивном вызове, например программы B_Search переменная i, объявленная в строке 2 алгоритма, появляется еще раз. Ее изменение никак не повлияет на состояние такой же переменной в вызывающей процедуре это просто две разные сущности, лежащие в разных местах памяти.

2.4 Бинарное дерево поиска

Бинарное дерево поиска — это дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовчм их левым и правым), и все вершины, кроме корня, имеют родителя. Вершины, не имеющие потомков, называются листами (рис.2.6).

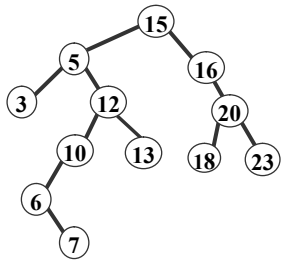


Рис. 2.6: Бинарное дерево поиска

Для каждой вершины бинарного дерево поиска выполняется правило: ключи всех элементов левого поддерева меньше ключа данной вершины, а все ключи в правом поддереве не меньше ключа данной вершины. Иначе это можно записать так:

$$\text{left and all childs} < \text{this, если left существует}$$
$$\text{this} \leq \text{right and all childs, если right существует}$$

Какие типы данных можно хранить в виде бинарного дерева поиска? Сразу хочется сказать "числа". На самом деле не только. Можно хранить любые данные, на которых задана процедура сравнения. К примеру, если мы хотим хранить белки, нам нужно условиться, как эти белки сравнивать (по массе, заряду, коэффициенту седиментации и др.), что значит "этот белок строго больше того" (масссивнее, более положительно заряжен, медленнее осаждается). Можно также хранить строки. Структура данных может быть представлена в виде:

```
1.  BTreeElm{
2.     BTreeElm parent;
3.     BTreeElm left;
4.     BTreeElm right;
5.     Data  d;
6.  }
```

Корень дерева не имеет родителей, его parent представляет собой пустой элемент (*null*), соответственно, листья дерева не имеют ни левого, ни правого потомка. Очевидно, что любое поддерево бинарного дерева поиска является бинарным деревом поиска.

Основное свойство любого дерева — его высота h. Для бинарного дерева поиска как для структуры данных это очень важная величина: от нее зависит время вставки, удаления и поиска элемента по дереву. В "хорошем" дереве все эти времена логарифмические: $T_x(N) = O(\log_2(N))$, где x — вставка, поиск, удаление. Почему поиск занимает логарифмическое время, понятно — вспомним бинарный поиск по сортированному массиву. Вставка сводится к поиску нужной позиции в зависимости от вставляемого элемента. Удаление элемента сводится к поиску наименьшего элемента в правом поддереве (см. ниже.)

2.4.1 Поиск в бинарном дереве

Поиск в бинарном дереве осуществляется по следующему алгоритму:

1. Если ключ равен вершине, то Ура!!!
2. Если он меньше вершины, то переходим на левую дочернюю вершину.
3. Иначе переходим на правую дочернюю вершину.
4. Если дерево кончилось, то Увы...

Здесь показан алгоритм поиска в поддереве, начинающемся в `treeElm`:

```
1.  TreeSearch(treeElm, key){
2.     if(treeElm == null) return "not found";
3.     if(treeElm.data == key ) return treeElm;
4.     if(treeElm.data < key)
5.        return TreeSearch(treeElm.right,key);
6.     else
7.        return TreeSearch(treeElm.left ,key);
8.  }
```

Для поиска в дереве надо вызвать эту процедуру для корня: `TreeSearch(root, key)`. Обратим внимание на то, что поиск происходит рекурсивно: если очередной элемент не найден, то ищем в соответствующем поддереве. Время работы $T(N) = O(h)$, где h — высота дерева.

2.4.2 Добавление элемента в бинарное дерево поиска

Бинарное дерево писка — упорядоченная структура, поэтому "просто так" элемент в него не добавишь. Для каждого нового элемента нужно искать место так, чтобы и после добавления элемента структура оставалась бинарным деревом поиска.

Проще всего добавлять новые элементы в качестве листьев. В зависимости от ключа нового элемента, он вставляется в то или иное поддерево. Для вставки идем по дереву вниз так же, как и при поиске до первого свободного места, и туда вставляем новый элемент в качестве листа дерева.

Можно предложить две формы записи вставки элемента в дерево: рекурсивную и циклическую запись.

Рекурсивная форма:

```
1.  Tree_Insert(tree_elm, new_elm){
2.     if(new_elm > tree_elm){
3.        if(tree_elm.right==null){
4.           tree_elm.right=new_elm;
5.           new_elm.parent=tree_elm;
6.           return;
7.        }
8.        Tree_Insert(tree_elm.right, new_elm);
9.     }
10.    else{   //Аналогично для left
11.       if(tree_elm.left==null){
12.          tree_elm.left=new_elm;
13.          new_elm.parent=tree_elm;
14.          return;
15.       }
16.    Tree_Insert(tree_elm.left, new_elm);
17.    }
18.  }
```

Вставка с использованием цикла:

```
1.  InsertTree(tree_elm, new_elm){
2.     while(true){
3.        if(tree_elm <= new_elm){
4.           if(tree_elm.right == null){
5.              tree_elm.right=new_elm;
6.              new_elm.parent=tree_elm;
7.              return;
8.           }
9.           tree_elm = tree_elm.right;
10.       }
11.       else{   //Аналогично для left
12.          if(tree_elm.left == null){
13.             tree_elm.left=new_elm;
14.             new_elm.parent=tree_elm;
15.             return;
16.          }
17.       tree_elm = tree_elm.right;
18.       }
19.    }
20. }
```

Какая форма лучше?

- Рекурсивный тип алгоритма представляется более прозрачным.

- Часто можно записать алгоритм в виде цикла, но не всегда это легко сделать
- Использование цикла предпочтительно, поскольку несколько быстрее, и нет опасности переполнения стека

В любом случае, добавление элемента в дерево стоит $T(N) = O(h)$, где h — высота дерева.

Упражнение 15. *Напишите алгоритм поиска элемента в двоичном дереве в виде цикла.*

Упражнение 16. *Напишите алгоритм двоичного поиска в массиве в виде цикла.*

2.4.3 Поиск минимального (максимального) элемента в дереве

Дерево — это частный случай поддерева. *Минимальный элемент в поддереве — это самый левый узел*, соответственно, *максимальный — самый правый узел*. Приведем псевдокод поиска минимального элемента в поддереве, растущем из элемента e:

```
1.  getMin(BTreeElm e){
2.     while(e.left!=null)
3.        e=e.left;
4.     return e;
5.  }
```

Аналогично для максимального элемента:

```
1.  getMax(BTreeElm e){
2.     while(e.right!=null)
3.        e=e.right;
4.     return e;
5.  }
```

Найти минимальный (максимальный) элемент в поддереве удается за время порядка $T = O(h)$, где h — высота поддерева.

2.4.4 Поиск следующего по возрастанию элемента

Часто встречается задача вывода элементов какой-либо структуры данных в порядке возрастания (убывания). Чтобы решить эту задачу для бинарного дерева поиска, нужно уметь искать следующий (предыдущий) по возрастанию элемент. Для данного элемента следующим будет либо самый левый элемент (если существует) в правом поддереве. Такого элемент может и не быть в случае, если текущий элемент не имеет правого поддерева (например, является листом) или в правом поддереве все поддеревья, в том

числе и листья, — левые (т.е. меньше, чем текущий элемент). В этом случае надо искать элемент среди предков. Если предок — левый, т.е. текущий элемент является правым потомком, то предок меньше, чем текущий элемент. В этом случае надо подниматься вверх до тех пор, пока не найдем правого предка. Если же и такого предка не находим, то заключаем, что не существует следующего по возрастанию элемента, т.е. текущий элемент является максимальным.

```
1.  GetNext(BTreeElm e){
2.     if(e.right != null)
3.        return getMin(e.right);
4.     while(e.perent != null && e.parent.right == e)
5.        e = e.parent;
6.     if(e.parent != null && e.parent.left == e)
7.        return e.parent;
8.     return null;
9.  }
```

Пояснение к псевдокоду. Строка 2 – проверка, есть ли правый ребенок. Если есть (строка 3), то ищем минимальный элемент в правом поддереве. Если такого элемента нет, и текущий элемент является правым ребенком, то надо подниматься по дереву вверх, пока текущий элемент не станет левым ребенком (строки 4,5). Строка 6 – проверка, есть ли родитель, и является ли текущий элемент левым ребенком. Если текущий элемент — левый ребенок, то следующий элемент — родитель (строка 7), иначе поднимаемся по дереву выше. Для данного элемента предыдущим по возрастанию является самый правый элемент левого поддерева либо ближайший левый предок:

```
1.  GetPrev(BTreeElm e){
2.     if(e.left != null)
3.        return getMax(e.left);
4.     while(e.parent != null && e.parent.left == e)
5.        e = e.parent;
6.     if(e.parent != null && e.parent.right == e)
7.        return e.parent;
8.     return null;
9.  }
```

Упражнение 17. *Напишите программу, которая печатает элементы дерева в возрастающем (убывающем) порядке.*

2.4.5 Удаление элемента из бинарного дерева поиска

В зависимости от расположения элемента в дереве алгоритм удаления будет разным. Есть три основных случая (рис.2.7):

- Элемент детей не имеет. Тогда просто берем и удаляем его (обнуляем ссылку на него у родителя).

- Элемент имеет одного ребенка. Элемент удаляется, а его ребенок усыновляется его родителем (переключаются ссылки)
- Элемент имеет двух детей. Находим следующий элемент (см. `GetNext`). Его удаляем со старого места и вставляем на место удаляемого элемента. (переключаем соответствующие ссылки).

Упражнение 18. *Почему в третьем случае следующий элемент всегда существует*

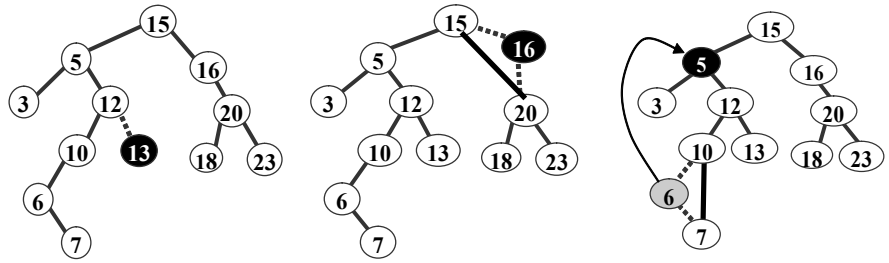


Рис. 2.7: Удаление элемента. Удаляемый элемент показан черным. Пунктиром показаны связи, которые исчезают, толстой линией — новые связи. Слева — удаления элемента без детей, в середине — удаление элемента, имеющего одного ребенка. Справа — удаление элемента, имеющего двух детей.

Псевдокод для удаление элемента:

```
1.  remove(BTreeElm e){
2.     BTreeElm x,y;
3.     if(e.left ==0 || e.right==0){
4.        y=e;
5.     }
6.     else
7.        y=getMin(e.right);
8.     if(y.left != null)
9.        x=y.left;
10.    else
11.       x=y.right;
12.    if(x!=null)
13.       x.parent=y;
14.    if(y.parent==null)
15.       root=x;
16.    else if(y==y.parent.left)
17.       y.parent.left=x;
18.    else
19.       y.parent.right=x;
20.    if(y!=e)
```

```
21.       e.data=y.data;
22.    return y;
23.  }
```

Поясним алгоритм. Идея заключается в том, чтобы перестроить дерево, а затем, если необходимо, переопределить данные на том месте, где был удаляемый элемент.

Строка 2 — создание двух временных элементов. y — это элемент, который встанет на место удаляемого элемента (строки 3-7). x — потомок элемента , который надо будет пересадить (строки 8-11). Отметим, что y имеет только одного потомка (**Упражнение** — почему?). Строки 14-15 обрабатывают удаление корня. Строки 16-19 пересаживают элемент x вместе с соответствующим поддеревом. Строки 20-21 восстанавливают данные в элементе e — просто переписывают данные. Поскольку данные, как правило, указатель, то это переопределение не требует много времени.

Подводя итог алгоритму удаления элементов, заметим, что в худшем случае удаление элемента сводится к поиску наименьшего элемента в правом поддереве, что занимает $O(h)$, где h — высота дерева. В двух других случаях удаление элемента будет стоить еще меньше — $O(1)$.

Бинарное дерево поиска		
Задача	Время	
	средн.	худш.
Вставка	$O(\log(n))$	$O(n)$
Удаление	$O(\log(n))$	$O(n)$
Поиск	$O(\log(n))$	$O(n)$

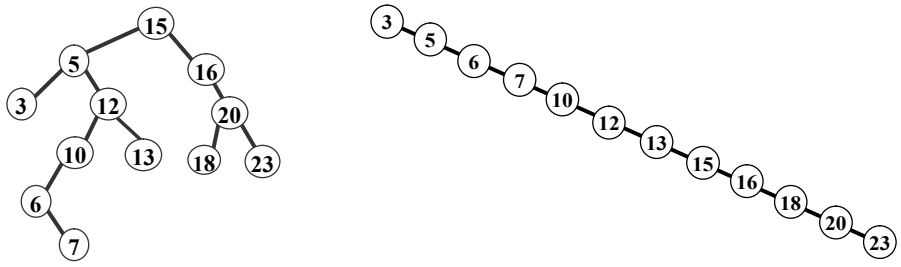


Рис. 2.8: Слева показано хорошо сбалансированное дерево (много узлов с двумя потомками), справа — несбалансированное дерево.

Время в худшем достаточно велико, поскольку в худшем случае дерево не сбалансировано (рис.2.8). Важно отметить, что оба дерева на этом рисунке соответствуют одному и тому же набору данных!

2.4.6 Преобразование деревьев поиска

Нужно помнить, что логарифмические времена поиска, вставки, удаления возможны только при условии "хорошего" дерева. Ясно, что для того, чтобы дерево поиска было эффективным, необходимо, чтобы оно имело небольшую высоту, и поэтому сильно ветвилось. Отметим, что деревья, представленные на рисунке содержат один и тот же набор данных. Это наводит на мысль, что, во-первых, существуют классы эквивалентности деревьев поиска, а, во-вторых, существуют преобразования, переводящие одно дерево поиска в другое, эквивалентное ему. Тогда, применив эти преобразования можно *сбалансировать* дерево. Такими преобразованием является вращение дерева.

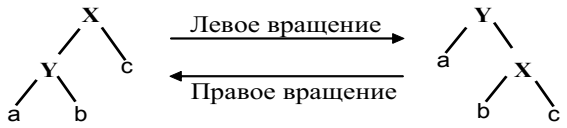


Рис. 2.9: Вращение деревьев поиска

Дано: некое бинарное дерево поиска с поддеревьями (см. рис.2.9) a, b и c. Узел Y имеет в качестве дочерних поддеревья a слева и b справа. Узел X имеет дочерние узлы: Y слева и c справа.

Требуется: поменять X и Y местами.

Решение: Так как данное дерево является бинарным древом поиска, то $Y < b < X$. Поэтому X может входить в правое поддерево вершины Y, а b — в левое поддерево вершины X, что не нарушит основное свойство бинарного дерева поиска. В связи с этим перестраиваем дерево таким образом (см. Рис.2.9).

Псевдокод алгоритма вращения:

```
1.  LeftRotate(X){
2.     Y=X.left;
3.     X.left= Y.right;
4.     if(X.parent !=null){
5.        if(X.parent.left==X)
6.           X.parent.left=Y;
7.        else
8.           X.parent.right=Y;
9.      }
```

```
10.     Y.parent=X.parent;
11.     X.parent=Y;
12.     Y.right=X;
13. }
```

```
1.  RightRotate(Y){
2.     X=Y.right;
3.     Y.right=X.letf;
4.     if(Y.parent != null){
5.        if(Y.parent.right==Y
6.        Y.parent.right=X;
7.     else
8.        Y.parent.left =X;
9.     }
10.    X.parent=Y.parent;
11.    Y.parent=X;
12.    X.left=Y;
13. }
```

2.4.7 Красно-черные деревья

Определение 1. *Дерево называется красно-черным (Red-Black tree, рис.2.10), если выполняются следующие условия:*

1. *дерево является бинарным деревом поиска*
2. *любая вершина имеет цвет — красный или черный*
3. *добавлены фиктивные листья, всегда черные. Поэтому любая настоящая вершина имеет двух детей*
4. *красная вершина имеет только черных детей*
5. *по пути от корня до любого листа количество черных вершин одинаково. Это число называется черной высотой дерева (bh).*

Из свойства (4)следует, что на любом пути длины l не может быть больше $l/2$ красных вершин. У красно-черного дерева любое поддерево красно-черное.

Лемма 1 (о высоте)**.** *Полная высота красно-черного дерева не превышает два логарифма числа вершин:*

$$h \leq 2 \cdot \log_2(n+1)$$

где h — высота дерева, n — число вершин дерева.

Доказательство. Докажем лемму методом математической индукции.

Основание индукции: для $n = 1$ утверждение верно: $1 < 2$

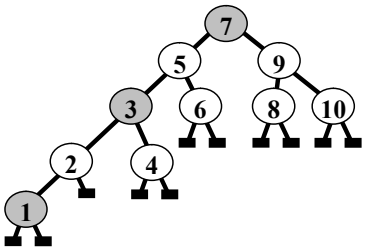


Рис. 2.10: Пример красно-черного дерева. "Красные" вершины отмечены серым. "черные" вершины отмечены белым. Чему равна черная высота?

Предположение индукции: Пусть для любого поддерева (левого и правого) утверждение леммы верно, т.е. $h \leq 2 \cdot \log_2(n+1)$, или $n \geq 2^{h/2}-1$.

Индукция: Пусть n_r — число вершин в правом поддереве, а n_l — в левом. $n = n_l + n_r + 1$ (число вершин дерева в точности равно сумме числа вершин в правом и левом поддеревьях плюс корень). Высота каждого поддерева не меньше, чем $h - 1$, иначе не выполнялось бы свойство одинаковости черной высоты по любому пути.

По предположению индукции для n_r и n_l верны оценки:

$$n_r \geq 2^{(h-1)/2} - 1, n_l \geq 2^{(h-1)/2} - 1$$

Отсюда получаем:

$$\begin{aligned} n &= n_l + n_r + 1 \geq 2^{(h-1)/2} - 1 + 2^{(h-1)/2} - 1 \\ &= 2 \cdot 2^{(h-1)/2} - 1 = 2^{(h-1)/2+1} - 1 = 2^{(h+1)/2} - 1 \\ &\geq 2^{h/2} - 1 \end{aligned}$$

или $h \leq 2 \cdot \log_2(n+1)$

□

Таким образом, время поиска, вставки и т.п. в красно-черное дерево всегда логарифмическое. Правда, после вставки или удаления может нарушиться одно из условий определения красно-черного дерева. Заметим, что "плохих" красно-черных деревьев не бывает. Нельзя сделать цепочку из красных вершин, а длинная цепочка из черных нарушит черную высоту. В этом преимущество красно-черных деревьев.

Поиск, вставка и удаление элементов из красно-черного дерева

Поиск элемента. Поиск элемента в красно-черном дереве ведется точно так же, как в обычном.

Добавление элемента. Новый элемент в черно-красное дерево добавляется так же, как и в любое бинарное дерево поиска и красится в красный цвет (чтобы не нарушать черную высоту). Если родителем нового элемента является черный элемент, ничего больше делать не нужно: черная высота не нарушена, дерево осталось красно-черным. Однако если родитель нового (красного) элемента тоже красный, возникает конфликт. Чтобы разрешить его и оставить дерево красно-черным, придется преобразовывать дерево — вращать и перекрашивать вершины. За добавлением новой красной вершины, пришедшейся "не к месту следует поэтапная перекраска дерева, сопровождающаяся вращениями. У элемента могут быть "дальний"и "ближний"дядя. Проще всего объяснить, чем ближний дядя отличается от дальнего, на рисунке 2.11: В зависимости от цвета "дядей"добавленного

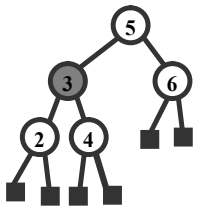


Рис. 2.11: Схема красно-черного дерева. Кружками показаны настоящие, квадратиками — фиктивные листья. Черная высота дерева — 3. Элемент 6 для элемента 4 является *ближним дядей*, а для элемента 2 — *дальним дядей*.

элемента указанные элементы перекрашиваются определенным образом. К примеру, если дальний дядя пришедшего элемента черный, вращаем дерево на уровне родителя нового элемента влево (рис. 2.12). Перекрасить

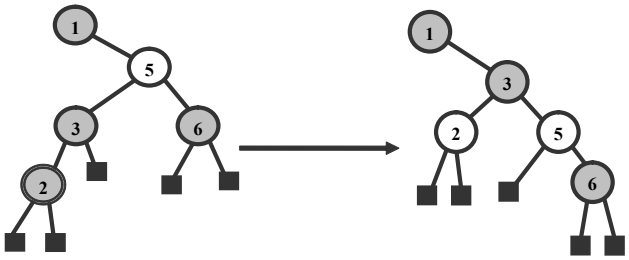


Рис. 2.12: Один из примеров перемещения конфликта. Белые квадраты означают красно-черные поддеревья, причем их высота такова, чтобы не нарушать основное свойство красно-черного дерева. Добавлен элемент 2. После вращения элемент 2 перекрашивается. Основное свойство сохранено.

дерево так, чтобы оно стало красно-черным, мы пока не можем, поскольку

ку не знаем, что находится выше. Тем не менее, после вращения конфликт перешел один на уровень вверх. Поэтапно вращая и перекрашивая участки дерева, можно "отогнать" конфликт к корню дерева. В корне конфликт легко разрешается перекрашиванием корня в черный цвет. При этом дерево опять становится красно-черным, а его черная высота либо остается прежней, либо становится на единицу больше по сравнению с той, что была до добавления элемента.

Пусть в нашем примере фигурирует не поддерево, а целое дерево. Предыдущим вращением конфликт уже переведен на уровень корня, поэтому можно начинать перекрашивать. Корень красим в черный цвет, и конфликт исчерпан. На предыдущих этапах мы этого сделать не могли, поскольку нарушился бы баланс между левым и правым поддеревьями у корня. Существует много различных вариантов вращения и перекрашивания в зависимости от цвета родителя и "дядей". Детали этого процесса мы опустим и отправим читателя к специальной литературе. В любом случае смысл изменений дерева состоит в перемещении конфликта (красная вершина имеет красного ребенка) к корню древа, где его можно спокойно разрешить.

Удаление элемента. Удаление красной вершины ничего не меняет, удаление черной вершины влияет на черную высоту, и кроме того, может привести к появлению красного ребенка у красного родителя. Поэтому вершину, вставшую на место удаленной черной "докрашиваем черным": красная вершина становится черной, а черная — "дважды черной". Такая вершина при подсчете черной высоты считается два раза. Далее вращаем и перекрашиваем, отгоняя дважды черную вершину к корню. Там ее объявляем однажды черной, на чем конфликт и исчерпывается.

Красно-черное дерево	
Вставка	$O(\log(n))$
Удаление	$O(\log(n))$
Поиск	$O(\log(n))$

2.5 Хеш-таблицы

Обращение к элементу массива происходит за время O(1). Представим себе, что мы как-то пронумеровали ВСЕ возможные ключи. Тогда процедура добавления, поиска и удаления элемента была бы простой — достаточно по ключу определить номер и посмотреть есть ли в соответствующем элементе массива данные, или положить новые данные, или удалить данные, которые там уже лежат. Беда только в том, что, как правило, множество возможных ключей очень велико и нам никогда не удастся построить массив такого размера с массой пустых ячеек. Тем не менее, допустим, у нас есть отображение g множества ключей на множество целых не отрицательных чисел:

$$g : key \to N$$

Это отображение называется хеш-функцией. Во многих случаях можно построить такое отображение. Например, есть простой способ отображения строк на пространство целых чисел. Каждый символ кодируется, скажем, байтом. Тогда можно написать:

$$g(s) = s[0] + 256 \cdot s[1] + 256_2 \cdot s[2] + ...$$

Разумеется, ни для какой строки длиннее 4 мы не сможем уместить такое число в переменную типа `int` и нельзя даже представить себе массив, способный хранить целочисленные образы всех строк. Если же вместо отображения на пространство всех чисел использовать отображение (не обязательно взаимно-однозначное) на небольшое подмножество множества целых неотрицательных чисел:

$$h : key \to 0, 1, ...m - 1$$

то для хранения данных можно было бы использовать массив размера m. Данные можно хранить в простой таблице, которая каждому значению хеш-функции ставит в соответствие какие-то данные. Однако, поскольку несколько ключей могут соответствовать одному хеш-значению, возможно возникновение *коллизий*, т.е. ситуаций, когда несколько ключей захотят записаться в одну клетку таблицы. Для выхода из этой ситуации в ячейках таблицы лежат не просто ключи, а связный список ключей.

Добавление элемента заключается в вычислении хеш-функции, затем происходит обращение к ячейке таблицы, и, если она пустая, то создается связный список из одного элемента, если там список уже есть, то добавляем элемент к списку.

Поиск. Вычисляем хеш-функцию, и обращаемся к соответствующей ячейке таблицы. Далее ищем в связном списке, лежащем в этой ячейке.

Удаление. Сначала находим элемент в связном списке, затем его удаляем из связного списка.

Эффективность работы хеш-таблицы определяется хеш-функцией. Хорошая хеш-функция должна удовлетворять двум основным требованиям: она должна быстро вычисляться, и она должна порождать хорошие ключи для равномерного распределения элементов по таблице. Поговорим о втором условии. С введением хеш-функций решилась проблема переполнения памяти: хеш-значений ограниченное число (m), а проблема коллизий решена с помощью связного списка. Обычно длина такого списка невелика и не превышает числа строк в хеш-таблице. Если хеш-функция достаточно хорошая, списки в ячейках короткие. Иначе элементы с большой вероятностью попадать в одно и то же место таблицы.

Приведем примеры хеш-функций:

- Вырожденная функция с одним хеш-значением :

$$h(key) \equiv 1$$

Понятно, что такая функция неприменима. В одной ячейке с хеш-значением 1 будет находиться длинный список из всех элементов. Это сводит хеш-таблицу к связному списку.

- Если у нас определена функция отображения множества ключей на множество неотрицательных целых чисел, то можно предложить такую хеш-функцию:

$$h(key) = g(key)\%m$$

где % обозначает остаток от деления. В таком случае, чем больше число n, тем короче будут списки в каждой ячейке. Понятно, что принимает значения от 0 до m , поэтому чем меньше число , тем меньше возможных хеш-значений, тем больше элементов соответствует каждому из них. Однако делать n слишком большим нельзя — это элиминирует все преимущества хеш-таблицы над простой таблицей ключей.

Вопрос построения эффективных хеш-функций является предметом достаточно сложных математических исследований. Важнейшим свойством правильной хеш-функции является равновероятность появления хеш-значений на множестве ключей. Поэтому для построения правильной хеш-функции необходимо знать распределение вероятностей появления тех, или иных ключей. Если хеш-функция равномерная, то эффективность основных операций (добавление, удаление, поиск)) в среднем определяется коэффициентом заполненности таблицы $\alpha = n/m$, где n — размер таблицы, а m — максимальное значение хеш-функции h: $T = O(1 + \alpha)$. Единица в оценке — определяет время вычисления хеш-функции. Вообще говоря, это время зависит от размера ключа. Например в случае если в качестве ключей используются строки, то время вычисления хеш-функции может зависеть от размера строки (ключа).

Упражнение 19. *Напишите основные алгоритмы работы с хеш-таблицами.*

Связанная с биоинформатикой хеш-функция: отображает сочетание двух букв, обозначающих нуклеотиды, в число от 0 до 15

хеш-функция

aa	0	ga	8
ac	1	gc	9
ag	2	gg	10
at	3	gt	11
ca	4	ta	12
cc	5	tc	13
cg	6	tg	14
ct	7	tt	15

Пусть данные data — это адрес начала определенного динуклеотида в нуклеотидной последовательности, а ключ key — этот динуклеотид. Тогда при данной последовательности

g	a	t	c	g	c	t	c	c	g	t	c	t	g	c	t	a	c	t	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Хеш-таблица будет иметь такой вид:

hash	data	hash	data
0		8	0
1	16	9	$4 \leftrightarrow 13 \leftrightarrow 19$
2		10	
3	1	11	9
4		12	15
5	7	13	$2 \leftrightarrow 6 \leftrightarrow 10$
6	$3 \leftrightarrow 8$	14	$12 \leftrightarrow 18$
7	$11 \leftrightarrow 14 \leftrightarrow 17$	15	

Рассмотрим один пример. Заполнение таблицы. Первому динуклеотиду `ga` соответствует хеш-значение 8. Кладем в таблицу адрес 0. Второму динуклеотиду `at` соответствует ключ 3. Кладем в таблицу адрес 1. И так далее, пока не дойдем до конца последовательности. Если возникает коллизия (например, для динуклеотида `tc`), то организуем связный список. Треть ячеек в ней оказалась пустой, в некоторых ячейках возникла коллизия. Длина последовательности небольшая, поэтому и в каждую ячейку попало всего по несколько элементов. Число хеш-значений данной хеш-функции ограничено (16), поэтому при работе с реальными последовательностями длина связных списков в каждой ячейке будет велика.

Теперь эту конструкцию можно использовать для поиска. Если мы хотим найти динуклеотид ct (его позиции). Мы берем ключ (строку "ct", обозначающую динуклеотид) и находим ее хеш-значение (7). В ячейку хеш-таблицы с хеш-значением 7 находим связный список со всеми координатами.

В связи с этим в реальной работе предпочтительнее использовать хеш-функции с большим числом значений. При работе BLAST задействована хеш-функция, отображающая не ди-, а 12-нуклеотиды на пространство целых чисел. У такой функции 4^{12} значений, что оптимально для работы с большими нуклеотидными последовательностями. В случае аминокислотных последовательностей используется хеш-таблица, основанная на триплетах аминокислотных остатков. Современный BLAST использует достаточно умную схему хранения данных и схему хеширования, которые позволяют достичь высокой эффективности.

2.6 Заключительные замечания

В таблице 2.2 показаны сравнительные характеристики рассмотренных структур данных, в том числе "накладные расходы" на хранение дополнительных указателей, максимальные минимальные и средние времена выполнения операций.

Таблица 2.2: Времена работы основных действий для различных структур данных

	Доп.	Поиск			Добавление			Удаление		
	указат.	сред	Max	Min	сред	Max	Min	сред	Max	Min
Простой массив	0	n	n	1	1	1	1	n	n	1
массив	0	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	1	n	n	1
Односвязный список	1	n	n	n	1	1	1	n	n	1
Двусвязный список	2	n	n	n	1	1	1	1	1	1
Двоичное дерево поиска	3	$\log n$	n	1	$\log n$	n	1	$\log n$	n	1
Красно-черное дерево	3	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
Хеш-таблица	2	1	n	1	1	n	1	1	n	1

Глава 3

Алгоритмы для строк

3.1 Определения, постановка задачи. Наивный алгоритм

В биоинформатике (и не только) очень часто встречается задача поиска подстроки. Сначала дадим несколько определений.

Строка S — массив символов $s_1, s_2, ..., s_{n-1}$, принадлежащих некоторому алфавиту A.

Подстрока $S[i, j]$ — фрагмент массива $s_i, s_{i+1}, ..., s_{j-1}$.

Префикс $S[i$ — подстрока $S[0, i]$.

Суффикс $S]i$ — подстрока $S[i, n]$.

Очевидно, что строка является своим префиксом и суффиксом. Это тривиальные префикс и суффикс (см. рис. 3.1). Любой нетривиальный префикс или суффикс называется еще *собственным*.

```
a a b a
a a b a c a
a a b a c a c b b a
a a b a c a c b b a c a b a c
      a c a c b b a c a b a c
                  c a b a c
                        a c
```

Рис. 3.1: Префиксы суффиксы. Верхние 3 строки показывают префиксы, нижние — суффиксы

Задача поиска образца. Даны две строки T (текст) и P (образец, паттерн). Нужно узнать, встречается ли подстрока, такая же, как P, внутри T, иными словами, найти хотя бы одно точное вхождение P в T. Существует также задача поиска числа точных вхождений P в T. Итак:

Дано: длинная строка $T = t_0, t_2, ..., t_{n-1}$ (текст) и короткая строка (образец, паттерн) $P = p_0, p_2, ..., p_{m-1}$, $m \leq n$.

Требуется найти: непрерывный фрагмент строки T, полностью совпадающий с P, т.е. подстроку $T[i, i+m]$, такую что $t_i = p_0, t_{i+1} = p_1, ..., t_{i+m-1} = p_{m-1}$

Наивный алгоритм: идем по сроке T, и начиная с каждой новой позиции, проверяем, не начало ли это нужной нам подстроки. Ниже приведена процедура `SubStrCmp` — процедура сверки подстроки $T[i, i + p.length]$ с образцом P. Если эта подстрока равна P, возвращается `true`, если хоть один символ отличается — возвращается `false`

```
1.  SubStrCmp (String t, int i, String p){
2.     for (j=0; j<= Length(p);  j++){
3.        if (t[i+j] != p[j]) return false;
4.        else return true;
5.     }
6.  }
7.  NaiveSearchPattern (String t, String p){
8.     for(int i=0; i<Length(t)-Length(p)+1; i++){
9.        if (SubStrCmp (String t, int i, String p)) return i;
10.       else return "Not found";
11.    }
12. }
```

Описанный алгоритм работает просто, но долго. В наихудшем случае наивный алгоритм поиска образца по строке работает за $O(m \times n)$ операций. Таких случаев можно придумать несколько, например для строки и образца

T="aaaaaaaaaaaaa"
P="aaaaax"

T не содержит ни одного вхождения P, но включает много похожих подстрок, отличающихся от p только последним символом. При поиске процедура `SubStrCmp` постоянно проходит почти всю длину, обрываясь постоянно на последнем символе, что занимает много времени. С учетом того, что роль строки T в бионформатических задачах часто играют нуклеотидные последовательности, не отличающиеся особой краткостью, время работы алгоритма может быть велико.

Оценим *среднее время* работы алгоритма на строке T длиной n с образцом P длиной m. Среднее время поиска равно

(количество вызовов `SubStrCmp` $= n - m$)$\times$
(среднее время исполнения `SubStrCmp`).

Для оценки среднего времени работы процедуры SubStrCmp можно применить схему Бернулли. Элементарным событием будет служить сравнение каждого следующего символа строки T с каждым следующим символом

образца P. При этом условие независимости элементарных событий выполнено: каждое следующее сравнение не зависит от предыдущего. У любого события возможны только два исхода: успех (символы совпали) и неудача (символы не совпали). Среднее время работы `SubStrCmp` зависит от параметра p — вероятности успеха, равной для каждого элементарного события. Пусть:

p — вероятность совпадения символов

$q = 1 - p$ — вероятность несовпадения символов

Время работы процедуры зависит от числа шагов. Работа процедуры обрывается при первом несовпадении, и чем больше совпадений идет подряд до первого несовпадения, тем дольше работает процедура. Вероятность того, что работа `SubStrCmp` обрывается на первом шаге, равна q(вероятности несовпадения символов в первой позиции). Вероятность обрыва на втором шаге равна pq (чтобы оборваться на втором шаге, нужно пройти первый шаг, то есть на первом шаге должно быть совпадение, вероятность чего равна p). Таким образом, среднее время работы процедуры равно $q + 2pq + 3p^2q + \ldots + (m-1)qp^{m-2} + mp^{m-1}$. Подобную сумму мы уже вычисляли при анализе поиска в массиве:

$$E = \frac{1 - p^m}{1 - p}$$

Предположим, что появление букв в тексте и в образце равновероятны. Тогда вероятность совпадения символов из строки T и образца P равна $1/\|A\|$ где A — алфавит, к которому относятся символы, входящие в T и P, $|A|$ — мощность алфавита A. Если алфавит большой, значение p маленькое. Тогда величиной p^m можно пренебречь, поэтому средняя скорость работы `SubStrCmp` имеет порядок

$$T(SubStrCmp) \approx \frac{p}{q} = O(1)$$

При выводе этой формулы мы делали следующие предположения:

- Строки T и P независимы
- Алфавит A имеет состоит из достаточно большого числа символов (имеет большую мощность)

Поскольку нам надо применить процедуру `SubStrCmp` n раз, среднее время наивного поиска, равно $O(n)$.

Из выведенной формулы понятно, что чем больше мощность алфавита и чем меньше длина строки, в которой мы ищем, тем быстрее (в среднем) идет поиск. Однако в реальности далеко не всегда частоты встречи символов одинаковые, кроме того строки могут быть и плохими (как в приведенном примере), т.е. условия поставленной задачи не обязательно удовлетворяют данным предположениям. Поэтому нужно искать другие алгоритмы.

3.2 Алгоритм Рабина-Карпа

Пусть алфавит состоит из цифр. Тогда любую стоку можно представить в виде целого числа. Для того, чтобы сравнить две строки надо просто сравнить эти целые числа. Если строки не слишком длинные, то сравнение двух чисел можно сделать за одну операцию. Надо только преобразовать строку цифр в число:

$$N(\alpha_0 1 \alpha_2 \dots \alpha_{n-1}) = \alpha_0 \cdot A^{n-1} + \alpha_2 \cdot A^{n-2} + \dots + \alpha_{n-1} \cdot A^0$$

где A — размер алфавита. Например, преобразование строки "3243" в число будет выглядеть так:

$$N(3243) = 3 \cdot 10^3 + 2 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0$$

Алгоритм поиска подстроки можно представить теперь так: преобразовываем образец P в число. Потом каждую подстроку текста T преобразовываем в число и сравниваем числа. Видно, что преобразование последовательности цифр в число требует времени $O(L)$, где L — длина последовательности. Поэтому пока идея не работает — общее время поиска образец будет требовать времени порядка $O(m \times n)$. Однако есть одно соображение, которое позволяет сделать алгоритм более эффективным. Пусть нам известно преобразование подстроки, начинающейся с позиции i: $N(i) = N(t_i t_{i+1} \dots t_{i+m})$. Тогда можно легко вычислить $N(i+1)$. Для этого надо отбросить первую цифру и дописать цифру после последней:

$$t_0 t_1 \dots \overbrace{t_i \underbrace{t_{i+1} \dots t_{i+m-1}}_{N(i+1)} t_{i+m}}^{N(i)} \dots t_{n-1}$$

Чтобы отбросить первый символ числа-строки надо взять число по модулю A^{n-1}, чтобы добавить символ справа надо умножить число на A и прибавить число, соответствующее добавляемому символу:

$$N(i+1) = (N(i)) \,\%\, \left(|A|^{n-1}\right) \cdot |A| + N(t_{i+m}) \tag{3.1}$$

Здесь мы для mod использовали обозначение %, принятое во многих языках программирования. Таким образом алгоритм становится более эффективным. Сначала вычисляем числа, соответствующие образцу P и префиксу текста $T[1, m]$. Затем в цикле сравниваем числа, потом сдвигаем в соответствии с 3.1 и повторяем сравнение.

```
1.  int StrToNum(String p, int l){
2.     n=0;
3.     for(i=0; i<l; i++){
4.        n=n*A+toInt(p[i]);
5.        return n;
6.     }
```

```
7.  }

8.  int RabinMatch(String t, String p){
9.     M=StrToNum(p,Length(p));
10.    N=StrToNum(t,Length(p));
11.    nn=power(A,Length(p)-1);
12.    for (i=0; i<Length(t)-Length(p); i++){
13.       if(M==N) return i;
14.       N=(N%n)*A + toInt(t [i]);
15.    }
16.    return "Not found";
17. }
```

Пояснения к алгоритму. Строки 1-7 вычисляют число по строке длины l. A — размер алфавита. Строки 9 и 10 вычисляют числа для образца P и префикса $T[0, n]$. Дальше в цикле происходит сравнение чисел (строка 13) и пересчет числа (строка 14)

Описанный алгоритм, носит имена Рабина и Карпа. Нетрудно заметить, что время поиска не зависит от длины образца, каждое следующее число получается за время порядка $O(1)$ и поэтому время работы имеет порядок $T = O(m + n)$.

Ограничения работы алгоритма: Алгоритм нельзя использовать на больших образцах, поскольку соответствующее число может "не вместиться" ни в один тип данных, и на больших алфавитах, так как невозможно будет отобразить буквенную строку в цифровую той же длины. Алгоритм Рабина-Карпа очень хорош для поиска в нуклеотидных последовательностях — алфавит содержит 4 буквы, поэтому использование int позволяет искать образцы длиной до 16, а применение long — до 32. Кроме того операции $\cdot$, /, % заменяются на соответствующие логические операции и сдвиг, которые исполняются гораздо быстрее, чем умножение, деление и вычисление остатка.

3.3 Алгоритм Кнута-Морриса-Пратта

Вспомним сложный случай, на котором наивный поиск работает максимальное время:

Строка P	`aaabaaaaaaaaaaabcaaaaaaadaaaaaaaaaaaaaaa`
Паттерн T	`aaaax`

Представим, что мы в процессе поиска мы получили наложение:

```
a a a b a a a a  a a a a a a b
        a a a a  x
```

Шли, шли и "потерпели неудачу" на букве x; нужно сдвигаться и искать дальше. Но ведь понятно, что проверять сходство первых букв, при следующем сдвиге не нужно — они заведомо совпадут, а мы проделаем лишнюю работу. Или другой пример:

```
a a a b a b a b a a a a a a b
      a b a b a x a
```

Видно, что делать следующую поверку, сдвинувшись на один символ, бессмысленно — совпадения не будет; для следующей попытки правильно сдвинуть на 2.

Получается, что, глядя заранее на *структуру* паттерна, можно сказать на какое количество символов нужно сдвигаться после "неудачи". Если q первых символов совпало, а в позиции $q+1$ есть несовпадение, то из анализа только образца можно сказать насколько имеет смысл сдвигать образец.

3.3.1 Префикс-функция $sp_i(P)$

Определение 1. *Префикс-функцией $sp_i(P)$ называется длина наибольшего собственного суффикса подслова $P[0, i-1]$ совпадающего с префиксом P:* $P[0, sp_i] = P[i - sp_i, i]$.

Пример: возьмем слово abcabcabba и найдем все sp_i

$sp_0 = 0$		abcabcabba
$sp_1 = 0$		a bcabcabba
$sp_2 = 0$		ab cabcabba
$sp_3 = 0$		abc abcabba
$sp_4 = 1$	подслово	abca bcabba
	префикс	abca
	суффикс	abca
$sp_5 = 2$	подслово	abcab cabba
	префикс	abcab
	суффикс	abcab
$sp_6 = 3$	подслово	abcabc abba
	префикс	abcabc
	суффикс	abcabc
$sp_7 = 4$	подслово	abcabca bba
	префикс	abcabca
	суффикс	abcabca
$sp_8 = 5$	подслово	abcabcab ba
	префикс	abcabcab
	суффикс	abcabcab
$sp_9 = 0$		abcabcabb a
$sp_{10} = 1$	подслово	abcabcabba
	префикс	abcabcabba
	суффикс	abcabcabba

Отметим некоторые особенности префикс-функции для этого примера. При размере префикса 0 значение sp_0 равно 0, поскольку если нет префикса, то и длина максимального суффикс-префикса равна 0. При размере префикса 1 sp_1 также равен 0, поскольку нас интересуют только собственные суффиксе-префиксы. Это выполняется для *любого* образца. Для префиксов размером от 4 до 8 увеличивается от 1 до 5. Это связано с тем, что образец содержит точный повтор. Для префикса длиной 9 нет суффикс-префикса, так что sp_9 равен 0. При длине префикса 10 sp_{10} равен 1. Отметим, что для случая sp_7 и sp_8 суффиксы пересекаются с префиксами. Это допустимо.

3.3.2 Алгоритм поиска образца

Лемма 1. *Если при наложении образца на текст первое несовпадение обнаружилось в позиции i образца, то мы можем безопасно сдвинуть образец на величину*

$$shift = i + 1 - sp_i$$

Доказательство. Приведенное утверждение доказывается от противного. Предположим, что мы передвинули образец на число символов $k < i+1-sp_i$, и при этом произошло совпадение (см. Рис3.2): Это значит, в частности,

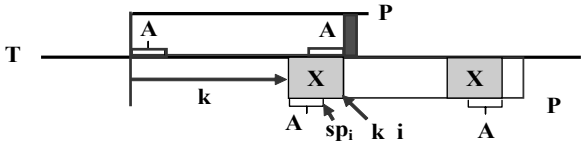


Рис. 3.2: К доказательству безопасности сдвига по Кнуту-Моррису-Пратту. A — суффикс-префикс

что фрагмент $X = P[0, i-k]$ совпадает с текстом. Но при предыдущем наложении у нас совпал без ошибок фрагмент образца $P[0, i]$. Поэтому слово X совпадает с фрагментом $P[k, i]$. Длина этого слова равна $i-k+1 > i-1+sp_i-i+1 = sp_i$. Это противоречит определению sp_i как *максимального* суффикса-префикса. Противоречие доказывает утверждение. □

Очевидно, что после сдвига нет необходимости снова проверять символы, принадлежащие соответствующему суффикс-префиксу, поскольку они совпадают согласно определению. Таким образом, мы на каждый символ текста T смотрим не более двух раз. Если префикс-функция определена, то время работы алгоритма равна $O(|T|)$. Осталось только найти префикс-функцию. Этим занимается так называемый препроцессинг.

Упражнение 20. *Напишите псевдокод для алгоритма Кнута-Морриса-Пратта поиска образца, при условии, что массив sp дан.*

3.3.3 Препроцессинг Кнута-Морриса-Пратта

Поиск $sp_i(P)$ каждый раз, когда мы сдвигаем паттерн относительно строки занимает много лишнего времени. Правильнее перед поиском подстроки, равной образцу, сделать препроцессинг — найти все $sp_i(P)$ для этого образца и записать в таблицу.

Наивный алгоритм препроцессинга:

```
1.  int[] Naive_Spi_Search(String P){
2.     for(i=1;i<p.length,i++) {
3.        len = 0;
4.           l = i-1;
5.           while(l > 0 && len==0){
6.              s = substring(P,i-l,i);
7.              f = substring(P,0,l);
8.              if(s==f)
9.              len=l;
10.             l--;
11.          }
12.       sp[i]=len;
13.     }
14. }
```

Для каждого значения i (размера префикса) мы определяем максимальный суффикс-префикс. При этом начинаем искать с самых длинных возможных sp. Строка 4: если бы мы начинали цикл с i, все найденные максимальные префиксы, совпадающие с суффиксами, были бы тривиальными. Строка 5: мы сокращаем размер sp. Если на какой-то итерации величина len стала отличной от нуля, то это значит, что мы нашли sp и искать более короткие суффикс-префиксы не имеет смысла. В строке 8 мы сравниваем суффикс и префикс. Алгоритм работает кубическое время, поскольку цикл строки 2 крутится m раз, цикл строки 5 крутится также порядка m раз (в среднем $m/2$). Внутри этого цикла есть сравнение строк (строка 8), которое также требует времени порядка m. Таким образом, время работы наивного алгоритма имеет порядок $O(m^3)$, то есть время работы кубическое.

Столь долго работающий алгоритм мы с негодованием отвергаем. Есть способ найти все sp_i для строки длиной m за линейное время $O(m)$. "Умный" алгоритм препроцессинга не "смотрит" многократно на каждую букву строки.

Лемма 2. *Отметим два свойства суффикс-префикса. Во-первых, $ps_i < i$. Это следует из того, что sp — собственный суффикс-префикс. Во-вторых, если l — длина какого-либо (не обязательно максимального) суффикс-префикса подслова $P[0, i]$, то sp_l определяет длину другого суффикс-префикса этого же подслова. Более того, не существует суффикс-префикса с длиной меньше l и больше sp_l.*

Доказательство. Действительно, поскольку есть суффикс-префикс длиной l, то верно $P[0,l] = P[i-l,i]$, но sp_l — означает, что существует суффикс-префикс у подслова $P[0,l]$, т.е. $P[0,sp_l] = P[l-sp_l,l] = P[i-l,i-sp_l] = P[l-sp_l,l]$ (см.рис 3.3). Второе утверждение следует из условия максимальности суффикс-префикса. □

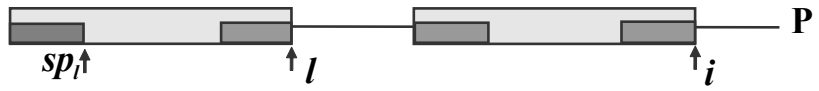


Рис. 3.3: К доказательству леммы 2. Поскольку есть суффикс-префикс, то по определению есть совпадающие фрагменты (светло-серые). Поскольку sp_l есть длина суффикс-префикса, то темно-серые прямоугольники совпадают. А это значит, что существует суффикс-префикс длины sp_l.

Следствие 1. *$sp(sp(\cdots sp(sp(i))))$ являются суффиксами-префиксами и других суффиксов-префиксов нет.*

Представим, что мы нашли sp_j для всех префиксов длиной меньше $j < i$. Тогда не представляет труда найти sp_i . Действительно, если $P[sp_i] = P[i]$, то $sp_i = sp_{i-1} + 1$. Если же совпадения нет, то воспользуемся леммой, и посмотрим на более короткие суффиксы-префиксы. Поэтому надо проверить совпадают ли $P[sp(sp(i-1)+1]$ с $P[i]$. Если совпадает, то $sp(i) = sp(sp(i-1))+1$; Если не совпадает, то надо смотреть предыдущий суффикс-префикс и т.д. (см. рис.3.4. Алгоритм препроцессинга.

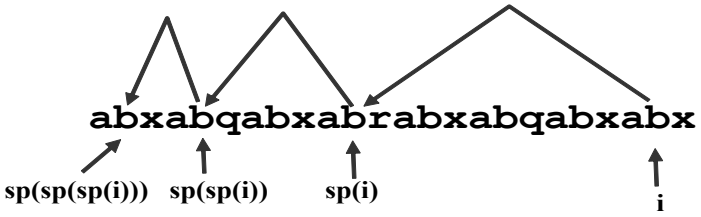


Рис. 3.4: Иллюстрация препроцессинга.

```
1.  KMP_Preprocessing(String p){
2.     int[] sp=new int[p.length+1];
3.     for (int j=1; j<p.length-1; j++){
4.        x=p.charAt(j);
5.        v=sp [j];
6.        while(p.charAt(v+1)!=x && v!=0){
7.           v=sp [v];
8.           if (v>0 && p.charAt(v+1)==x)
```

```
9.              sp[j+1]=v+1;
10.          else
11.             sp[j+1]=0;
12.       }
13.    }
14. }
```

Можно строго доказать, что время работы алгоритма препроцессинга линейно по длине образца $T = O(|P|)$. Таким образом, время работы алгоритма Кнута-Морриса-Пратта:

$$T_{KPM} = O(|T| + |P|)$$

3.4 Конечные автоматы

Определение 2. *Детерминированным конечным автоматом называют следующее множество:*

$$\{Q, q_0 \in Q, A \subseteq Q, \Sigma, \delta\}$$

где Q — конечное множество, называемое множеством ***состояний*** *конечного автомата; q_0 — некое выделенное состояние, называемое* ***начальным состоянием*** *конечного автомата; A — подмножество множества состояний (финальное или* ***допускающее множество состояний****); Σ — конечный алфавит; $\delta : Q \otimes \Sigma \to Q$ — **функция перехода**, которая отображает множество пар вида "символ-состояние" на множество состояний. Аргументами этой функции служат некий символ алфавита и состояние конечного автомата, значением — новое состояние конечного автомата.*

Автомат начинает работать из начального состояния q_0. Далее автомат считывает по одному символу из входной строки. Полученный на вход символ изменяет состояние автомата на новое из конечного набора возможных состояний $Q = q_0, q_1, \cdots, q_m$ в соответствии с имеющейся функцией переходов δ. Как видно здесь ключевым элементом является функция переходов. Множество состояний, алфавит и функцию переходов можно представить в виде таблицы переходов:

	алфавит		
состояния	a	b	c
0^*	0	2	1
1	2	3	1
2	3	0	1
3^+	0	2	1
* — начальное состояние $^+$ — допускающее состояние			

Часто работу конечного автомата изображают в виде графа переходов (см. рис.3.5). Обратите внимание на то, что в конечном автомате из каждого состояния выходит столько стрелок, сколько букв в алфавите.

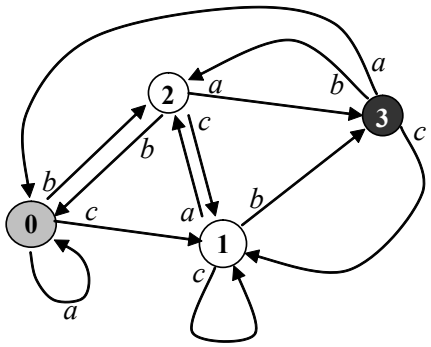


Рис. 3.5: Изображение конечного автомата в виде графа переходов. Начальное состояние 0 (выделено серым). Допускающее состояние 3 (темно-серое).

Пусть на вход автомата подается строка:

a	a	a	b	c	b	c	c	b	a	a	a	b	a	b	b	c	b	c	a	a	a	b	c	c	a

Автомат пройдет через следующую последовательность состояний. Начальное состояние 0 (по определению) первые три символа a не изменят состояния, поскольку по букве a переход происходит из состояния 0 в состояние 0. Затем автомат перейдет в состояние 2 - символ b переводит автомат из состояния 0 (текущее состояние) в состояние 2, и т.д. Последовательность состояний следующая:

a	a	a	b	c	b	c	c	b	a	a	a	b	a	b	b	c	b	c	a	a	a	b	c	c	a
0	0	0	2	1	3	1	1	3	0	0	0	2	3	2	0	1	3	1	2	3	0	2	1	1	2

Автомат 5 раз прошел через допускающее состояние. Вообще-то приведенный автомат не понятно зачем нужен. И то, что он несколько раз прошел через допускающее состояние нам ничего не говорит. На самом деле конечные автоматы — достаточно мощный инструмент для распознавания текстов. Как мы увидим позже, конечные автоматы можно применять для поиска сразу нескольких слов и для поиска регулярных выражений, которые составляют основу для разбора текстов.

Обозначим Σ^* множество всех слов, которые можно породить из алфавита. Функция $\varphi : \Sigma^* \to Q$ — *функция конечного состояния*: $\varphi(w)$ — конечное состояние автомата при предъявлении ему слова w (иными словами это состояние, в которое придет автомат после обработки w). Автомат допускает строку w тогда и только тогда, когда $\varphi(w) \in A$. Функцию φ можно определить *рекуррентно*:

- $\varphi(\varepsilon) = q_0$ — начальное состояние ε — пустое слово);
- $\varphi(w\mathtt{a}) = \delta(\varphi(w), \mathtt{a})$ для любых $w \in \Sigma^*, \mathtt{a} \in \Sigma$ (т.е. мы находились в состоянии $\varphi(w)$ и совершили переход в a.)

Ряд алгоритмов поиска подстрок начинаются с построения конечного автомата, который в некотором тексте T находит все вхождения образца P.

3.4.1 Поиск образца с помощью конечного автомата

Конечные автоматы используются для поиска подстроки в стоке. Они позволяют находить не только единичное вхождение образца в строку, но и фиксировать все подстроки, равные образцу, сколько бы их ни было.

Пример: Хотим найти в какой-либо строке паттерн вида abab. Пусть эта строка состоит только из символов "a" и "b", для построения конечного автомата будем пользоваться алфавитом из тех же символов. Конечный

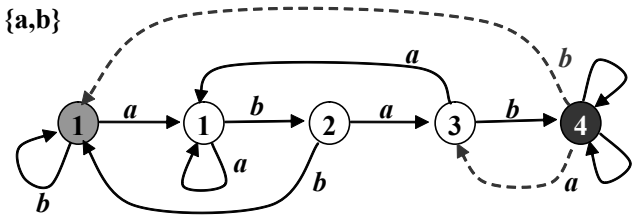


Рис. 3.6: Автомат для поиска образца abab. Автомат, включающий только черные стрелки, ищет первое вхождение образца abab в строку. Если в автомате заменить стрелки, исходящие из допускающего состояния на пунктирные стрелки.

автомат, изображенный на рис.3.6 был придуман напряжением мысли. А существует ли регулярный способ построения конечных автоматов для поиска вхождений образца? Первым шагом для построения автомата, соответствующего строке-образцу P является построение так называемое суффикс-функции.

Определение 3. *Суффикс-функцией называется вспомогательная функция $\sigma : \Sigma^* \to \{0, 1, \cdots, m\}$ — отображение слов на множество чисел. По определению, $\sigma(w)$ — длина максимального (не обязательно собственного) суффикса слова w, который совпадает с префиксом* образца P.

Важно, что для каждого образца своя суффикс-функция. что На первый взгляд суффикс-функция $\sigma(w)$ чем-то напоминает префикс-функцию sp, которую мы рассматривали в алгоритме Кнута-Морриса-Пратта. Однако есть существенная разница. Суффикс-функция определена на множестве всех слов, а префикс-функция определена на множестве чисел $\{0, 1, \cdots, m\}$,

поэтому их сравнивать бессмысленно. Ясно, что если $\sigma(w) = m$, то конец слова w полностью совпадает с образцом.

Простой пример суффикс-функции для образца $P = \texttt{ab}$:

$$\sigma(\varepsilon) = 0$$
$$\sigma(\texttt{accabca}) = 1$$
$$\sigma(\texttt{bcabcab}) = 2$$

Очевидные свойства суффикс-функции:

- если длина P равна m, то $\sigma(x) = m$ тогда и только тогда, когда P является суффиксом x
- если x является суффиксом y, то $\sigma(x) \leq \sigma(y)$.

Теперь определим конечный автомат, соответствующий образцу P, таким образом:

Множество состояний	$Q = \{0, 1, \cdots, m\}$
Начальное состояние	$q_0 = 0$
Единственное допускающее состояние	$A = m$
Функция переходов δ	$\delta(q, \texttt{a}) = \sigma(P_q a)$ (P_q — префикс длины q образца P)

Вот пример построения функции переходов для образца `abbabba`

P_i	P_ia	$\sigma(P_i a)$	P_ib	$\sigma(P_i b)$
abbabba	aa	1	ab	0
abbabba	aba	1	abb	3
abbabba	abba	4	abbb	0
abbabba	abbaa	1	abbab	5
abbabba	abbaba	1	abbabb	6
abbabba	abbabba	7	abbabbb	0
abbabba	abbabbaa	1	abbabbab	5

Докажем, что построенный конечный автомат действительно способен находить образец P и ничего больше. Для этого докажем пару лемм и теорему.

Лемма 1. $\sigma(x\texttt{a}) \leq \sigma(x) + 1$

Доказательство. Допустим $\sigma(x\texttt{a}) > \sigma(x) + 1$. Тогда отбросим последний символ от наибольшего суффикса $x\texttt{a}$, совпадающего с префиксом P. Тогда получим суффикс строки x, который длиннее $\sigma(x)$ и является префиксом P, что противоречит определению (рис.3.7). □

Лемма 2. *Пусть* $q = \sigma(x)$. *Тогда* $\sigma(x\texttt{a}) = \sigma(P_q\texttt{a})$.

Доказательство. Действительно, поскольку $\sigma(x\texttt{a}) \leq \sigma(x) + 1$, $\sigma(x\texttt{a})$ не изменится, если отбросить начало от строки $x\texttt{a}$, оставив только последние $q + 1$ символов (поскольку $q = \sigma(x)$), то $\sigma(x\texttt{a}) = \sigma(P_q\texttt{a})$ □

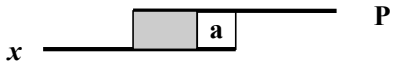


Рис. 3.7: К доказательству леммы 1

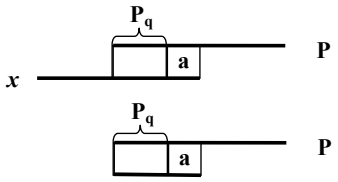


Рис. 3.8: К доказательству леммы 2

Теорема 1. *$\varphi(T_i) = \sigma(T_i)$, где T_i — суффикс слова T, начинающийся с индекса i.*

Доказательство. Доказательство по индукции. При $i = 0$ — очевидно. Покажем, что если утверждение теоремы верно при некотором i, то будет верно и при $i + 1$. Пусть $\mathtt{a}$ — это символ строки T в позиции i, то есть следующий за префиксом T_i символ. Тогда

$\varphi(T_{i+1}) = \delta\left(\varphi(T(i), \mathtt{a})\right)$	по свойству функции конечного состояния $\varphi(w\mathtt{a}) = \delta\left(\varphi(w), \mathtt{a}\right)$
$\delta\left(\varphi(T(i), \mathtt{a})\right) = \delta\left(\sigma(T_i), \mathtt{a}\right)$	по предположению индукции равенство $\varphi(T_i) = \sigma(T_i)$ верно для i
$\delta\left(\sigma(T_i), \mathtt{a}\right)) = \delta(q, \mathtt{a}) = \sigma(P_q\mathtt{a})$;	$q = \sigma(T_i)$ — обозначение для q. Функция переходов автомата по определению задается так: $\delta(q, \mathtt{a}) = \sigma(P_q\mathtt{a})$
$\sigma(P_q\mathtt{a}) = \sigma(T_{i+1})$	согласно лемме 2 о суффикс-функции

Таким образом доказано, что $\varphi(T_{i+1}) = \sigma(T_{i+1})$ □

Следствие 1. *Следствие. Автомат приходит в допускающее состояние, только если суффикс строки T_i совпадает с образцом P.*

Поиск подстроки с применением конечного автомата работает за время порядка $O\left(n + T(\sigma)\right)$, где n — длина строки, в которой ищем вхождение, $T(\sigma)$ — время построение суффикс-функции. Это немного больше, чем время работы алгоритма Кнута-Морриса-Пратта. Однако у конечных автоматов есть большое преимущество: они позволяют искать вырожденные или неточные вхождения.

3.4.2 Поиск регулярных выражений. Недетерминированные конечные автоматы.

Все изученные нами способы поиска подстроки (наивный алгоритм, алгоритмы Рабина-Карпа, Кнута-Морриса-Пратта, поиск с помощью конечных автоматов) служат для поиска точного вхождения в строку. Если же мы хотим найти не точный паттерн, а регулярное выражение, описывающее сразу несколько образцов? Регулярные выражения используются для сжатого описания некоторого множества строк с помощью шаблонов, без необходимости перечисления всех элементов этого множества. Пример регулярного выражения: `R = (abb)*(c)+(abc)`.

*	Символ или группа символов может появиться несколько раз или ни разу
+	Символ или группа символов должна появиться хоть один раз.
выражение, после которого не стоит никаких знаков	Символ или группа символов должна появиться ровно один раз.

Нашему регулярному выражению соответствуют строки `ccccabc` и `abbabc` и много других строк. Как найти любую из них пользуясь регулярным выражением?

Недетерминированный конечный автомат отличается от детерминированного тем, что результатом функции перехода является некоторое множество состояний (возможно пустое). Поэтому недетерминированный конечный автомат в каждый момент времени находится не в одном состоянии а в множестве состояний. Кроме того, переходы не обязательно определены для всех символов из входного потока. Вместо этого мы введем дополнительный переход ε, который отвечает "считыванию пустого символа", т.е. не снимает символ из строки. Распознавание образца состоит из двух этапов

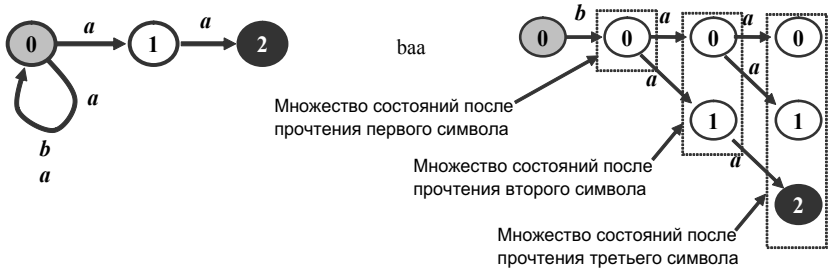


Рис. 3.9: Недетерминированный конечный автомат и множества состояний в процессе обработки строки baa.

— построение автомата и по образцу и работа автомата. При построении

автомата используется рекурсивное определение регулярного выражения R:

```
R::=<символ> | RR | (R) | R+ | R*
```

Эта формула означает, что регулярное выражение — это либо просто символ, либо регулярное выражение, за которым следует другое регулярное выражение, либо другое регулярное выражение заключенное в скобки, либо регулярное выражение, после которого идет знак +, либо регулярное выражение, за которым следует знак *. Такая рекурсия позволяет разложить регулярное выражения на элементарные составляющие. Для каждого составляющего можно построить конечный автомат, а затем эти автоматы можно объединять. При этом если один из элементов заканчивается символом *, то от начала (начального состояния) соответствующего автомата в его конец (во все допускающие состояния) идет ε-стрелка, что означает, что соответствующий фрагмент можно пропустить. Если после элементарного выражения стоит + или *, то из всех допускающих состояний идет ε-стрелка в начальное состояние, что означает, что группу можно повторить (рис. 3.10).

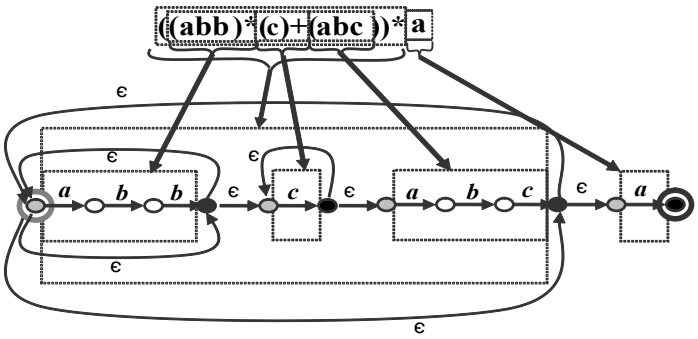


Рис. 3.10: Пример построения конечного автомата для регулярного выражения. Детали переходов в самых внутренних автоматах ОПУЩЕНЫ! Начальное и допускающие состояния общего автомата выделены двойными кругами. Начальные состояния — серые, допускающие — черные.

Теперь опишем, как этот автомат работает. При прочтении слова автомат может перейти в некоторое множество состояний. Введем некоторые обозначения:

N_0 — множество, куда можно попасть из начального состояния по стрелкам с буквой ε; N_{i+1} — множество состояний, в которое автомат может попасть из предыдущего множества состояний, прочитав $i+1$-ю букву строки T. Если на каком-то шаге автомат нашел соответствие (match), то множество состояний автомата имеет непустое пересечение с множеством допускающих состояний. Если очередное множество состояний пусто, то автомат отвергает строку. Разбор строки заключается в перестраивании множества

N_i. Ясно, что размер множества N_i не превышает длину образца, поэтому время разбора строки оценивается как $T = O(n \times m)$.

Минус описанного автомата — он ищет регулярное выражение только в префиксе строки. Чтобы искать регулярное выражение в любом месте строки нужно модифицировать регулярное выражение R: $R' = [\xi]R$, где ξ обозначает все буквы алфавита Σ, конечный автомат примет такой "кудрявый" префикс (рис. 3.11).

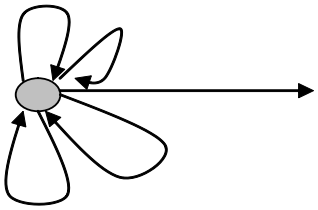


Рис. 3.11: Префикс текста для поиска образца с помощью недетерминированного автомата

3.5 Поиск многих образцов.Алгоритм Ахо-Корасик

А если перед нами стоит такая задача (ее еще называют задачей точного поиска или задачей поиска по групповому запросу): задано множество образцов $P = \{P_1, P_2, \cdots, P_z\}$. Требуется обнаружить все вхождения в текст T хотя бы одного образца из P. Здесь неприменимы недетерминированные конечные автоматы. Можно, конечно, предложить проверить каждый из образцов по очереди, но их может быть очень много.

Алгоритм классического решения задачи точного сопоставления множеств был предложен в 1975 г. Альфредом Ахо (Alfred V. Aho) и Маргарет Корасик (Margaret J. Corasick). Маргарет Корасик — женщина, ее фамилия не склоняется :, поэтому неверно название "алгоритм Ахо-Корасика". Это эффективный алгоритм, работающий за время $T = O(n + \sum_i m_i)$, где m_i — длина образца P_i.

Языком L называется некий набор слов над алфавитом Σ. Пусть $\Sigma = \{$*а,бЖю,я*$\}$. Возьмем такой *язык* над этим алфавитом:

$$\{\textit{абрау, абрек, брусника, брусья, руслан, русь}\}$$

Для языка L можно построить язык всех слов, содержащих хотя бы одно слово исходного языка в качестве подслова. Будем обозначать такой язык L'.

Все слова можно расположить друг под другом в лексикографическом порядке. При это станет очевидным, что некоторые из них содержат общий префикс (отмечен серым):

абрау
абрек
брусника
брусья
руслан
русь

Список слов можно представить в виде дерева, где общие префиксы не повторяются

абр –ау
–ек
брус–ника
–ья
рус –лан
–ь

Представим автомат для поиска слов нашего языка в виде дерева рис.3.12. Показаны не все стрелки. Любая стрелка из любого состояний с "не той" буквой будет вести в начальное состояние. Ясно, что такой автомат не все-

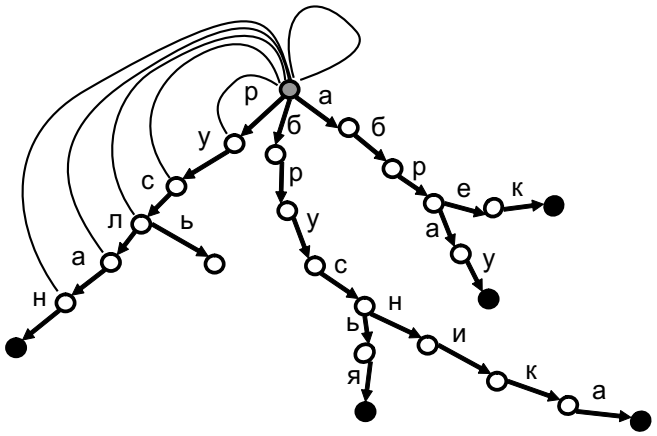


Рис. 3.12: Автомат для распознавания группы слов (языка). Показаны не все стрелки. Любая стрелка из любого состояний с "не той" буквой будет вести в начальное состояние

гда будет работать правильно. Например, в тексте "абрусья" не будет найдено слово брусья. Поэтому следует поступить по аналогии с конечным автоматом для распознавания одного слова, т.е. нам необходимо провести обратные стрелки в какие-то разумные вершины. Это такие вершины, для которых префикс (путь от корня до вершины) совпадает с суффиксом той части слова, которая была принята до момента неудачи.

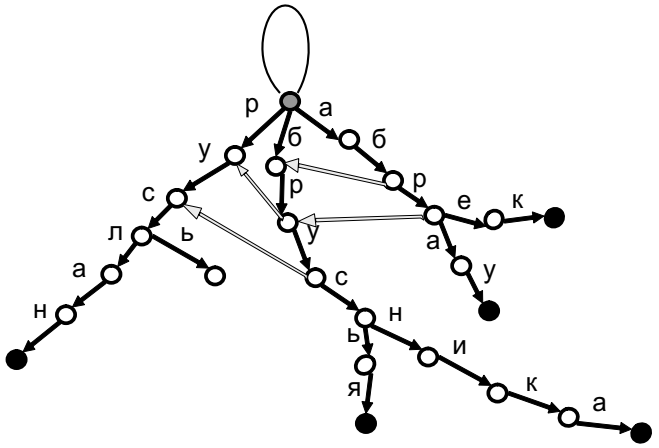


Рис. 3.13: Автомат Ахо-Корасик. Обратные ребра (неудачи) обозначены серым. Ребра, ведущие в корень не показаны.

Для того, чтобы нарисовать стрелки обратных переходов. Для этого введем функцию неудач. Функция неудач — куда надо идти, если идти некуда, т.е. для каждой вершины определено: $f(v) : V \to V$. Метку ребра, ведущего в вершину v будем обозначать $g(v)$. Для каждой вершины (состояния) можно определить ее уровень — расстояние до корня. Ясно, что функция неудач должна идти в вершину меньшего уровня (догадайтесь, почему?), но далеко не всегда должна идти в корень: $level\,(f(v)) < level(v)$ если $level(v) > 0$. Алгоритм построения обратных переходов (функции неудач) основан на просмотре вершин по уровням. Ясно, что если, находясь в корне, мы не примем буквы (это значит, что ни одно из слов языка на эту букву не начинается) то мы должны в корне остаться. Отсюда первое правило: $f(0) = 0$. Если, приняв первую букву какого-то слова из языка у нас нет продолжения, то ясно, что надо вернуться в корень (собственный суффикс слова длины 1 равен 0). Отсюда второе правило: $\forall v : level(v) = 1 f(v) = 0$. Если на более высоком уровне нас постигла неудача, то надо посмотреть, куда мы шли в случае неудачи из предыдущей вершины. Если мы шли в корень, то из текущей вершины мы можем принять букву в качестве первой буквы слова из языка (если такое слово есть). Поэтому следующее правило: $if\,(f(v_p rev) = 0)$ ищем такую вершину u, что $u : level(u) = 1 \& g(u) = g(v)$. Если находим, то определяем $f(v) = u$, иначе у нас нет слова, начинающегося с буквы $g(v)$, потому $f(v) = 0$. Если же предыдущая вершина указывала в случае неудачи на какую-то другую вершину $w = f(v_p rev)$, то надо проверить, а не совпадают ли метки ребра в текущую вершину $g(v)$ и метка какого-либо ребра, исходящего из w (не совпадает ли суффикс подслова с префиксом другого слова). Иными словами $if(f(v_p rev \neq 0)$ ищем вершину u такую, что $w = f(v_p rev); u : u_p rev = w \& g(w) = g(v)$. Если такая вер-

шина нашлась, то $f(v) = u$. Иначе пробуем найти среди начал слов (см. предыдущее правило). Итак алгоритм:

```
1.  f(0)=0;
2.  for(level from 1){
3.     for(v: v.level=level){
4.        f(v)=0;
5.        if(level >1){
6.          w= f(v.prev);
7.          if(w!=0){
8.             for(u: u.prev=f(w)){
9.                if(g(u)=g(v))
10.               f(v)=u;
11.            }
12.            if(f(v)=0){
13.               for(u: u.level=1){
14.                  if(g(v)=g(u))
15.                  f(v)=u;
16.               }
17.            }
18.         }
19.      }
20.   }
21. }
```

Пояснение. Строка 1 определяет функцию неудач для корня. Строка 2 определяет цикл по уровням. Строка 3: назначаем функцию неудач 0, впрочем потом можем ее изменить (строки 9, 13). Строки 4 и далее описывают работу для случая, когда уровень вершин больше 1. Строка 7 — проверка, указывает ли предыдущая вершина куда-либо. Строка 8 — проходим по всем потомкам этой вершины и проверяем не совпали ли метки (строка 9). Если метки совпали, то определяем f (строка 10). Если до сих пор функция неудач не определилась (строка 12), то пытаемся начать новое слово (строки 13, 14, 15).

Важное замечание. функция неудач показывает стрелки, на которых не написан символ (метка). Это значит, что при таком переходе символ на самом деле не снимается и после перехода в новое состояние мы должны еще раз сделать проверку этого символа. Исключение представляют только переходы неудач, исходящие из корня. При таких переходах символ снимается из входного потока и более не проверяется. Поэтому в автомате Ахо-Корасик есть три вида переходов (стрелок) - стрелки с метками, стрелки без метки и без снятия символа и переходы без метки, но со снятием символа. На самом деле последний тип перехода только один, и на нем можно

поставить метку - любой символ, кроме тех, которые стоят на 'правильных' стрелках.

Автомат, построенный по алгоритму Ахо-Корасик (рис.3.13), имеет более "приятный" вид, чем автомат (3.12). В нем нет "избыточных" состояний. Такой автомат называют еще деревом Ахо-Корасик. Проверим с помощью нашего автомата, принадлежит ли слово "абруслабрек" к языку L'. Проиллюстрируем шаги работы автомата с помощью таблицы. Состояниям будем индексировать префиксами слов.

Автомат в состоянии	На входе строка	Автомат переходит в состояние
""	а	"а"
"а"	б	"аб"
"аб"	р	"абр"
"абр"	у	"бру"
"бру"	с	"брус"
"брус"	л	"русл"
"русл"	а	"русла"
"русла"	б	"аб"
"аб"	р	"абр"
"абр"	е	"абре"
"абре"	к	"абрек"

Автомат Ахо-Корасик имеет широкий спектр применений. Например, поиск компьютерных вирусов часто использует этот алгоритм. Вся база данных вирусов собирается в автомат Ахо-Корасик, после чего этот автомат применяется для сканирования компьютера. В биоинформатике также часто применяется этот автомат для поиска множества возможных образцов. Идеи, лежащие в основе автомата Ахо-Корасик можно использовать также для статистического анализа частот встречаемости мотивов.

3.6 Поиск многих образцов в тексте. Препроцессинг текста

В предыдущих параграфах мы рассмотрели задачу поиска подстроки в длинном тексте. При этом подстрока (в общем случае образцы) заранее готовились — строились массивы суффикс-префиксов, создавались разного рода конечные автоматы. При этом подготовленный образец можно многократно применять для поиска в разных текстах. Однако иногда полезно провести подготовку текста, чтобы потом можно было эту подготовленную структуру использовать для поиска разных образцов. Например, хорошо бы подготовить последовательность генома, чтобы потом в нем эффективно искать разного рада образцы. Рассмотрим все ту же основную задачу поиска точного вхождения строки-образца P в длинном тексте T.

3.6.1 Суффиксные массивы

Образец присутствует в тексте, если и только если с него начинается один из суффиксов текста. Наивный алгоритм поиска на самом деле заключался в последовательном сравнении образца со всеми суффиксами — прикладывание образца к тексту в позиции i эквивалентно сравнению образца и начала суффикса. Рассмотрим, например текст `mississippi`. Чтобы в нем найти слово, скажем `sip`, мы прикладываем образец к началам суффиксов:

0	mississippi
1	ississippi
2	ssissippi
3	sissippi
4	issippi
5	ssippi
6	sippi
7	ippi
8	ppi
9	pi
10	i

Поиск образца в этой интерпретации напоминает поиск в массиве (не сортированном), что требует времени порядка $T = O(n \times t)$, где t — среднее время одного сравнения. Это время не превышает длину образца. Но в начале книги мы узнали, что искать в сортированном массиве гораздо эффективнее объект можно найти за логарифмическое время. Отсортируем массив суффиксов в лексикографическом порядке:

10	i
7	ippi
4	issippi
1	ississippi
0	mississippi
9	pi
8	ppi
6	sippi
3	sissippi
5	ssippi
2	ssissippi

В таком массиве можно найти образец за время порядка $T = O(\log n \times t) \leq O(\log n \times m)$, что намного быстрее. Здесь надо заметить, что совсем нет необходимости реально помнить суффиксы в отдельном массиве — ведь у нас есть исходный текст. Поэтому в массиве достаточно иметь позиции суффиксов. Хотя поиск происходит быстро надо потратить время на построение суффиксного массива. Время сортировки массива составляет величину порядка $n \times \log n \times tc$, где tc — среднее время сравнения суффиксов (чтобы

понять кто больше). Таким образом, чтобы провести поиск одного образца в тексте требуется время порядка $T = O(n \times \log_2 n \times t \times tc)$, что больше, чем время, необходимое для поиска образца алгоритмом Кнута-Морриса-Пратта. Однако при массовом поиске этот метод имеет значительное преимущество.

3.6.2 Суффиксное дерево

При поиске в суффиксном массиве мы тем не менее несколько раз прикладывали образец к суффиксам. Однако, при поиске слова, скажем, "`sip`" разумно было бы начинать сравнение прямо с той части массива, который начинается с буквы '`s`', а при поиске слова "`miss`" разумно было бы начинать поиск в том месте массива, где лежат слова, начинающиеся с буквы '`m`'. В алгоритме Ахо-Корасик все слова были организованы в виде дерева. Суффиксы также можно организовать в виде дерева. Однако, чтобы в дальнейшем не было путаницы введем еще символ конца строки, например символ $. В результате получим такое дерево (рис. 3.14).

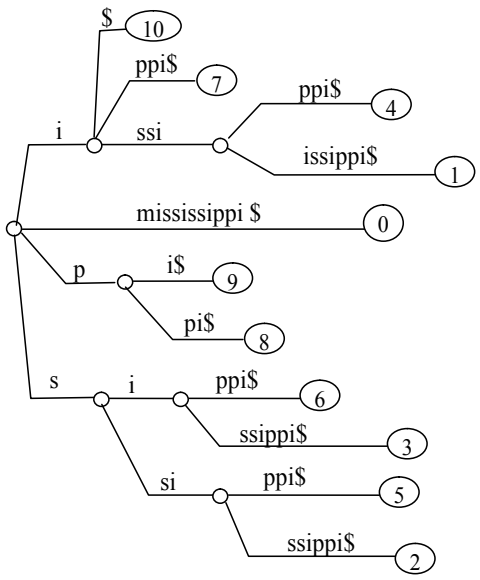


Рис. 3.14: Суффиксное дерево для слова `mississippi`

Основные свойства суффиксного дерева:

- Листьев столько же, сколько и позиций в слове
- На ребрах определены строки (метка ребра)

- Каждый путь от корня до листа — суффикс
- На листьях указаны позиции начал соответствующих суффиксов
- Можно построить такое дерево за время $T = O(n)$, где n — длина слова

Разберемся теперь, как хранить суффиксное дерево, используя линейную память. Для этого оставим в суффиксном дереве только вершины, имеющие более одного ребенка. Вместо строки $[t_i \cdots t_j]$ для ребра будем хранить ссылку на соответствующий сегмент слова. В описанном виде суффиксное дерево называется сжатым. Заметим, что, так как теперь каждая внутренняя вершина является вершиной разветвления, она добавляет к дереву как минимум один лист. Листьев в любом суффиксном дереве столько же, сколько букв в слове (пусть $n > 1$), поэтому число внутренних вершин в сжатом суффиксном дереве варьирует от 1 до n-1. Таким образом, всего вершин и ребер в сжатом суффиксном дереве будет линейное зависеть от длины и дерево будет занимать линейную память.

3.6.3 Алгоритмы построения суффиксного дерева

В предыдущем параграфе был предложен алгоритм построения суффиксного дерева. Однако этот алгоритм предполагает, что вес текст доступен одновременно. Однако для многих приложений нам не хочется анализировать сразу весь текст, а строить дерево по мере чтения текста. Такие алгоритмы называются *on-line* алгоритмами. On-line подход построения суффиксного дерева строит дерево последовательно для всех префиксов слова, получая в итоге полное дерево.

Прежде чем обратиться к описанию алгоритма, введем понятие *неявного* суффиксного дерева. Неявное суффиксное дерево — это суффиксное дерево для текста без знака \$ на конце. Некоторые суффиксы в нем заканчиваются на ребрах или во внутренних вершинах, и их номер нигде не хранится. В таком дереве листьев, как правило, меньше, чем позиций в слове. Обычное суффиксное дерево называется еще явным.

Есть слово $T = t_0 t_1 \cdots t_n$. Построим его суффиксное дерево с помощью *on-line* алгоритма. Будем строить деревья по очереди для всех префиксов слова. На первом шаге строим дерево для слова, состоящего из одной буквы t_0. На следующем шаге получаем букву t_1, теперь нам нужно строить дерево для слова $t_0 t_1$. Не будем строить заново, а достроим уже имеющееся дерево. Вот план работы алгоритма над словом $t_0 t_1 \cdots t_n$ (рис.3.15):

0	Строим суффиксное дерево для t_0
1	Расширяем его до дерева для t_1
	$\cdots$
n-1	Расширяем дерево для $t_0 t_1 \cdots t_{n-2}$ до дерева для $t_0 t_1 \cdots t_{n-1}$.
n	Расширяем дерево для $t_0 t_1 \cdots t_{n-1}$ до дерева для $t_0 t_1 \cdots t_n$ (делаем из неявного дерева явное)

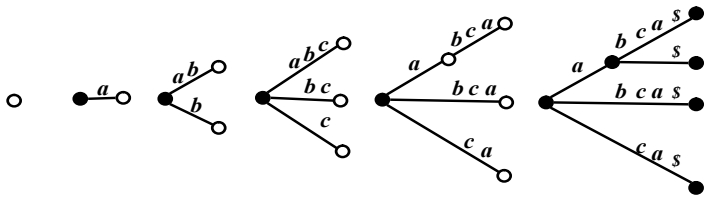


Рис. 3.15: Построение суффиксного дерева слова "abca" с помощью on-line алгоритма. Первые пять деревьев являются неявными, последнее — явное суффиксное дерево

Рассмотрим i-й этап работы алгоритма. На этом этапе мы перестраиваем неявное дерево для префикса строки $T_i = t_0 t_1 \cdots t_{i-1}$ в неявное дерево для префикса $T_i = t_0 t_1 \cdots t_{i-1}$. С этой целью для каждого j от 0 до $i-1$ находим в суффиксном дереве конец суффикса $t_j \cdots t_{i-1}$. Далее продляем его буквой t_i, если необходимо. При этом действуем по одному из следующих трех правил:

1. Правило продления. Если, прочитав суффикс $t_j \cdots t_{i-1}$, мы пришли в лист суффиксного дерева, удлиняем ребро, ведущее в этот лист, добавляя к строке, записанной на ребре, новую букву t_i.

2. Правило ответвления. Прочитав суффикс $t_j \cdots t_{i-1}$, мы можем остановиться не на листе, а в какой-нибудь внутренней вершине или даже прямо на ребре. Тогда определяем новую "точку роста" и определяем на ней начало соответствующего суффикса. Если остановились на ребре, а следующая буква — не t_i, придется разбить ребро на две части. Новая вершина добавляется после буквы t_{i-1}. От этой вершины будут отходить два ребра: остаток старого и новое ребро, несущее букву t_i.

3. Пустое правило. Если, прочитав суффикс $t_j \cdots t_{i-1}$, мы видим, что дальше уже есть нужная нам буква t_i, не создаем ничего нового, кроме точки роста.

Оценка времени работы алгоритма. Для построения суффиксного дерева слова длиной n on-line алгоритм должен пройти n шагов. На i-ом шаге мы продлеваем i суффиксов (к примеру, на втором шаге мы продлеваем два суффикса: $t_0 t_1$ и t_1). Продление каждого j-го суффикса на i-ом шаге занимает $O(i-j)$ операций. Действительно, счетчик j на каждом шаге меняется от нуля до i. Рассмотрим опять второй шаг работы алгоритма: нам нужно продлить слово длиной 2 ($i = 2$) буквой t_1. Счетчик j сначала равен нулю, и мы продлеваем суффикс $t_0 t_1$ (длиной $i - j$), это также займет $O(i - j)$ операций, поскольку читаем суффикс побуквенно. Затем $j = 1$, продлеваем слово t_1. Когда $i = j = 2$, продлеваем пустое слово. В итоге продление каждого суффикса на i-ом шаге стоит $O(i - j)$ операций, где j для каждого i

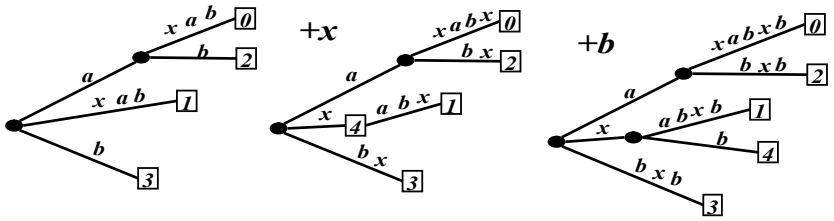


Рис. 3.16: К алгоритму построения суффиксного дерева. При приходе нового символа мы прежде всего продолжаем "точки роста". Потом проверяем, не привело ли появление нового символа к появлению новых точек роста. Точки роста отмечены квадратиками

меняется от 0 до i. С учетом сказанного составим выражение для скорости работы алгоритма над словом длины n:

$$T(n) = O\left(\sum_{i=0}^{n}\sum_{j=0}^{i}(i-j)\right) = O(n^3)$$

Существуют алгоритмы, формирующие суффиксное дерево слова длиной n за линейное время $O(n)$, но они значительно сложнее кубического алгоритма. Один из них строится модификацией описанного on-line подхода, другой (алгоритм Мак-Крейта) начинает работу с пустого дерева и добавляет суффиксы, начиная с самого длинного. Алгоритм Мак-Крейта не является on-line алгоритмом, т.е. для его работы необходима вся строка целиком. Мы не будем обсуждать здесь эти алгоритмы.

3.6.4 Сравнение суффиксных деревьев и суффиксных массивов

Суффиксные массивы считаются более простым способом хранения текста, чем суффиксные деревья. Действительно, алгоритм построения суффиксного массива гораздо проще, чем любой алгоритм построения суффиксного дерева.

Огромный плюс суффиксных массивов — их размер в памяти определяется только размерами текста T и никак не зависит от его алфавита, в отличие от суффиксных деревьев. Несмотря на то, что время построения массива хуже, чем дерева, суффиксное дерево требует большого количества памяти — $T = O(|T| \cdot |\Sigma|)$, где $|\Sigma$ — мощность алфавита. В некоторых задачах о поиске подстрок алфавит очень велик (например, естественные языки с большими алфавитами, сравнение изображений, где алфавит - все возможные значения цвета пикселя) и иногда память, которую занимает суффиксное дерево, делает его неприемлемым. Суффиксные массивы гораздо более рационально используют память: для их хранения достаточно

одной строки длины $|T|$ и $4 \cdot |T|$ байтов для хранения целых чисел суффиксного массива. При этом поиск по суффиксному массиву работает почти так же быстро как поиск по суффиксному дереву.

3.6.5 Примеры применения суффиксных деревьев

Поиск подстроки. Знакомая нам уже задача: поиск подстрок, равных образцу P длиной m, в тексте T длиной n. Некоторые уточнения: текст фиксирован, ищем много разных образцов, текст длинней образца, в образце не встречается знак конца строки '$'. В таком виде задача поиска выгоднее всего решается с помощью суффиксных деревьев, при условии достаточного объема памяти (иначе лучше задействовать суффиксные массивы, см. ниже).

Примеры задачи: поиск разных французских слов в тексте "Войны и мира" Л.Н. Толстого; биоинформатический пример — поиск регуляторных последовательностей в эукариотическом геноме. Для решения задачи построим суффиксное дерево для текста T. Будем читать паттерн вдоль дерева от корня. Если в какой-то момент не сможем прочитать следующую букву паттерна, значит, в тексте T ни разу не встречается строка, равная P. Допустим, что такая строка с тексте все-таки есть, тогда, прочитав эту строку, мы приходим либо во внутреннюю вершину v, либо останавливаемся на ребре (прийти в лист мы не можем, поскольку в паттерне не встречается знака '$'). Если остановились на ребре, проходим до ближайшей вершины v вниз по дереву. Далее читаем числа на листьях потомков вершины . Эти числа — номера суффиксов, начинающихся с подстроки P, а значит, индексы вхождений P в текст T.

Такой поиск занимает приблизительно $T = O(m)$ операций. Точно оценивая, заметим, что, прочитав все буквы паттерна по дереву за $O(m)$ операций, нужно еще дойти до ближайшей вершины, а потом узнать у ее потомков номера соответствующих суффиксов. Поэтому боле точная оценка времени работы поиска говорит о $O(m + l)$, где l — число листьев у вершины v. С помощью описанного поиска мы находим позиции всех вхождений паттерна P в текст T. Быстрее работает алгоритм, который ищет только первое вхождение слова в текст. Если каждая вершина суффиксного дерева "знает" число, хранящееся в ее ближайшем ребенке, дойдя до вершины v по прочтении слова P, мы узнаем индекс первого вхождения P в текст T. Этого часто достаточно для решения задачи (особенно если мы хотим просто знать, есть или нет в неком тексте некое слово). Алгоритм поиска первого вхождения работает в точности за $O(m)$.

Поиск повторов. Любая внутренняя вершина суффиксного дерева соответствует повторам. Позиции на всех листьях-потомках этой вершины есть точные повторы.

Поиск наибольшего общего подслова нескольких слов. Кроме задачи о поиске образца в фиксированной строке, суффиксные деревья помогают решать задачу о поиске максимального общего подслова. Пусть у нас есть два слова, к примеру, нуклеотидные последовательности T_1="`atgcat`" и T_2="`ttatgc`". Найдем их максимальную общую подпоследовательность. Наивный алгоритм заключается примерно в следующем: перебираем все подслова первого слова и каждый раз ищем их во втором слове, к примеру, с помощью алгоритма Кнута-Морриса-Пратта. Такой алгоритм работает за $O(n^4$ операций, где n — длина большего слова. С учетом большой длины реальных нуклеотидных последовательностей наивный алгоритм работает очень медленно. Решение той же задачи с помощью суффиксных деревьев. Для начала необходимо объединить слова, в которых ищем общее подслово, вместе, и построить для нового слова суффиксное дерево.

Объединяем: "`atgcat`" + "`ttatgc`" = "`atgcat*ttatgc`", получаем слово вида $T_1 * T_2$. Чтобы избежать химер можно вставить между словами T_1 и T_2 дополнительный незначащий символ, иной чем \$ (к примеру, *). Теперь нам надо в этом объединенном слове найти повтор, причем такой, что существует как в первом, так и во втором слове. Строим суффиксное дерево (рис.3.17), при этом отмечаем вершины, у которых есть потомки как из T_1 так и из T_2.

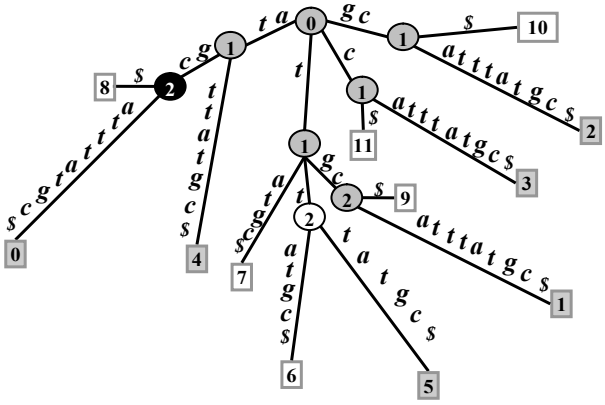


Рис. 3.17: Поиск общего слова. Длинные суффиксы отмечены серым

Введем обозначения: длинными суффиксами будем называть те, что начинаются в слове T_1 (суффиксы с нулевого по пятый), короткими — те, что начинаются в слове T_2 (суффиксы с шестого по одиннадцатый). Для каждой внутренней вершины дерева выясним, есть ли у нее потомки, относящиеся одновременно к короткому и длинному суффиксам. Если вершина удовлетворяет этому условию (назовем такие вершины ДК и выделим на рисунке серым), значит, строка, соответствующая этой вершине (строка, написанная на ребрах от корня дерева до этой вершины), встречается в слове как минимум в двух местах, начинаясь в T_1 и в T_2. Самая далекая от

корня ДК-вершина определит искомое максимальное подслово (выделено черным цветом на рисунке). На рисунке все внутренние вершины подписаны с учетом их уровня. Самые глубокие ДК-вершины имеют уровень 2, таких вершин три штуки. Так как нам нужно найти максимальную общую подстроку, выбираем вершину, в которую ведут ребра с максимальной по длине строкой. Такая строка и является решением задачи, в нашем случае это строка "atgc". Алгоритм работает за $O(|T_1| + T_2|)$ — за линейное время. Согласитесь, это лучше, чем тетричное время. Понятно, что общее подслово можно искать не только в двух, но и в большем количестве слов. Для трех слов придется вводить, кроме длинных и коротких, еще и "средние" суффиксы, начинающиеся во втором слове.

3.7 Сложность текста

Вы всегда легко запоминаете телефонные номера своих друзей? Поспорим, что есть такие номера, что, увидев, запомните сразу, а есть такие, что всегда приходится записывать. Сложность здесь заключается не обязательно в вашей памяти. Телефонные номера — это тексты. Все тексты разные: есть "Война и мир", а есть геном E.coli; телефонные номера — тоже разные. Как оценить сложность текста? Вот, к примеру, два примера нуклеотидных текстов:

```
aaaaaaaaaaaattttttt
   atatatatatatat
```

Оба текста простые, потому что их просто описать. Вместо первого текста можно сказать: "12 раз `a`, 8 раз `t`". Второй текст еще проще — "7 раз `at`". А попробуйте описать такой текст:

```
cactgaaactgttgtactta
```

По длине он такой ж как первый, но гораздо сложнее. Такой текст проще прочитать (назвать все буквы), чем как-то описывать.

В 1960-х годах русский математик А.Н.Колмогоров поставил вопрос: "Какова внутренняя сложность описания строки двоичных символов?". Из ответа на этот вопрос сегодня мы с вами знаем, что сложность текста это — длина самой короткой программы, которая генерирует данный текст. Такая длина называется сложностью текста по Колмогорову и обозначается $K(s)$, где s — текст (строка).

Введем некоторые формальные понятия. Во-первых, будем для простоты рассматривать двоичный алфавит 0,1. Множество всех слов над этим алфавитом обозначим $\Sigma = \{0,1\}^*$. Рассмотрим множество $E = \{x, y\}$ пар слов такое,что:

1. если $(x, y_1) \in E$ и $(x, y_2) \in E$, то $y_1 = y_2$
2. Существует программа П такая, что $\forall (x, y) \in E : \text{П}(x) = y$

Такое множество и программа называются системой описания текста. На самом деле достаточно одной программы и ее области определения.

Во-первых, это похоже на архивирование-разархивирование. Здесь x — архив, y — разархивированный файл, а программа Π — разархиватор. Очевидно, что далеко не любой файл является архивом чего-либо (здесь и появляется ограничение на область определения программы). Во-вторых, здесь нет взаимной однозначности — разным x может соответствовать один y. А раз так, тот можно поставить вопрос о наиболее коротком описании строки.

Определение 4. *Сложностью $K_\Pi(y)$ текста y относительно программы Π называется минимум:*

$$K_\Pi(y) = \min_x (\Pi(x) = y)$$

Системы описания текста можно сравнивать. Говорят, что система Π_1 не хуже системы Π_2, если существует константа C, не зависящая от слова x такая, что

$$\forall x : K_{\Pi_1}(x) \leq K_{\Pi_1}(x) + C$$

Если есть две системы описания Π_1 и Π_2, то может быть на одних словах первая система более эффективна, а на других — вторая порождает более короткие описания. Есть теорема,что если есть две системы описания, то можно построить третью, которая будет не хуже двух исходных. Здесь нельзя забывать о константе! За каждый переход на новую систему описания надо платить добавлением константы в размер описателя. Тем не менее, можно поставить вопрос об описателе (программе), который был бы не хуже всех остальных и о размере описания текста с точки зрения этого описателя.

Любая программа имеет конечный размер. Поэтому количество программ счетно, а значит их можно пронумеровать. Если задан текст, о можно попробовать описать оптимальным способом всеми программами, и в начало описания поставить номер соответствующей программы. Таким образом можно построить универсальный описатель.

Определение 5. *Сложностью по Колмогорову называется длина самого короткого описания текста относительно некоторого универсального описателя.*

Свойства сложности по Колмогорову.

Свойство 1. *Сложность по Колмогорову не вычислима, т. е. не существует алгоритма, который глядя на последовательность за конечное время вычислит его сложность.*

Свойство 2. *Количество слов W сложности K может быть оценена как:*

$$2^{K-C} \leq W \leq 2^{K+1}$$

Случайный текст по Колмогорову — текст, любое описание которого не короче самого текста.

Конечно, эта характеристика относится к любым текстам, а не только к двоичным. Удобно относить ее к нуклеотидным текстам. Заметим, что большинство последовательностей, относящихся к структурным генам, имеют высокую сложность, тогда как некодирующие последовательности (сателлитные, например, центромерные и теломерные участки) могут на многие килобазы состоять из повторов короткого олигонуклеотида.

Различные архиваторы данных работают как раз описывая текст внутри архивируемого файла подобной программой. Fasta-файлы быстро пакуются любым архиватором, поскольку в длинных текстах над маленьким алфавитом всегда много повторов. Кстати, знаете, чем отличается написанный человеком "случайный" нуклеотидный текст от созданного природой? В рукотворном тексте почти никогда нет длинных повторов вида `ttttttttttt`, поскольку человеку подсознательно кажется, что наличие таких повторов говорит о неслучайности текста.

Пример программы-описателя. В нее могут входить два вида элементарных команд, а именно:

Добавить символ в конец текста

Скопировать блок текста из уже сгенерированной последовательности

Сгенерируем с помощь этих команд первый текст (пусть нумерация в строке начинается с единицы):

```
add a
copy 1 символа с позиции 1
copy 2 символа, начиная с позиции 1
copy 4 символа, начиная с позиции 1
copy 4 символа, начиная с позиции 1
add t
copy 1 символ с позиции 13
copy 2 символа начиная с позиции 13
copy 3 символа начиная с позиции 13
```

На выходе `aaaaaaaaaaaattttttttt`.

Фрагменты низкой сложности часто встречаются в нуклеотидных и аминокислотных последовательностях. Поэтому при поиске сходства, скажем программой BLAST, находки достаточно протяженных фрагментов низкой сложности зачастую занимает верхние строки выдачи, затеняя собой биологически-значимые результаты. Для того, чтобы избежать нерелевантных результатов, программа BLAST имеет встроенный модуль подавления фрагментов низкой сложности (они заменяются на букву `x`). Поэтому Вы никогда не найдете гомологов для последовательности

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Впрочем, эту опцию можно отключить.

3.8 Сжатие информации по Лемпелю-Зиву

Алгоритм Лемпеля-Зива — это универсальный алгоритм сжатия данных без потерь. Алгоритм основан на поиске и кодировании повторов в тексте. Общая идея алгоритма: если в уже прочитанном слове уже встречалась подобная последовательность символов, причем запись о ее длине и смещении от текущей позиции короче чем сама эта последовательность, то в выходной файл записывается ссылка (смещение, длина), а не сама последовательность. К примеру, строку "КОЛОКОЛ_ОКОЛО_КОЛОКОЛЬНИ" короче записывается в памяти компьютера так: "КОЛО"(-4,3)"_"(-5,4)"О_"(-14,7)"ЬНИ".

Для сжатия данных текста T длиной n строится суффиксное дерево (время построения $O(n)$). Что такое повтор в суффиксном дереве? Если ребро l ведет в вершину, разветвляющуюся на два ребра, строка, записанная на ребре l, встречается в слове два раза, если в три ребра — три раза и т.п. В

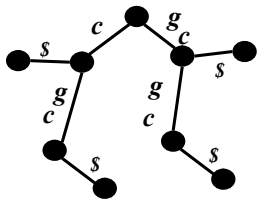


Рис. 3.18: Суффиксное дерево

изображенном на рис.3.18 суффиксном дереве для слова "gcgc" ребро "gc" ведет в вершину разветвления на два потомка. Подслово gc встречается в слове "gcgc" два раза. То же можно сказать и о подслове "c".

Алгоритм Лемпеля-Зива использует on-line подход, он читает и кодирует текст частями. Для начала строится суффиксное дерево части слова. Затем следующая порция текста как бы "протаскивается" через существующее суффиксное дерево — процесс, сходный с продлением ветвей при построении суффиксного дерева. Пусть уже упаковано $i-1$ символов. Нам надо вычислить следующую команду (позиция, сдвиг) или сгенерировать следующую букву. Берем суффикс $S(i,n)$ и пропускаем его через существующее дерево (как при поиске слов). Либо находим на ребрах дерева максимальное слово, которое является префиксом $S(i,n)$, и тогда ставим команду скопировать это слово. В противном случае генерируем новую букву. Суффиксное дерево можно генерировать одновременно с поиском.

Распространенный метод сжатия RLE (Run Length Encoding), который заключается в записи вместо последовательности одинаковых символов одного символа и их количества, является подклассом данного алгоритма. Слово "AAAAAAA" будет закодировано RLE в виде (A,7), а алгоритмом Лемпеля-Зива в виде "A"(-1,6). Алгоритм RLE менее универсален по сравнению с алгоритмом Лемпеля-Зива, он может сжать не любой текст.

Глава 4

Алгоритмы для графов

Определение 1. *Графом называется пара множеств* $G = \{V, E\}$ *и отображение* $g : V \otimes V \to \{E, null\}$. V *называется множеством* вершин *графа, а* E *— множеством* ребер *графа. Если отображение* g *коммутативно* $(g(v_1, v_2) = g(v_2, v_1))$*, то граф называется неориентированным, иначе граф ориентированный. Заметим, что обычно функция* g *обладает свойством* $g(v, v) = null$*, т.е. нет ребер из вершину в себя же. Иначе мы имеем дело с псевдографом. Граф* $G' = \{V', E'\}$ *называется подграфом графа* G*, если* $V' \subset V$ *и* $E' \subset E$.

Здесь мы будем рассматривать только *конечные графы*, т.е. такие графы, в которых множество вершин (и ребер) *конечно*. Ниже приведены еще некоторые определения, которые могут понадобиться.

путь (цепь) последовательность вершин, которой каждая вершина, кроме последней, соединена с предыдущей вершиной ребром.

цикл путь, в котором первая и последняя вершины совпадаю.

простой путь путь, ребра в котором не повторяются.

элементарный путь простой путь, вершины в котором не повторяются.

ациклический граф граф, не содержащий циклов.

взвешенный граф граф, на ребрах которого определены веса.

связная компонента подграф G' данного графа, такой, что для любых вершин $v1, v2 \in G'$ есть путь из $v1 \to v2$, а между любой вершиной $v3 \in G'$ и любой вершиной $v4 \notin G'$ пути нет.

связный граф граф, в котором для любых вершин $v1, v2$ есть путь из $v1$ в $v2$. Связный граф имеет одну связную компоненту.

сильно связный граф ориентированный граф, у которого из любой вершины в любую имеется путь.

дерево связный ациклический граф.

лес граф, у которого все связные компоненты — деревья.

полный граф граф, любые две вершины которого соединены ребром (такой граф — сам себе максимальная клика).

клика подмножество вершин графа, такое, что между любой парой вершин есть ребро (полный подграф).

смешанный граф граф, содержащий ориентированные и неориентированные ребра.

двудольный граф (k-дольный граф) граф, вершины которого можно разбить на 2 (k) непересекающихся подмножества, так, что не будет ребер, соединяющих элементы одного и того же подмножества.

индекс вершины неориентированного графа число ребер, выходящих из вершины.

индекс вершины ориентированного графа число, зависящее от количества ребер, входящих и выходящих из вершины. Одно входящее в вершину ребро прибавляет к этому числу единицу, одно выходящее — отнимает единицу.

ε-граф граф, в котором число ребер равно $|E| = (1+\varepsilon)|V|, \varepsilon \ll 1$

4.1 Способы представления графов

Большинство привыкло рассматривать и изображать графы в таком виде, как изображено на рис.4.1. Безусловно, это очень наглядный способ, тем не менее, непонятный компьютеру. Удобно хранить графы в виде табли-

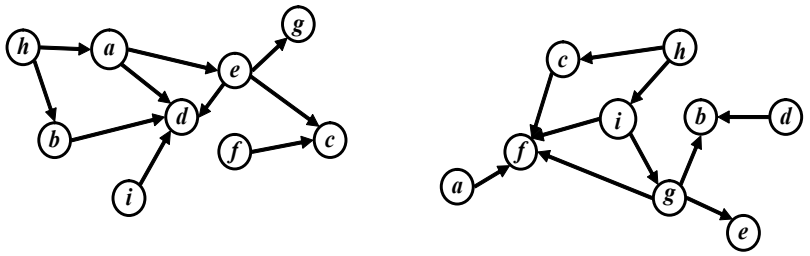


Рис. 4.1: Пример графического представления графов. Одинаковые (изоморфные) ли графы представлены на рисунке?

цы, где в столбцах и строках заданы вершины графа, если на пресечении

столбца и строки есть отметка (+ или, удобней, true), между соответствующими ребрами есть ребро. Изображенный нами справа граф в виде таблицы представляется так:

	a	b	c	d	e	f	g	h	i
a	0	0	0	1	1	0	0	0	0
b	0	0	0	1	0	0	0	0	0
c	0	0	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0	0	0
e	0	0	1	1	0	0	1	0	0
f	0	0	1	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0	0
h	1	1	0	0	0	0	0	0	0
i	0	0	0	1	0	0	0	0	0

Это ориентированный граф. Если бы граф был неориентированным (рис.4.2), таблица приняла бы несколько иной вид:

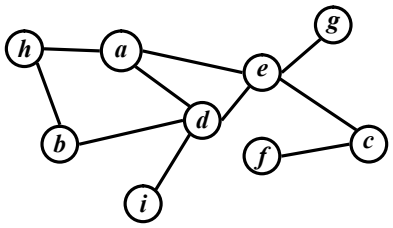


Рис. 4.2: неориентированный граф

	a	b	c	d	e	f	g	h	i
a	0	0	0	1	1	0	0	1	0
b	0	0	0	1	0	0	0	1	0
c	0	0	0	0	1	1	0	0	0
d	1	1	0	0	1	0	0	0	1
e	1	0	1	1	0	0	1	0	0
f	0	0	1	0	0	0	0	0	0
g	0	0	0	0	1	0	0	0	0
h	1	1	0	0	0	0	0	0	0
i	0	0	0	1	0	0	0	0	0

Если на пересечении строки и столбца подобной таблицы указано true (1), значит, из вершины в строке идет ребро в вершину в столбце. Подобная таблица называется матрицей смежности графа. Матрица смежности полного графа будет включать только единицы. Как хранить такую матрицу в памяти? Либо в виде двумерного массива, либо закодировав с применением алгоритма Лемпеля-Зива. Ни первое, ни второе в данном случае удобным не кажется.

Для записи и хранения графа можно составить список ребер этого графа. Смежными назовем вершины, соединенные ребром (в неориентированном графе). Для приведенного выше примера неориентированного графа список смежных вершин будет выглядеть так: ad; ae; bd; ed; eg; ec; fc; ha; hb; id;. Если граф ориентированный, порядок в парах важен.

Еще граф можно хранить в виде нескольких связных списков смежных вершин. Из каждой вершины графа пишем связный список, состоящий из вершин, смежных данной вершине (вершин, с которыми данная вершина связна ребром, для неориентированного графа, и вершин, в которые идет "стрелка"из данной вершины для ориентированного графа, *список соседей*). Тогда представление ориентированного графа будет следующим:

вершина	Список смежных вершин	вершина	Список смежных вершин
a	$d \to e$	f	c
b	d	g	
c		h	a→b
d		i	d
e	$d \to g \to c$		

Для неориентированного графа:

вершина	Список смежных вершин	вершина	Список смежных вершин
a	$d \to e \to h$	f	c
b	$d \to h$	g	e
c	$e \to f$	h	$a \to b$
d	$a \to b \to e \to i$	i	d
e	$a \to c \to d \to g$		

Графы являются удобным языком дискретной математики, и многие задачи формулируются в форме задач на графах. В частности, многие задачи биоинформатики имеют графовую постановку. Примеров можно привести множество. Задача выравнивания последовательностей формулируется как задача поиска кратчайшего пути в графе. Задача поиска ортологов может быть сформулирована как задача поиска клики в графе. Задача построения белковых семейств формулируется как задача поиска связной компоненты в графе или как задача поиска двусвязной компоненты, или как задача поиска клик в графах. Задачи кластеризации, восстановления последовательности по результатам секвенирования, задачи, связанные со скрытыми Марковскими моделями и т.д. и т.п. Часто графы имеют гигантские размеры. Например, при выравнивании двух последовательностей длиной по тысяче символов возникает граф размером порядка миллиона вершин. Поэтому построение эффективных алгоритмов на графах является актуальной задачей.

4.2 Обход графа в ширину

Многие задачи на графах требуют просмотра всех вершин в некотором порядке. Есть два основных способа обхода графа — обход в ширину и в глубину. Где у графа ширина, а где глубина, сказать сложно, но под обходим графа в ширину подразумевают просмотр всех соседей выбранной вершины, потом соседей их соседей и т.п. Для обхода графа в ширину используется такая структура данных, как очередь (first input — first output). Для работы алгоритма добавим к вершинам атрибут — цвет вершины:

белый — вершина, которую еще не видели
серый — вершина, которая находится в стадии обработки (находится в очереди)
черный— вершина, для которой обработка завершена

Вначале все вершины белые, поскольку на них еще ни разу не смотрели. Затем работаем по алгоритму:

1. Берем произвольную вершину, помещаем в очередь и красим в серый цвет.

2. Изымаем очередную вершину из очереди (красим в черный цвет) и все смежные не пройденные (белые) вершины помещаем в конец очереди и красим в серый цвет

3. Если очередь не пуста, переходим к 2.

4. Если очередь пуста, но остались не просмотренные вершины, берем из них произвольную, помещаем в очередь и переходим к 2

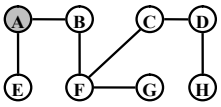
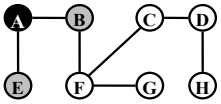
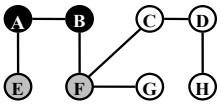
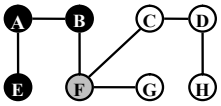
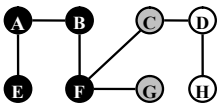
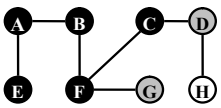
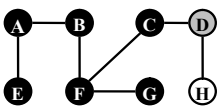
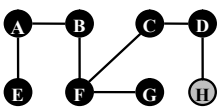
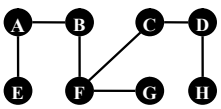
5. Если очередь пуста и все вершины графа черные, заканчиваем работу — граф обойден

В таблице 4.2 показан порядок работы алгоритма обхода графа в ширину.

Псевдокод алгоритма поиска в ширину:

```
1.  WideSearch(){
2.     for(each v in V){
3.     if(v.color==white) VertexWideSearch(v);
4.     }
5.  }

6.  VertexWideSearch(Vertex v){
7.     Queue q;
8.     q.put(v);
9.     v.color=gray;
10.    while(!q.empty){
11.       v=q.get();
12.       v.color=black;
13.       for(w in Neighbours(v)){
```

Таблица 4.1: Пример обхода графа в ширину.

Граф	Очередь	Действие
	A	Берем первую попавшуюся вершину (A), кладем в очередь и красим серым.
	BE	Снимаем вершину A из очереди, помечаем черным, кладем в очередь соседей — E и B
	EF	Снимаем первую вершину из очереди (B), красим в черный цвет, а ее соседей (F) кладем в очередь.
	F	Снимаем первую вершину из очереди (E), красим в черный цвет, а ее соседей (а их нет!) кладем в очередь.
	CG	Снимаем первую вершину из очереди (F), красим в черный цвет, а ее соседей (CG) кладем в очередь.
	GD	Снимаем первую вершину из очереди (C), красим в черный цвет, а ее соседей (D) кладем в очередь.
	D	Снимаем первую вершину из очереди (G), красим в черный цвет, а ее соседей (а их нет!) кладем в очередь.
	H	Снимаем первую вершину из очереди (D), красим в черный цвет, а ее соседей (H) кладем в очередь.
		Очередь пуста и нет не просмотренных вершин! Мы обошли весь граф

```
14.          if(w.color==white){
15.             q.put(w);
16.             w.color=gray;
17.          }
18.       }
19.    }
20. }
```

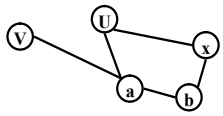
Пояснение. Здесь две подпрограммы. Первая ищет очередную белую вершину, чтобы с нее стартовать. Вторая подпрограмма осуществляет собственно поиск в ширину. Строка 2 означает, что мы просматриваем все вершины. Поскольку множество вершин не обязательно организовано в массив, то цикл не объявлен явным перечислением значений индекса. Строка 7 объявляет очередь, в которой будут стоять вершины. Стартовую вершину помещаем в очередь. Пока очередь не пуста (цикл в строках 10-20) выполняем следующее. Снимаем вершину из очереди, красим в черный, просматриваем всех соседей (цикл в строках 13-17) и помещаем белых соседей в очередь.

Время работы алгоритма $T = O(|V| + |E|)$. Действительно, мы смотрим по одному разу на каждую вершину в цикле строки 2 и смотрим на каждое ребро в цикле строки 13. Видно, что наиболее подходящим способом хранения графа для этой задачи является хранение списков соседних вершин. Впрочем, часто бывает, что ребра графа не заданы явно, а определяются "на лету как функция вершин. В этом случае время работы алгоритма может быть больше — $T = O(|V| \cdot |V|)$, поскольку цикл строки 13 заменится на просмотр всех вершин графа и появится проверка дополнительного условия являются ли вершины v и w соседями. Описанный алгоритм может помочь в решении задач анализа графов с целью выявления их структуры и вычисления ряда характеристик. С помощью поиска в ширину можно находить связные компоненты графа. Когда в какой-то момент обхода графа очередь становится пустой, мы просмотрели одну связную компоненту. Если в графе остались не просмотренные вершины, значит, имеется, как минимум, еще одна связная компонента. Просматриваем ее, ищем следующую Ж Таким образом можно находить и перечислить все связные компоненты графа. Поиск в ширину применим для нахождения расстояния между вершинами, а также поиска кратчайшего пути между вершинами.

Дано: произвольный граф $G = \{V, E\}, V > 1$. **Требуется** определить расстояние между вершинами $u \in V, v \in V$, а также указать кратчайший из путей, связывающих u и v (если такие пути существуют).

Решение: стартуя из вершины v, делаем обход графа в ширину, пока не дойдем до u. Если между событиями "v в очереди"и "u в очереди"очередь оставалась пустой, пути между v и u нет. Если между указанными событиями очередь ни разу не была пуста, мы нашли путь между v и u, притом этот путь кратчайший. Допустим, ищем кратчайший путь между вершинами v

и u в таком графе:



Обходим граф в ширину, начиная с v:

1. В очереди вершина v
2. В очереди a
3. В очереди вершины b и u , останавливаем поиск, так как дошли до вершины u.

Между шагом 1 и шагом 3 очередь не была пуста, значит, u и v принадлежат к одной связной компоненте, и между ними есть путь (или несколько путей). Мы нашли один из таких путей с помощью обхода по ширине. Этот путь $v - a - u$ (смотрим, какие вершины побывали в очереди между v и u). Итак, расстояние между v и u — два ребра или одна вершина. Мы решили правую часть задачи — нашли расстояние между вершинами v и u. ***Докажем***, что найденный нами с помощью поиска в ширину путь является кратчайшим путем между v и u.

Доказательство. от противного — пусть мы нашли не кратчайший путь и есть путь еще короче. Тогда вершина на кратчайшем пути встретилась бы раньше при просмотре, поскольку каждый раз мы смотрим на *ближайших* соседей (на смежные вершины) той вершины, то изымается из очереди. Возникло противоречие — утверждение доказано. □

Упражнение 21. *Напишите псевдокод для поиска связной компоненты.*

Упражнение 22. *Напишите псевдокод для поиска минимального расстояния между двумя вершинами.*

4.3 Обход графа в глубину

Идея этого метода — идти по графу от вершины к вершине вглубь, пока не "упремся". Затем возвращаемся на этап раньше, снова идем в глубину и т.д. В отличие от обхода в ширину здесь используется не очередь, а стек (first input — last output). При обходе графа будем красить вершины также: белым — вершины, которые еще не обработаны, серым, вершины, которые находятся в процессе обработки, черным — уже обработанные вершины. Итак, алгоритм:

1. Берем произвольную вершину и помещаем в стек (красим в серый цвет)

2. Смотрим на вершину, лежащую не вершине стека. Берем произвольного белого соседа этой вершины и, покрасив в серый цвет, кладем в стек. Если нет белых соседей, то переходим к п. 2., иначе повторяем п.2.

3. Если не осталось белых смежных вершин, то берем вершину из стека и заканчиваем ее обработку (красим в черный) и переходим к 2.

4. Если остались не просмотренные вершины, то берем из них произвольную, помещаем в стек и переходим к 2

В таблице 4.3 показана работа алгоритма поиска в глубину. Псевдокод алгоритма обхода в глубину:

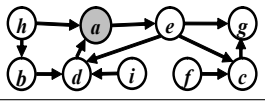
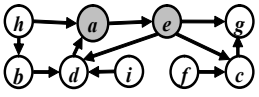
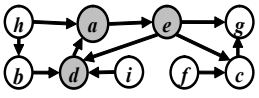
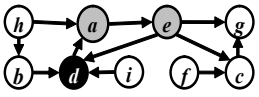
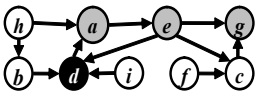
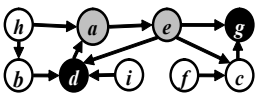
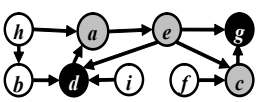
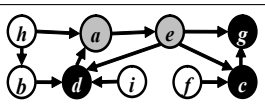
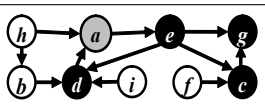
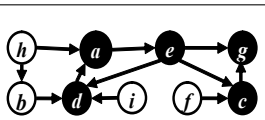
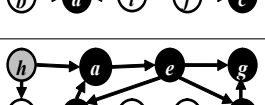
```
1.  DepthSearch(){
2.     for( each v   V){
3.     if(v.color=white)
4.     VertexDepthSearch(v);
5.     }
6.  }

7.  VertexDepthSearch(v){
8.     v.color=gray;
9.     for( each w   Neighbours(v)){
10.       if(w.color=white)
11.       VertexDepthSearch(w);
12.    }
13.    v.color=black;
14. }
```

Пояснения к псевдокоду. Подпрограмма DepthSearch просматривает все вершины, и, если находит белую вершину, то начинает просмотр в глубину, начиная с этой вершины (VertexDepthSearch). Подпрограмма поиска в глубину с заданной стартовой вершиной работает рекурсивно. Она красит текущую вершину в серый цвет (строка 8), затем находит среди соседей белую вершину и использует ее в качестве стартовой для дальнейшего поиска в глубину (строка 11). Когда все соседи обработаны (а каждый из них прошел в глубину до конца), вершину можно покрасить в черный цвет (строка 14). Также как и для поиска в ширину время поиска в глубину составляет $O(|V| + |E|)$. Работа со стеком, как и с очередью, занимает $O(|V|)$. Время на рассмотрение ребер $O(|E|)$.

Упражнение 23. *Перепишите псевдокод поиска в глубину в виде цикла вместо рекурсии. Здесь Вам кроме стандартных методов стека (`push` — положить элемент в стек и pop — снять элемент с вершины стека) понадобится метод `peek` — прочитать элемент с вершины стека не изымая его со стека.*

Разумеется, поиск в ширину и поиск в глубину не являются самостоятельными задачами. Обычно изменение цвета вершины сопровождается дополнительными действиями, например записью вершины в какой-нибудь список, вычислением каких-либо характеристик и т.п. Важно, что при обоих типах поиска мы проходим по соседним вершинам, т. анализируем связность графа. Еще раз стоит напомнить, что при решении реальных задач

Таблица 4.2: Пример обхода графа в глубину.

Граф	Очередь	Действие
	a	Берем произвольную вершину (a) и кладем в стек
	ae	На вершине стека лежит вершина a, поэтому берем ее произвольного белого соседа (вершину e) и кладем в стек.
	aed	На вершине стека лежит вершина e. поэтому берем ее произвольного белого соседа (вершину d) и кладем в стек.
	ae	У вершины d нет белых соседей, поэтому заканчиваем ее обработку (красим в черный)
	aeg	На вершине стека лежит вершина e, поэтому берем ее произвольного белого соседа (вершину g) и кладем в стек
	ae	На вершине стека лежит вершина g, и у нее нет белых соседей. Снимаем ее со стека и красим в черные цвет.
	aec	На вершине стека лежит вершина e, поэтому берем ее произвольного белого соседа (вершину c) и кладем в стек
	ae	На вершине стека лежит вершина c, и у нее нет белых соседей. Снимаем ее со стека и красим в черные цвет.
	a	На вершине стека лежит вершина e, и у нее нет белых соседей. Снимаем ее со стека и красим в черные цвет.
		На вершине стека лежит вершина a, и у нее нет белых соседей. Снимаем ее со стека и красим в черные цвет. Стек пуст.
	h	Находим произвольную белую вершину, кладем в стек.

приходится иметь дело с графами достаточно большого размера — тысячи, миллионы и более вершин. Так что нарисовать картинку и на нее посмотреть не удастся.

4.3.1 Типы ребер графа

В результате поиска в глубину получается дерево (или лес из нескольких деревьев) — дерево обхода. Каждая вершина графа v имеет время окончания обработки $t(v)$, т.е. тот шаг алгоритма, при котором вершина стала черной. После завершения обхода графа в глубину все ребра графа оказываются разбитыми на два множества — ребра дерева (леса) обхода и остальные ребра. В ориентированном графе ребра подразделяются на четыре разных типа (рис.4.3):

Ребра деревьев Ребра деревьев поиска в глубину (на рисунке толстые).

Прямые ребра. Вершины u, v принадлежат дереву, но (uv) не является ребром дерева, и $t(u) > t(v)$. Другими словами, это ребра (uv) не являющиеся ребрами дерева и соединяющие вершину u с ее потомком v в дереве поиска в глубину (на рисунке — двойные).

Обратные ребра. Вершины u, v принадлежат дереву и $t(u) < t(v)$. Другими словами, это ребра (uv), соединяющие вершину u с ее предком v в дереве поиска в глубину (на рисунке — пунктирные).

Перекрестные ребра. Все остальные ребра. Они могут соединять вершины одного и то-го же дерева поиска в глубину, когда ни одна из вершин не является предком другой, или соединять вершины в разных деревьях (на рисунке — серые).

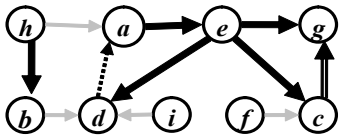


Рис. 4.3: Классификация ребер графа после обхода в глубину. Толстые ребра — ребра дерева обхода, двойные — прямые ребра, пунктирные — обратные ребра, серые — перекрестные ребра.

4.3.2 Построение покрывающего дерева

Покрывающим дерево графа называется совокупность деревьев обхода в глубину, соединенных особыми перекрестными ребрами. Чтобы построить такую структуру, нужно сначала сделать обход графа в глубину, создав лес

обхода. Если лес обхода состоит из одного дерева, оно и является покрывающим деревом. Если в лесе обхода несколько деревьев, посмотрим, нет ли перекрестного ребра из вершины одного дерева обхода в корень другого дерева обхода. Такое ребро, направленное из вершины A одного дерева обхода в корень другого, включаем в покрывающее дерево. Новым корнем становится A. Когда все такие ребра исчерпаны, покрывающее дерево готово. Оно состоит из всех деревьев обхода графа, соединенных особыми перекрестными ребрами

4.3.3 Поиск связных компонент

Мы уже обсуждали выше, как искать связные компоненты с помощью обхода графа в ширину. Этот метод подходит для любых графов, тем не менее, на ориентированных графах можно применять другой способ поиска связных компонент. Для нахождения связных компонент в орографе следует ко всем ребрам графа добавить перекрестные ребра (в отличие от покрывающего дерева это не обязательно ребра, идущие в корень дерева из леса обхода в глубину).

4.3.4 Поиск циклов

Поиск в глубину позволяет найти циклы в графе, причем несколькими способами. 1) Если при обходе в глубину мы положили в стек вершину v и увидели среди ее соседей серую вершину, в графе есть цикл, верно и обратное: если граф циклический, обязательно на каком-либо этапе обхода возникнет встреча с серым соседом. В приведенном примере такая ситуация возникала при обработке вершины d. Впрочем с таким же успехом можно применять 2) Циклы также можно найти, построив покрывающее дерево, как в ориентированном, так и в неориентированном графе. В ориентированном графе для нахождения циклов строим покрывающее дерево. Обратные ребра замыкают циклы. В неориентированном графе также строим покрывающее дерево и ищем все ребра, не принадлежащие к нему. Эти ребра порождают циклы. Отметим, что задача поиска всех циклов в графе весьма трудоемкая, поскольку количество всех возможных циклов очень велико. Действительно, если в графе есть два цикла с общей вершиной, то можно построить еще один цикл, объединяющий их. Если таких циклов несколько, то возникает большое комбинаторное разнообразие. Тем не менее можно эффективно решать задачу поиска хотя бы некоторых циклов.

Упражнение 24. *Напишите псевдокод для алгоритма поиска циклов в графе. Здесь по-видимому Вам понадобится дополнительная функция в структуре стека, позволяющая просмотреть стек не изымая вершины со стека.*

Упражнение 25. *Двусвязной компонентой неориентированного графа называется такое множество вершин, что через любые две вершины графа*

проходит цикл такой, в котором вершины проходятся по одному разу. Придумайте алгоритм поиска двусвязных компонент.

4.3.5 Некоторые приложения к биоинформатике

Есть множество аминокислотных последовательностей (скажем, UNIPROT). Задача состоит в том, чтобы найти белковые семейства. Здесь мы рассмотрим простейший наивный подход к этой задаче. На самом деле процедура кластеризации последовательностей в семейства содержит множество дополнительных деталей.

Итак, первым шагом кластеризации является определение расстояний между последовательностями (например, с помощью BLAST или Смита-Ватермана). Теперь можно построить граф: вершины — последовательности, а ребра проводятся, если вес выравнивания превышает заданный порог. Разумеется, в реальности этот порог выбирается из статистического анализа и/или ребро проводится с учетом структуры выравнивания. Здесь есть (и применяется) множество различных вариантов.

Теперь мы имеем граф. Можно предположить, что в этом графе белковому семейству отвечает *клика*, т.е. такой подграф, в котором все вершины соединены со всеми (полный подграф). Действительно, в пределах семейства каждый белок должен быть похожим (в смысле выбранной меры сходства) на любой другой белок семейства. Однако такой подход таит в себе ряд проблем. Во-первых, алгоритмическая проблема — поиск клики в графе является сложной задачей, для которой не существует эффективного алгоритма (см. главу 4). Во-вторых, есть содержательная проблема — клика не определяется однозначно — одна вершина (ребро, подграф) может принадлежать сразу нескольким кликам (см. рис. 4.3). Поэтому поиск клик в этой задаче не применяют.

Представляется более подходящим подход, основанные на поиске связной компоненты. Такой метод часто применяют. Действительно, связная компонента не обладает недостатками клики — есть эффективные алгоритмы поиска, каждая вершина принадлежит одной и только одной связной компоненте. Для каждой связной компоненты можно оценить ее "качество— степень насыщенности ребрами. Если количество ребер равно $|V| \cdot (|V|-1)/2$, то, как не трудно догадаться, эта связная компонента является кликой. Правда, появляется другая содержательная проблема. Посмотрите на рис.4.4б и 4.4в. Эти графы имеют одинаковое количество вершин и ребер, но интуитивно ясно, что граф б) представляет два семейства. Поэтому связная компонента не лучший способ искать семейства. Кстати, этот случай соответствует выравниванию на рис.4.5, т.е. случаю, когда многодоменные белки объединяют в одно семейство группы однодоменных белков. Поэтому представляется более рациональным использовать для этой задачи поиск двусвязной компоненты. Такой подход допускает ситуацию, когда вершина может принадлежать двум или более двусвязным компонентам. Этот недостаток на самом деле является достоинством, поскольку позволяет находить многодоменные белки.

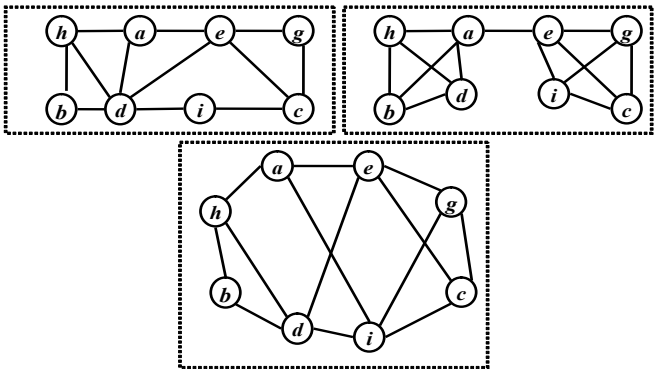


Рис. 4.4: граф для семейства белков. а) Вершина ребро hd принадлежит двум кликам, вершина e принадлежит двум другим кликам. б) связный граф, который очевидно соответствует двум разным семейства. в) Граф, который соответствует одному семейству

4.4 Задача Эйлера

Многим известно, что теория графов зародилась в 18 веке после того, как Леонарда Эйлера заинтересовала задача о семи Кенигсбергских мостах: как пройти по всем мостам, не проходя ни по одному из них дважды (рис.4.6 4.5). Отсюда и задача Эйлера, которая формулируется следующим образом: *найти на графе цикл, проходящий через все ребра, причем по каждому ребру один раз.* Эйлеровым циклом называется цикл в графе такой, что он проходит через все ребра графа по одному разу. Если в графе есть эйлеров цикл, то граф называется эйлеровым, или, говорят, что граф обладает свойством эйлеровости. Понятно, что эйлеровым может быть только связный граф. Для любого связного графа можно сразу сказать, обладает он свойством эйлеровости, или нет.

В связном неориентированном графе есть эйлеров цикл тогда и только тогда, когда индексы всех вершин четные.

В связном ориентированном графе есть эйлеров цикл тогда и только тогда, когда индексы всех вершин равны нулю.

Доказательство необходимости лежит на поверхности: рисуя эйлеров граф, мы пройдем через каждое ребро лишь однажды, а значит, если ребро привело нас в некую вершину, сможем выйти из нее только по другому ребру. Из этого следует, что, сколько ребер входит в вершину, столько из нее и выходит — значит, вершина четная. Достаточность четности индекса вершин доказывается ниже.

Если мы не требуем условия замкнутости пути, возникает эйлеров путь — путь, содержащий все ребра графа и такой, что по нему можно пройти, не проходя одно и то же ребро дважды. Требования для существования эйлерова пути мягче, чем для эйлерова цикла. В неориентированном гра-

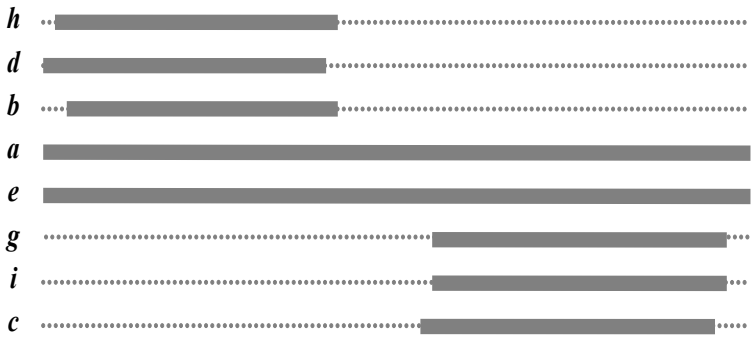


Рис. 4.5: Выравнивание, продуцирующее граф на рис.4.4б

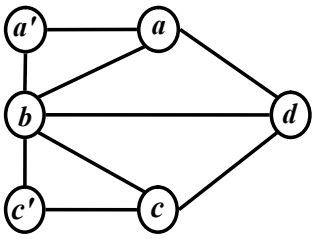


Рис. 4.6: Схема мостов Кенигсберга. Ребра графа — мосты. Вершины a' и c' являются дополнительными, поскольку между a и b и b и c есть по два моста, поэтому нет прямого отображения мостов на граф.

фе не более чем две вершины могут быть нечетными (начало и конец), все остальные обязаны быть четными для существования эйлерова пути в графе. Аналогично и с орграфом — все вершины должны иметь индекс 0, не более чем две вершины могут иметь нечетные индексы (отрицательные у начала и положительные у конца графа).

Как искать эйлеров путь и эйлеров цикл в графе, даже если мы знаем, что он там есть? С эйлеровым путем поступим, как любят делать математики (помните задачку про чайник?) — проведем ребро между нечетными вершинами и будем искать эйлеров цикл. Если между этими вершинами уже есть ребро, то вводим дополнительную вершину и соединяем нечетные (ненулевые) вершины ребрами с этой новой вершиной. Таким образом, перед нами стоит задача поиска эйлерова цикла в графе.

4.4.1 Алгоритм поиска Эйлерова цикла в графе

При поиске Эйлерова цикла основывается на наблюдении. Если в эйлеровом графе удалить ребра, образующие цикл, либо останется эйлеровым, ли-

бо распадется на эйлеровы графы. Действительно, после удаления такого цикла четность вершин либо не изменится (если эти вершины не участвуют в удаляемом цикле), либо уменьшится на 2, если вершина принадлежит циклу. Поэтому четность вершин после удаления цикла не изменится. Единственное, что может случиться неприятного — потеря связности графа. Нетрудно доказать от противного, что процедуру удаления можно повторять до тех пор, пока в графе не останется ребер. На основе этого наблюдения можно построить алгоритм. Но предварительно докажем лемму.

Лемма 1. *Если в графе все вершины четные, то через любую вершину проходит цикл.*

Доказательство. Начнем просто строить такой цикл. Пометим все ребра как не пройденные. Затем, начиная с выбранной вершины идем в глубину по непомеченным ребрам. При просмотре в глубину (в отличие от стандартного просмотра в глубину) мы помечаем ребра, а не вершины. В конце концов при таком просмотре мы упремся в ситуацию, когда из очередной вершины не идет непомеченных ребер. Покажем, что эта вершина совпадает со стартовой. Действительно, допустим это некоторая промежуточная вершина. Тогда либо из нее не выходит вообще ребер (эта вершина - тупик), и поэтому она не четная (одно ребро пришло и ни одного не вышло). Либо из этой вершины выходят несколько помеченных ребер. Тогда это значит, что через эту вершину мы проходили, т.е. входили в нее и выходили из нее. Следовательно количество помеченных ребер, выходящих из этой вершины - четное плюс еще одно ребро, по которому мы только что пришли, т.е количество ребер - нечетное. Таким образом при этом просмотре мы пришли в ту вершину, с которой стартовали. □

1. Помечаем все ребра, как не пройденные.

2. Выбираем произвольную вершину v.

3. Идем в глубину, начиная с вершины v. При этом помечаем не вершины а ребра. Поиск в глубину заканчивается, если из очередной вершины не выходит белых ребер. Согласно лемме мы получили цикл, который можно записать в список, который можно зациклить, установив ссылку из последнего элемента списка на первый элемент списка.

4. В построенном списке вершин находим вершину v, из которой выходят не помеченные ребра. Если такая вершина не найдется, то найденный цикл является эйлеровым. Действительно, если в построенном цикле нет вершин с не помеченными ребрами, а где-то в графе остались таковые, то граф не связный.

5. Повторяем поиск, начиная с вершины v. Найденный цикл записываем в новый связный список и вставляем новый список в основной список на место вершины

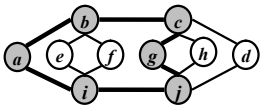
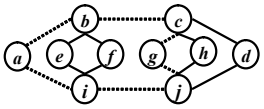
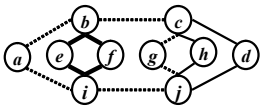
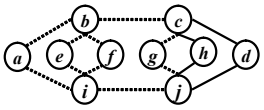
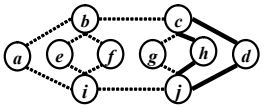
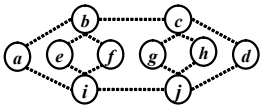
Псевдокод алгоритма:

```
1. Euler(G){
2.   List L;
3.   Vertex v=<any vertex>;
4.   do{
5.     List L1;
6.     SimpleCycle(v,L1);
7.     insert(L,L1,v);
8.     for(v=L.root;v!=null;v=v.next)
9.       if(v has white edges) break;
10.   }while(v!=null)
11.   return L;
12. }

13. SimpleCycle(v,L1){
14.   L1.add(v);
15.   for(e in edges_from_v){
16.     if(e.color==white){
17.       e.color=black;
18.       if(e.v1==v)
19.         SimpleCycle(e.v2,L1);
20.       else
21.         SimpleCycle(e.v1,L1);
22.       break;
23.     }
24.   }
25. }
```

Пояснения: Алгоритм содержит процедуру построения простого цикла (после строки 13). В строке 2 создаем пустой список L. Затем выбираем произвольную вершину v(строка 3). Здесь не описано как мы это делаем, поскольку это зависит от организации данных. Далее в цикле перебираем все вершины, имеющие белые ребра (цикл в строках 4-10). Внутри цикла мы организуем новый список L1, куда запишем результаты очередного поиска в глубину. Затем вставляем найденный цикл (список L1) в исходный список L (строка 7). Далее в полном списке L ищем вершину v с белыми ребрами (строки 8-9). Если такая вершина не найдена, то заканчиваем работу (строка 10). Процедура поиска цикла, начиная с заданной вершины (строка 13). Мы добавляем стартовую вершину в список. Затем ищем белое ребро из этой вершины. Как только находим такое ребро (строка 16), красим это ребро в черный цвет и с противоположной вершины этого ребра продолжаем поиск (строки 18-21). Для оптимизации работы можно на вершинах поддерживать количество исходящих белых ребер и список вершин, которые имеют белые ребра — чтобы не искать каждый раз вершину

с белыми ребрами. В таблице 4.4.1 показано как это работает.

Таблица 4.3: Пример поиска Эйлерового цикла.

граф	список вершин	
		Стартуя с вершины `a` проходим в глубину по ребрам. В момент покраски вершины `i` в черный цвет у нее серый сосед - стартовая вершина `a`. В списке `L1`: `abcgjia`
	`abcgjia`	Помечаем ребра как пройденные.
	`abcgjia`	В списке вершин остались вершины с ненулевым числом ребер. Стартуя с вершины, скажем, `i`, на-ходим цикл: `iebfi`
	`abcgj iebfi a`	и вставляем цикл `iebfi` на место.
	`abcgj iebfi a`	В списке вершин остались вершины с ненулевым числом ребер. Стартуя с вершины, скажем, `c`, находим цикл: `cdjhc`.
	`abcd jhcdj iabei a`	Вставляем в список на место вершины `c` цикл `cdjhc`. Помечаем ребра как пройденные.

4.4.2 Биоинформатика и задача Эйлера: секвенирование геномов

Современная стратегия секвенирования геномов подразумевает массовое прочтение случайных фрагментов генома и затем сборку генома. Характерная длина фрагмента — 300-800 нуклеотидов. Дальше возникает задача восстановления полной последовательности генома. Дальше возникает задача сборки генома — необходимо прочитанные фрагменты упорядочить. Поскольку читаются случайные фрагменты, то многие из них пересекаются (имеют общие фрагменты). Эти общие фрагменты и являются ключом

для сборки генома. Итак, можно построить граф: вершины — прочитанные фрагменты, ребра проводятся, если фрагменты имеют общий кусок. Задача заключается в том, чтобы провести на этом графе путь, проходящий через все вершины. К сожалению, секвенирование случайных фрагментов может привести к не связному графу, поскольку по случайным причинам некоторые участки генома могут оказаться непокрытыми прочитанными фрагментами. Другая проблема — задача поиска пути, проходящего через все вершины графа — задача Гамильтона, является трудной задачей, для которой не существует в принципе эффективного алгоритма (эта проблема будет обсуждаться в заключительной главе книги). Однако задачу сборки генома можно сформулировать как задачу Эйлера. Для этого разобьем все последовательности на блоки так, чтобы в пределах одного блока было сходство с фиксированным набором фрагментов. Например, фрагмент A мы разобьем на 4 блока — первый блок (A1) не имеет сходства с другими фрагментами (множество сходных фрагментов пусто). Блок A2 имеет сходство только с фрагментом H. Блок A3 имеет сходство с фрагментами H B. Последний блок имеет сходство с фрагментами H, B, C. Ребра проводим, если два блока соприкасаются. Направление ребер определяется последовательностью блоков. Ясно, что для того, чтобы объяснить экспериментальные данные, надо найти путь в этом графе, который проходит через все ребра. В простейшем случае наш граф представляет собой просто линейку из блоков. К сожалению, в реальности такого практически никогда не бывает. Ситуация, представленная на рис.4.7 — это еще не самое худшее, что встречается. В задаче сборки геномов главную проблему представляют повторы. Появление повтора приводит к появлению разветвлений в графе. В случае повтора один фрагмент будет накладываться на разные другие фрагменты, что приведет к появлению циклов в графе. Граф, представленный на рис.4.7 заведомо не имеет Эйлерового пути, поскольку имеет 5 вершин с ненулевым индексом. Это, скорее всего, отвечает случаю, когда прочитано два независимых фрагмента, но эти фрагменты имеют сходные участки. Если бы блоки d, b1, и a3 имели сходство, то надо было бы их склеить. Тогда граф имеет Эйлеров путь, но, к сожалению, он не однозначен. Глядя на рис.4.7 справа внизу, можно построить несколько Эйлеровых путей, например, $A4 \to Z \to b2 \cdots \to Z \to H4 \to Z \to g2 \to Z \to C4 \cdots \to Z \to F4$, или $A4 \to Z \to H4 \to Z \to C4... \to Z \to b2... \to Z \to g2 \to Z \to F4$. Блок Z представляет из себя повтор. Реальна ситуация осложняется еще тем, что последовательности читаются не совсем точно, поэтому при сборке необходимо учитывать качество прочтения того или иного фрагмента.

4.5 Топологическая сортировка

Дан ориентированный граф. **Задача:** перечислить вершины в таком порядке, чтобы из вершины с большим номером не шло ребер в вершины с меньшим номером, иными словами надо спроецировать вершины на прямую так, чтобы стрелки графа шли только направо. Если две вершины в графе

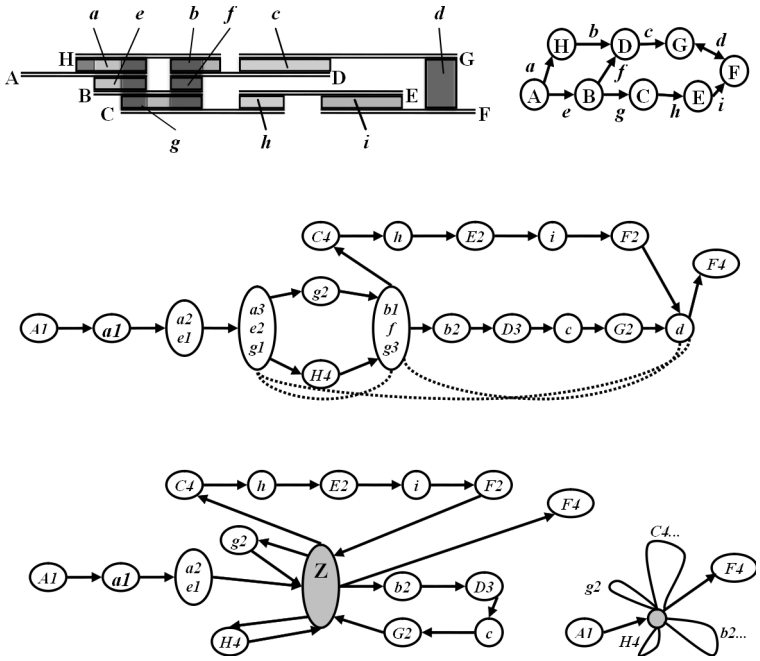
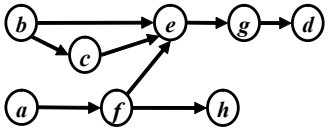


Рис. 4.7: Графы, порождаемые задачей сборки геномов. Слева наверху — наложение фрагментов и отмеченные участки сходства, справа — граф для Гамильтоновой постановки, в середине и внизу — граф для Эйлеровой постановки. Внизу справа показана топология графа. Обозначения: Прописные буквы - фрагменты, строчные - области перекрытия. Индекс — номер соответствующего блока. Например, `a2` — это второй блок в области перекрытия `a2`, а `A1` — область фрагмента `A` без перекрытия. Пунктиром на втором рисунке обозначено сходство областей — область `{a3,e2,g1}` совпадает с областью `{b2,f,g3}` и с областью `g`. На нижнем рисунке вершина `Z` собирает все области повтора.

независимы (нельзя сказать, какая из них стоит "позже"другой), их взаимное положение в списке неважно. Однако они могут быть упорядочены опосредовано, например, если между двумя вершинами нет непосредственно ребра, но между ними есть вершина X такая, что из первой вершины есть ребро в X, а из вершины X есть ребро во вторую вершину. Задача топологической сортировки является важной составной частью общей задачи планирования. Если у Вас есть список взаимосвязанных дел, то надо составить порядок их исполнения. Какие-то дела можно делать только после того, как завершатся некоторые другие дела, а выполнение каких-то других дел не связано друг с другом. Эту ситуацию можно представить в виде ориентированного графа. Вершины — это дела, ребра — связи. Будем проводить ребро из вершины A в вершину B , если дело B можно делать

только после дела A. Ясно, что связи должны быть устроены так, чтобы в полученном ориентированном графе не было циклов. Высокий пилотаж бюрократии заключается в создании циклов в таком графе дел.

Основное правило проецирования: Если из вершины A в вершину B идет ребро, то вершина B обязательно должна стоять в списке позже A. Следовательно, для графа из двух вершин вида $A \to B$ список будет AB. Если две вершины в графе независимы (нельзя сказать, какая из них стоит "позже"другой), их взаимное положение в списке неважно. Спроецируем согласно приведенным правилам такой граф:



Сортированный список будет:

```
a b c f e g h d
```

или

```
b a f c h e g d
```

Оба варианта допустимы. Дело в том, что вершины a и b, h и e, h и g и другие независимые вершины можно менять местами. Главное, чтобы вершина f всегда стояла после a, e после c, d после f — вершины в списке должны находиться согласно направлению ребер графа. Можно придумать несколько вариантов таких записей, и все они будут верными. Таким образом, ациклический ориентированный граф всегда можно спроецировать на список вершин, при этом часто верных проекций может быть несколько. Описанный выше способ проецирования графа на список его вершин называется *топологической сортировкой*.

Смотря на граф, на направление ребер в нем, мы сортируем список его вершин надлежащим образом. Рассмотрим топологическую сортировку на наглядном примере. У нас есть неупорядоченный список предметов одежды (список вершин графа). Пусть у нас есть проблема — надо надеть следующие вещи:

- Пиджак
- Брюки
- Галстук
- Рубашка
- Ремень
- Ботинки
- Носки

- Часы

Ясно, что некоторые вещи нельзя надеть раньше других. Например, попробуйте надеть носки после того, как надеты ботинки. Построим ориентированный граф по принципу "что за чем можно надевать". Если вещь a надевается раньше вещи b, ребро идет из в b. Получаем такой граф (рис.4.8).

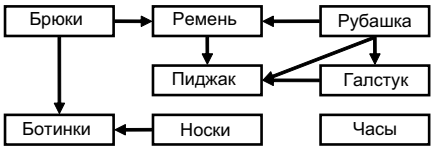


Рис. 4.8: Граф процесса одевания.

4.5.1 Алгоритм топологической сортировки

1. Берем произвольную вершину графа, начинаем с нее поиск в глубину
2. При поиске в глубину, когда вершина чернеет, добавляем ее в список, за ней почерневшую следующей и так далее.
3. Когда все вершины почернели полученный список разворачиваем.

Реализуем приведенный алгоритм на нашем "одежном"примере. Берем неупорядоченный список вершин и начинаем обход в глубину соответствующего графа с произвольной вершины, к примеру, "`брюки`". Обход в глубину дает нам следующий порядок вершин:

`Ботинки` $\rightarrow$ `Пиджак` $\rightarrow$ `Ремень` $\rightarrow$ `Брюки`

Остались не просмотренные вершины. Стартуем с "`рубашки`". Получаем:

`Ботинки` $\rightarrow$ `Пиджак` $\rightarrow$ `Ремень` $\rightarrow$ `Брюки` $\rightarrow$ `Галстук` $\rightarrow$ `Рубашка`

Продолжая процедуру получаем наконец:

`Ботинки` $\rightarrow$ `Пиджак` $\rightarrow$ `Ремень` $\rightarrow$ `Брюки` $\rightarrow$
`Галстук` $\rightarrow$ `Рубашка` $\rightarrow$ `Носки` $\rightarrow$ `Часы`

Разворачивая список находим, наконец порядок надевания вещей:

`Часы` $\rightarrow$ `Носки` $\rightarrow$ `Рубашка` $\rightarrow$ `Галстук` $\rightarrow$ `Брюки` $\rightarrow$
`Ремень` $\rightarrow$ `Пиджак` $\rightarrow$ `Ботинки`

Топологическая сортировка часто используется при планировании каких-то мероприятий, экспериментов. В частности, она используется в качестве предварительного этапа алгоритма динамического программирования поиска оптимального пути в графе.

Упражнение 26. *Напишите псевдокод процедуры топологической сортировки.*

4.6 Поиск оптимального пути в графе

Есть множество побуждающих примеров для постановки поиска оптимального пути в графе. Это может быть и самый короткий путь (в смысле километров), и самый быстрый — он не обязательно самый короткий — на коротком пути может быть пробка, и самый дешевый — дорога может быть такой, что проезд по ней обойдется большим ремонтом. Этого рода задачи возникают сплошь и рядом при маршрутизации пакетов в компьютерной сети, в частности в Интернете. В любом случае нам важно, что:

1. дан ориентированный граф;
2. на ребрах графа определены веса (т.е. граф взвешенный);
3. Вес пути определяется как сумма весов ребер, через которые проходит этот путь.

Задача состоит в том. чтобы найти путь минимального (или максимального) веса.

Рассмотрим один из примеров, возникающих в биоинформатике, когда необходимо найти оптимальный путь. В результате секвенирования ДНК (например, генома) получен ряд последовательностей, которые надо собрать в полный геном. Тогда поступим следующим образом. Строим граф, вершина — прочитанная последовательность. Ребро проводится, если суффикс первой последовательности равен префиксу второй. Вес равен длине общей части. Путь с наибольшим весом дает последовательность генома.

Задача: дан связный ориентированный взвешенный (т.е. на каждом ребре графа написано неотрицательное число) граф.

Найти: путь между двумя вершинами, имеющий минимальный (максимальный) вес.

4.6.1 Динамическое программирование для поиска оптимального пути

Эта задача решается по-разному в зависимости от некоторых дополнительных условий. Итак, задача. Дан *ациклический* ориентированный взвешенный граф. Задача — найти путь минимального (максимального) веса между двумя вершинами (B и E).

Рассмотрим некоторую вершину v. Допустим, мы знаем веса w_i оптимальных путей до всех вершин, непосредственно предшествующих вершине v, т.е. для всех вершин $u_i \in \{prev(v)\}$, из которых есть ребра $e_i = (u_i, v)$.

Тогда несложно вычислить вес оптимального пути, ведущего в вершину v. Действительно, вес пути, проходящего через вершину u_i равен $w(B, u_i, v) = w_i + w(e_i)$. Тогда наименьший вес пути из начальной вершины B в вершину v равен:

$$w(B, v) = \min_i\{w(B, u_i, v)\} = \min_i\{w_i + w(e_i)\}$$

Это соображение позволяет построить алгоритм поиска веса оптимального пути из вершины B в вершину E. Припишем каждой вершине v вес $w(v)$, равный весу оптимального пути из начала B в эту вершину. Ясно, что вес, приписанный вершине B равен 0, поскольку путь из B в B не содержит ни одного ребра. Дальше мы можем просмотреть всех соседей вершины B и определить вес оптимального пути в них. Когда мы обработаем вершину E, мы найдем вес оптимального пути. *На самом деле это не совсем так.* Рассмотрим фрагмент графа на рис.4.9. Если мы будем подсчитывать вес

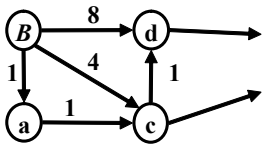


Рис. 4.9: Побудительный пример для необходимости топологической сортировки.

для вершины d, не определив веса у вершины c, то мы положим ее равной 8. Хотя очевидно, что оптимальный путь в вершину проходит через вершины a и c. С другой стороны, если мы будем сначала считать оптимальный вес для вершины c, а потом для вершины d, то мы получим вес 5, что также не является оптимальным весом. А правильный порядок вычисления весов такой: сначала идет вершина a, потом вершина c и только затем вершина d. Таким образом, порядок обхода вершин должен быть таковым, чтобы сначала были обработаны вершины, которые не зависят от последующих вершин, иными словами необходимо провести топологическую сортировку вершин. При этом все вершины, которые в результате топологической сортировки перед вершиной B, равно и вершины, стоящие после вершины E, можно исключить из рассмотрения. Из сказанного ясно, почему для такого алгоритма необходимо, чтобы граф был ациклическим. Итак, алгоритм:

1. Провести топологическую сортировку вершин.
2. Отбрасываем все вершины, предшествующие стартовой вершине B и следующие за вершиной E.
3. В цикле по сортированным вершинам определяем вес вершин, как минимум сумм весов непосредственно предшествующих вершин и ребер. После обработки последней вершины (E) мы определим вес оптимального пути.

Описанный алгоритм находит только вес оптимального пути, но при этом у нас нет оптимального пути (а хотелось бы...). Для того, чтобы найти оптимальный путь надо, чтобы в вершинах графа был еще один атрибут — ссылка на вершину на которой достигается минимум:

$$\pi(v) = \text{argmin}_i \left(w\left(u_i\right) + w\left(u_k(v)\right)\right)$$

Обратим внимание на то, что сеть оптимальных переходов образует дерево. Алгоритм, основанный на рекурсивном вычислении целевой функции, называется алгоритмом динамического программирования. Алгоритмы динамического программирования применяются не только при оптимизации на графах, но и при оптимизации других объектов. Для того, чтобы восстановить путь надо провести обратный просмотр. Начинаем с вершины E. Ее атрибут $\pi(E)$ скажет нам откуда мы пришли в вершину E. Переходя в соответствии со ссылкой в новую вершину $v_1 = \pi(E)$ определим ее предшественника $v_2 = \pi(v_1)$, и т.д.

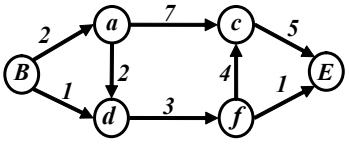
Псевдокод алгоритма:

```
1.  DinProg(B,E){
2.     Vertex v[]=Topol_Sort();
3.     int ib, ie;
4.     for(i=0; i<v.length; i++){
5.        if(v[i]==B) ib=i;
6.        if(v[i]==E) ie=i;
7.     }
8.     if(ie < ib) return "no path";

9.     v[ib].weight=0;
10.    for(i=ib+1; i <= ie; i++){
11.       v[i].weight=INFINITY;
12.       for(each w in v.previous){
13.          ww=w.weight+weight(w,v[i]);
14.          if(ww < v[i].weight) {
15.             v[i].weight=ww;
16.             v[i].pi=w;
17.          }
18.       }
19.    }

20.    Stack path;
21.    w=E;
22.    while(w!=B){
23.       path.push(w);
24.       w=w.pi;
25.    }
26.    return path;
```

```
27. }
```

Пояснения. Программа состоит из трех этапов: топологическая сортировка и анализ исходных данных (сроки 2-8). В строке 8 определяется существует ли путь из вершины B в вершину E. В строках 9-19 происходит поиск оптимального веса. Строка 9 означает, что оптимальный вес пути из начала в начало равен 0. Строки 11-17 находят оптимальный вес для вершины `v[i]`, в строке 16 запоминаем оптимальный путь. Строки 20-25 посвящены обратному просмотру и восстановлению пути. Путь запоминается в стеке.

В таблице 4.6.3 приведен пример работы алгоритма динамического программирования для графа:



Заметим, что нам на самом деле неважно, положительные или отрицательные веса заданы на ребрах. Более того, этот алгоритм применим как для минимизации веса пути, так и для его максимизации — стоит только заменить операцию min на max.

4.6.2 Оценка времени работы динамического программирования

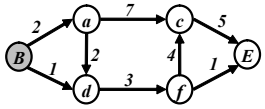
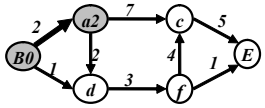
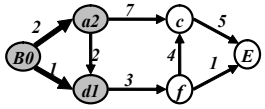
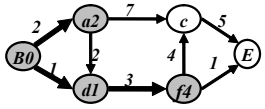
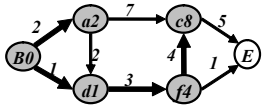
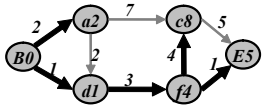
Оценим время работы алгоритма. Топологическая сортировка требует времени порядка $T = O(|V|+|E|)$. Цикл в строках 4-7 обрабатывается за время $T = O(|V|)$. Время работы цикла 10-19 составляет $T = O(|V|+|E|)$, поскольку в этом цикле (при правильной организации данных) просматривается каждая вершина и каждое ребро по одному разу. Время обратного прохода равно длине оптимального пути. В худшем случае $T = O(|V|)$. Итого:

$$T = O(|V| + |E|) + O(|V|) + O(|V| + |E|) + O(|V|) = \ = O(|V| + |E|)$$

4.6.3 Биоинформатические применения поиска оптимального пути в графе. Выравнивание

Задача выравнивания последовательностей возникает во многих задачах биоинформатики и является одной из основных классических задач биоинформатики. На самом деле эта задача возникла намного раньше как задача определения редакционного расстояния. Редакционным событием будем называть одно из действий: удаление символа из слова, вставка символа в слово, замена символа. Редакционным расстоянием между словами называется минимальное количество редакционных событий переводящих одно

Таблица 4.4: Пример динамического программирования для поиска оптимального пути в графе.

	B(0); a(∞,?); d(∞,?); c(∞,?); f(∞,?); E(∞,?);	Веса всех вершин, кроме начальной не определены.
	B(0); a(2, B); d(∞,?); c(∞,?); f(∞,?); E(∞,?);	Обрабатываем вершину a. У нее единственный предшественник — вершина B. $$w(a) = w(B) + w(Ba) = \\ = 0 + 2 = 2$$
	B(0); a(2, B); d(1, B); c(∞,?); f(∞,?); E(∞,?);	Вершина d — два предшественника a и B. $$w(d) = \min\{(w(B) + w(Bd)), \\ (w(a) + w(ad))\} = \\ = \min\{(0 + 1), (2 + 2)\} = 1$$
	B(0); a(2, B); d(1, B); c(∞,?); f(4, d); E(∞,?);	Вершина f — один предшественник d. $$w(f) = w(d) + w(df) = \\ = 1 + 3 = 4$$
	B(0); a(2, B); d(1, B); c(8, f); f(4, d); E(∞,?);	Вершина c — два предшественника a и f. $$w(c) = \min\{(w(a) + w(ac)), \\ (w(f) + w(fc))\} = \\ = \min\{(4 + 4), (2 + 7)\} = 8$$
	B(0); a(2, B); d(1, B); c(7, f); f(4, d); E(f, 5);	Вершина E — два предшественника c и f. $$w(E) = \min\{(w(c) + w(cE)), \\ (w(f) + w(fE))\} = \\ = \min\{(5 + 7), (4 + 1)\} = 5$$

слово в другое. Задача о редакционном расстоянии возникает во многих контекстах. Например, система исправления опечаток может пытаться построить выравнивание напечатанного слова с одним из слов словаря и тем самым исправить опечатку. Аналогичные алгоритмы используются при поиске плагиата. В биологии задача имеет трактовку: найти минимальное количество элементарных эволюционных событий.

Для решения задачи построим граф. Вершины графа соответствуют сопоставлению префиксов слов. Ребра графа — редакционные события, которые удлиняют один или оба префикса, т.е. добавляют символ в одну последовательность (а в другую — добавляют пропуск символа) или в другую последовательность, или в обе последовательности. Тогда можно построить граф для определения редакционного расстояния и найти кратчайший путь в этом графе. Итак, определим редакционное расстояние для слов `aacggatcg` и `aaggattcg`. Граф редакционных расстояний организован в виде таблицы (рис.4.10). Горизонтальные переходы порождают символы в горизонтальной последовательности, или (что то же самое) пропуск в вертикальной последовательности. Вертикальные стрелки соответствуют вставке в вертикальной последовательности (пропуск в горизонтальной последовательности). Этим переходам всегда соответствует одно событие редактирования. Диагональные стрелки соответствуют сопоставлению символов (они либо совпадают, либо различаются). В зависимости от того, совпадают соответствующие символы или нет вес этих строк будет либо 0 (нет события редактирования) либо 1 (редактирование — замена). Важно, что в этом графе есть строки и столбцы, которые не соответствуют символам последовательности (отмечены серым). Эти строки (ряды) соответствуют краевым делециям. Любой путь на этом графе соответствует некоторой последовательности редакционных событий, или, что то же самое, выравниванию.

Упражнение 27. *Какому выравниванию соответствует путь: сначала по верхней горизонтальной строке до конца, затем по правому вертикальному столбцу до конца?*

Упражнение 28. *Есть ли необходимость проводить для этого графа топологическую сортировку?*

Таблица 4.6.3 показывает вычисление редакционного расстояния между последовательностями. Оно оказывается равным 2 — есть вставка в последовательность 1 и вставка в последовательность 2:

```
aacggat-cg
aa-ggattcg
```

4.6.4 Полукольцо. Динамическое программирование над полукольцом

Алгоритм динамического программирования использует рекурсию

$$w(B,v) = \min_{i:u_i \in \{prev(v)\}} \{w(B,u_i,v)\} = \min_i \{w_i + w(e_i)\}$$

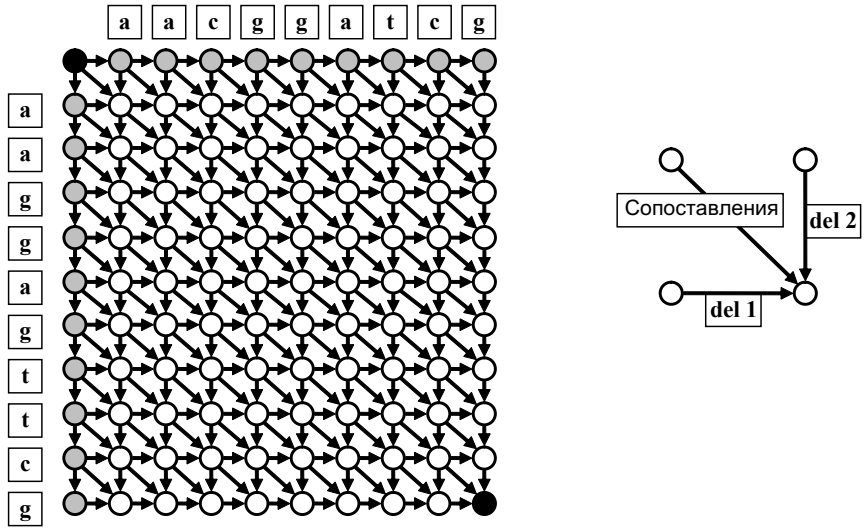


Рис. 4.10: Динамическое программирование для определения редакционного расстояния. Черным отмечены стартовая и финальная вершины.

Операцию минимума по всем вариантам на самом деле есть обобщение бинарной операции минимума: берем первые два элемента, вычисляем минимум, затем вычисляем минимум от полученного значения и следующего элемента т т.д.:

$$\min_i\{w_i + w(e_i)\} = \min\left(v_1 + e_1, \min\left(v_2 + e_2, \min\left(\cdots\right)\right)\right)$$

Минимум на самом деле является бинарной операцией, такой же как, скажем сложение. Если мы введем странные обозначения, а именно вместо $+$ будем писать $\otimes$, а вместо min будем писать $\oplus$ то можно переписать это выражение в виде:

$$\begin{aligned}&\min\left(v_1 + e_1, \min\left(v_2 + e_2, \min\left(\cdots\right)\right)\right) = \\ &= (v_1 \otimes e_1) \oplus (v_2 \otimes e_2) \oplus \cdots \oplus v_n = \\ &= \bigoplus_i (v_i \otimes e_i)\end{aligned}$$

Итак, у нас при рекурсии используем две операции — сложение и взятие минимума. При этом нам важны некоторые вещи, а именно хотелось бы, чтобы, например, было свойство $a \oplus (b \oplus c) = (a \oplus b) \oplus c$. Принципиально важно, чтобы вес пути был суммой весов подпутей и, наверное, ряд других свойств. Итак,

Определение 2. *Полукольцом называется множество M на котором определены две бинарные операции $\oplus$ и $\otimes$ обладающие свойствами:*

Таблица 4.5: Пример динамического программирования для поиска оптимального пути в графе.

		a	a	c	g	g	a	t	c	g
	0	→1	→2	→3	→4	→5	→6	→7	→8	→9
a	↓1	↘0	→1	→2	→3	→4	→5	→6	→7	→8
a	↓2	→1	↘0	→1	→2	→3	→4	→5	→6	→7
g	↓3	→2	↓1	↘1	↘1	→2	→3	→4	→5	→6
g	↓4	→3	↓2	↓2	↘1	↘1	→2	→3	→4	→5
a	↓5	→4	↘3	↓3	↓2	↘2	→1	→3	→4	→5
t	↓6	→5	↓4	↘4	↘4	→3	↘2	→1	→2	→3
t	↓7	↓6	↓5	↘5	↘5	↓4	↓3	↘2	→3	→4
c	↓8	↓7	↓6	↘5	↘6	↓5	↓4	↓3	↘2	→3
g	↓9	↓8	↓7	↓6	↘5	↘6	↓5	↓4	↓3	↘2

Таблица 4.6: Примеры полуколец.

Множество	$\oplus$	$\otimes$	**0**
целые неотрицательные числа	$+$	$\times$	0
квадратные матрицы	$+$	$\times$	нулевая матрица
true, false	or	and	false
R, $+\infty$	min	$+$	$+\infty$
R, $-\infty$	max	$+$	$-\infty$

- $a \oplus b$ — *коммутативность по "сложению"*
- $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ — *ассоциативность по "сложению"*
- $a \otimes (b \otimes c) = (a \otimes b) \otimes c$ — *ассоциативность по "умножению"*
- $\exists 0 : a \oplus 0 = a$ — *Существование 0*
- $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$ — *правая дистрибутивность*
- $(a \oplus b) \otimes c = a \otimes b \oplus a \otimes c$ — *левая дистрибутивность*

Таким образом, динамическое программирование можно применять не только для поиска минимума или максимума, но и для решения других задач. Вопрос только каких ... Возьмем, к примеру, полукольцо неотрицательных целых чисел с естественными операциями. На ребрах определим значение веса = 1. В стартовой вершине B определим вес 1. Тогда оптимальный вес, подсчитанный в вершине E, даст нам число путей из B в E. Действительно, путь из B в B единственный. Допустим все на всех вершинах u_i, предшествующих вершине v определено количество путей из B. Тогда количество путей, ведущих в вершину v равно сумме чисел путей в предшествующие вершины. На самом деле существуют более изощренные

способы применения полуколец при подсчете различных величин. Упомянем подсчет статистических сумм или полных вероятностей. Эти задачи подробно обсуждаются в теории скрытых Марковских моделей.

4.6.5 Поиск минимального пути в графе, содержащем циклы. Алгоритм Дейкстры

Представим себя водителем автомобиля, пытающегося проехать от ФББ до станции метро "Университет"в условиях сильных пробок. Составим план дорог, соединяющих эти места (рис.4.11). Кружками на нем показаны вершины "графа объезда— наиболее важные пункты (ФББ и метро — тоже вершины). Стрелки — это ребра графа, показывающие всевозможные прямые и окольные пути между пунктами-ребрами. На ребрах, вообще говоря, есть веса (варианты: сложность объезда, длина пути, вероятность пробки). Все это вместе образует ориентированный взвешенный граф в котором есть циклы. Чтобы "с комфортом"добраться от ФББ до метро, хорошо было бы найти оптимальный путь в этом графе.

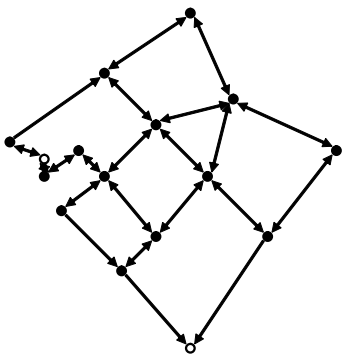


Рис. 4.11: Схема проезда в районе Университета.

Выше мы упоминали, что динамическое программирование на циклическом графе невозможно, поскольку для такого графа нельзя сделать топологическую сортировку, служащую во многом основой динамического программирования. Для построения такого алгоритма нам потребуется новая структура данных — *очередь с приоритетами*. Элементы очереди с приоритетами имеют дополнительный параметр — приоритет. Вход элемента в очередь такой же, как в обычной очереди. Однако из очереди снимается элемент с наивысшим приоритетом. Если есть несколько элементов с одинаковыми приоритетами, то снимется тот элемент, который пришел раньше. Это то, как работает обычная очередь (например, в сбербанке), когда ветераны имеют право внеочередного обслуживания. Если одновременно придет несколько ветеранов, то они образуют свою очередь, которая обслуживается прежде всего. В компьютерной очереди с приоритетами таких

приоритетов может быть много.

Нетрудно догадаться, что очередь с приоритетами может быть реализована с помощью двоичного дерева поиска (лучше красно-черного). Основой для построения двоичного дерева поиска является операция сравнения, которая позволяет сравнивать два любых элемента и в зависимости от результатов сравнения кладет или ищет элемент в правом или в левом поддереве. Введем понятие времени входа элемента в очередь. Это просто порядковый номер появления элемента в очереди. В начале счетчик равен нулю. Когда приходит очередной элемент, он приобретает номер, равный текущему значению счетчика, а счетчик увеличивается на 1. Теперь определим сравнение. Элемент больше элемента , если приоритет элемента a меньше приоритета b, а если приоритеты равны, то больше тот элемент, номер входа которого больше.

```
1. int Compare (a,b){
2.   if(a.pri < b.pri) return 1;
3.   if(a.pri = b.pri) {
4.   if(a.num < b.num) return 1;
5.   if(a.num > b.num) return -1;
6.   return 0;
7.   }
8.   return -1;
9. }
```

Пояснение. Эта подпрограмма возвращает 1, если $a > b$, -1, если $a < b$ и 0, если $a = b$. Отметим, что последнее условие никогда не должно выполняться, поскольку все элементы имеют заведомо разные номера. Теперь можно построить двоичное дерево поиска, основанное на этом сравнении элементов. Если мы будем добавлять в это дерево элементы с использованием описанного сравнения, а снимать с дерева наименьший элемент, то это и будет очередь с приоритетами.

Так же, как и в динамическом программировании, в каждой вершине будем записывать вес оптимального (или какого-либо) пути из начала в эту вершину. В начальный момент все вершины, кроме начальной имеют вес, равный ∞. Стартовая вершина имеет вес, равный 0. Основной промежуточный шаг алгоритма поиска оптимального пути — это релаксация одной вершины относительно другой. Если вес вершины v больше суммы веса вершины u и веса ребра $e(uv)$, то вес вершины v приравнивается этой сумме, и в вершине v запоминается переход на вершину u.

```
1.  Relax(v,u){
2.     if(u.d+w(uv) < v.d){
3.        v.d= u.d+w(uv);
4.        v.pi=u;
5.     }
6.  }
```

Теперь опишем алгоритм Дейкстры поиска кратчайшего пути. Он очень прост и несколько напоминает алгоритм обхода графа в ширину (за исключением того, что используется очередь с приоритетами). В качестве приоритета используется текущий вес вершины. Алгоритм Дейкстры:

1. Все вершины необработанны (белые, вес ∞), кроме последней (серая, стоит в очереди, вес 0).
2. Снимаем вершину из очереди, красим в черный цвет.
3. Для всех соседей этой вершины (в том числе и серых) делаем Relax, и, если они не серые, то помещаем в очередь.
4. Если очередь пуста, заканчиваем обработку, иначе переходим к п.2

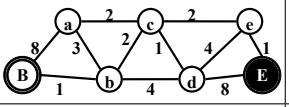
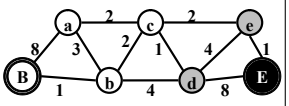
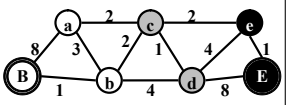
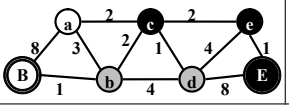
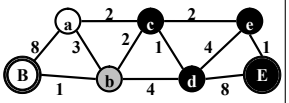
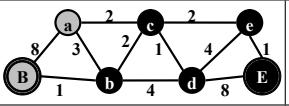
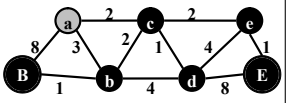
Отметим, что в процессе работы алгоритма вершины, стоящие в очереди могут менять приоритеты. Поэтому после релаксации следует вершину переместить в правильное место очереди. На самом деле для реализации очереди с приоритетами используется другая, более удобная структура данных, но ее изучение выходит за рамки курса.

Псевдокод:

```
1.  Deikstra(){
2.     PriQueue q;
3.     q.put(E); E.color=gray; E.d=0;
4.     do{
5.        v=q.get();
6.        for(each u in v.neighbours){
7           if(u.color != black){
8.            Relax(u,v);
9.            if(u.color == gray)  q.correct(u);
10.           if(u.color == white){
11.                q.put(u);
12.                u.color=gray;
13.           }
11.         }
12.       }
13.       v.color=black;
14.    }while(v!=B);
15. }
```

Пояснение. Этот код предполагает, что есть как-то организованная очередь с приоритетами. Эта структура данных имеет стандартные для очереди операции `put` и `get`. Кроме того есть операция перестройки очередиcorrect — она необходима, если элемент, уже стоящий в очереди изменил свой уровень приоритета. В начале все вершины белые. Кладем в очередь

Таблица 4.7: Пример работы алгоритма Дейкстры.

Граф	Очередь и веса вершин	Действия
	E B: ∞, a: ∞, b: ∞, c: ∞, d: ∞, e: ∞, E: 0	В очереди стоит вершина E
	e(1) d(8) B: ∞, a: ∞, b: ∞, c: ∞, d: 8, e: 1, E: 0	Снимаем E из очереди и релаксируем всех соседей и помещаем в очередь (если они не там)
	c(3) d(5) B: ∞, a: ∞, b: ∞, c: 3, d: 5, e: 1, E: 0	Снимаем e из очереди и релаксируем соседей относительно нее
	d(4) b(5) B: ∞, a: ∞, b: 5, c: 3, d: 4, e: 1, E: 0	Снимем из очереди c.
	b(5) B: ∞, a: ∞, b: 5, c: 3, d: 4, e: 1, E: 0	Снимем из очереди d. Ни одна вершина не релаксировала
	B(6) a(8) B: ∞, a: 8, b: 5, c: 3, d: 4, e: 1, E: 0	Снимаем вершину b.
	a(8) B: 6, a: 8, b: 5, c: 3, d: 4, e: 1, E: 0	Снимаем вершину B. Поскольку вершина B обработана, то мы нашли оптимальный путь (путь через a не оптимален!)

конец нашего пути (строка 3). Потом снимаем из очереди очередной элемент (строка 5) и релаксируем относительно него всех не черных соседей — черные уже приобрели свой конечный вес (строки 6-12). После релаксации не забываем поправить очередь (строка 9) и положить белых соседей в очередь (строки 11, 12). Процедура повторяется до тех пор, пока из очереди не снимем начальную вершину пути. Поскольку процедура `Relax` запоминает оптимальный переход, то восстановление пути проблемы не составит.

Корректность работы алгоритма Дейкстры

Теорема 1. *По завершении обработки алгоритмом Дейкстры взвешенного ориентированного графа $G = V, E$ с неотрицательной весовой функцией w и истоком s для всех вершин $u \in V$ выполняется равенство $d(u) = \delta(s, u)$, где $d(u)$ — вес, подсчитанный в соответствии с алгоритмом Дейкстры, $d(u) = \delta(s, u)$ — вес кратчайшего пути.*

Доказательство. Для доказательства этого утверждения используют следующие соображения (см. рис. 4.12). Есть множество обработанных вершин (черные). Есть множество "пограничных"вершин (серые), которые имеют ребра к черным вершинам, есть множество белых вершин, которые не обрабатывались.

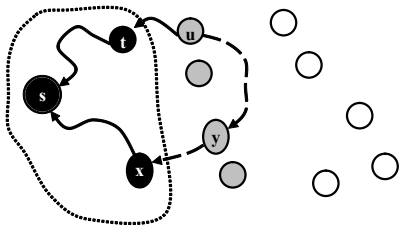


Рис. 4.12: К доказательству правильности алгоритма Дейкстры.

1. Никакое ребро в черную вершину не лучше отмеченных ребер, (поскольку серые — релаксированы).

2. Выберем вершину u, которая стоит первой в очереди. Она имеет минимальное значение d среди всех серых вершин (по принципу построения очереди). Допустим есть путь из u в s, лучше, чем путь $u \to t \to s$. Тогда этот путь обязательно пройдет через серую вершину $y : u \to y \to x \to s$, поскольку путь в конце концов войдет в черную область, а соседями черной области - серые. Но:

   $w(u \to y \to x \to s) = w(u \to y) + w(x \to s)$ — вес пути есть сумма весов подпутей

   $w(y \to x \to s) = d(y) > d(u) = w(u \to t \to s)$ — вес u минимален по построению очереди с приоритетами.

   $w(u \to y) > 0$ — веса положительны!

   $w(u \to y \to x \to s) > w(u \to t \to s)$ — **противоречие!**

□

Заметим, что поскольку допустимы циклы, то алгоритм Дейсктры можно применять и для неориентированных графов — неориентированный граф можно трактовать как ориентированный, где каждое неориентированное

ребро (vu) эквивалентно двум ориентированным ребрам (vu) и (uv). Кстати пример, приведенный в таблице 4.6.5 является неориентированным графом.

Упражнение 29. *Приведите пример графа с некоторыми отрицательными ребрами, когда алгоритм Дейкстры не даст оптимального решения.*

Время работы алгоритма Дейкстры

Время работы алгоритма Дейкстры на графе немного больше, чем время работы алгоритма динамического программирования. Каждая вершина проходит одну релаксацию относительно всех соседей, поэтому надо потратить время O(|E|). Но организация очереди стоит времени, причем перестройка очереди происходит при каждой релаксации (в худшем случае на каждом ребре). В худшем случае алгоритм работает за $T = O(|E| \cdot \log(|V|)$. Логарифм в выражении появляется потому, что мы пользуемся бинарным деревом для реализации очереди с приоритетами

4.6.6 Алгоритм Беллмана-Форда

Описанные ранее алгоритмы поиска оптимального пути имели ограничения — либо ациклический граф, либо только неотрицательные веса. Для случая применим следующий алгоритм.

Алгоритм Беллмана-Форда основывается на повторе релаксации всех ребер (в произвольном порядке) $|V| - 1$ раз. Вообще говоря, неочевидно, что такой алгоритм находит оптимальный путь. Докажем это.

Теорема 2. *Алгоритм Беллмана-Форда находит оптимальный путь в графе.*

Доказательство. Если есть оптимальный путь $\{v_1, v_2, \cdots, v_n\}$, то на i-шаге будет определена длина пути от i-вершины до конца. Доказательство по индукции. Инициация тривиальна — в начале мы имеем пустой список (тривиальный оптимальный путь из end в end нам известен заранее, это 0).

Пусть на i-шаге $d(v_i)$ равна длине кратчайшего пути до вершины E. Тогда на следующем шаге произойдет релаксация ребра (v_i, v_{i+1}) и таким образом определится вес оптимального пути от v_{i+1} до E.

После первой релаксации первая вершина на оптимальном пути приобретет правильный вес, потом вторая вершина ... В конце концов все вершины оптимального пути приобретут нужный вес. Самый длинный путь, который можно представить себе, состоит из $|V|$ вершин, поэтому в худшем случае нам необходимо пройти $|V|$ шагов. □

Заметим, что алгоритм Беллмана-Форда позволяет очень просто определить, существует ли в графе отрицательный цикл, достижимый из вершины end. Достаточно произвести релаксацию ровно |V| раз. Если при исполнении последней релаксации длина кратчайшего пути от какой-либо вершины строго уменьшилась, то в графе есть отрицательный цикл, достижимый из E. Отметим, что наличие отрицательного цикла делает задачу вырожденной — мы можем сколько угодно раз крутиться по этому циклу и уменьшать длину пути до $-\infty$.

Таблица 4.8: Сводка по алгоритмам поиска оптимального пути.

Алгоритм	Область применимости	Время работы
Динамическое программирование	Минимум и Максимум. Ориентированный ациклический граф. На веса нет ограничений	$O(\|V\| + \|E\|)$
Дейкстра	Минимум. Ориентированный и неориентированный граф, веса строго положительны	$O(\|E\| \cdot \log \|V\|)$
Беллман-Форд	Минимум. Ориентированный граф, веса любые, нет циклов отрицательного веса.	$O(\|V\| \times \|E\|)$

4.6.7 Сводка основных алгоритмов, ищущих оптимальный путь на графе

В таблице 4.6.7 приведена сводка по рассмотренным алгоритмам поиска оптимального пути в графе.

4.7 Сети и потоки в них

4.7.1 Основные понятия

Представим себя владельцем нефтеперерабатывающего завода. Пусть есть нефтяная вышка S, от которой нефть течет к нашему заводу по некому множеству труб (рис.4.7.1). Каждая труба обладает пропускающей способностью. Кроме труб есть узлы, в которых трубы сливаются, раздваиваются ... Допустим, по какой-то причине поменять трубы на трубы с большей пропускающей способностью мы не можем (запрещает экологическая инспекция). Тогда использовать ресурсы нашего источника максимально мы можем, только искусно распределив поток нефти по имеющимся трубам.

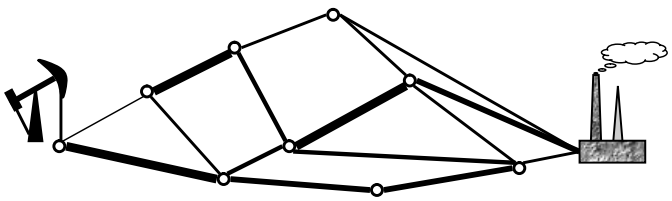


Рис. 4.13: Сеть труб

Перед нами стоит задача: распределить поток нефти по трубам с известной пропускной способностью так, чтобы была максимальная прибыль.

Задача, на самом деле далеко не простая. Если пустить слишком мало нефти по трубе — прибыль понизится, если слишком много — труба лопнет и прибыль понизится тоже.

Аналогичные задачи решаются при передаче информации через Интернет. Нужно оптимально распределить пакеты информации так, чтобы они максимально быстро и без потерь пересылались через существующие каналы, которые имеют разную пропускную способность. Условия перечисленных задач очень сходны: есть сеть, есть источник (данных, веществ и др.), есть место, куда эти данные и др. нужно доставить. Необходимо найти наилучший поток в этой сети.

Дадим формальное определение сети и потока:

Определение 3. *Сеть — это*

1. *Ориентированный граф* $G = \{V, E\}$;
2. *На каждом ребре* (uv) *определен строго положительный вес* $c(uv) > 0$*(пропускная способность).*
3. *На каждом несуществующем ребре (на каждой паре вершин, где нет ребра), определен нулевой вес*
4. *Выделены две вершины* s *— источник,* t *— сток.*
5. *Для любой вершины* v *существует путь от* s *через* v *к* t*:* $s \to v \to t$.

Условия 2, 3 означают, что для любой пары вершин определено неотрицательное число $c : V \times V \to R^+$. Это число называется *пропускной способностью ребра.* Если ребра нет, то пропускная способность равна 0.

Определение 4. *Потоком в сети называется функция, определенная на парах вершин* $f : V \times V \to R$ *и обладающая свойствами:*

1. $f(uv) \leq c(uv)$ *для любого ребра (в том числе и для несуществующего ребра) поток ограничен весом ребра — "пропускной способностью".*
2. $f(uv) = -f(vu)$*. Это свойство называется кососимметричностью.*
3. *для любой вершины* $v \in V \setminus \{s, t\}$ $\sum_{u \in V} f(uv) = 0$*, это свойство можно назвать законом сохранения — сколько в вершину "втекает столько и "вытекает". Обратим внимание на то, что закон сохранения выполняется для всех вершин. кроме источника и стока;*
4. $|f| = f(s) = \sum_(v \in V) f(s, v)$ *— суммарный поток (величина потока) равен потоку в источнике и сумме потоков, текущих из источника в каждую вершину.*

Упражнение 30. *Из этих свойств потока следует, что* $f(t) = -f(s)$*, то есть сколько вытекло из источника, столько втекло в сток. Докажите это утверждение.*

Задача о максимальном потоке. Дана сеть $G = \{V, E, c\}$ (см. определение). Найти максимальный поток, т.е. $|f| = f(s) = \sum_(v \in V) f(s, v) \to \max$.

Попробуем найти хоть один поток. Для этого найдем на нашем ориентированном графе путь P (любой) из вершины s в вершину t, проходящий по настоящим ребрам (там, где пропускная способность строго больше нуля). Это можно сделать с помощью поиска в ширину или в глубину. На этом пути можно найти ребро с минимальной пропускной способностью $c_{min} = \min_{e \in P} c(e)$. Теперь определим поток:

$$f(uv) = \begin{cases} c_{\min} & uv \in P \\ -c_{\min} & vu \in P \\ 0 & uv \notin P \end{cases} \quad \text{где } u \text{ и } v \text{ — смежные вершины}$$

Теперь, наверное, можно поток улучшить. Метод поиска максимального потока, основанный на последовательном увеличении потока (если оно возможно) был придуман Фордом и Фалкерсоном и носит их имя.

4.7.2 Метод Форда-Фалкерсона

Пусть у нас дан поток f. Определим остаточную пропускную способность следующим образом $c_f(uv) = c(uv) - f(uv)$. Поскольку по определению потока он не может превышать пропускную способность, остаточная пропускная способность неотрицательна. Кроме того остаточная пропускная способность может превратить некоторые виртуальные ребра (для которых пропускная способность была равна 0), в реальные ребра с положительной пропускной способностью. Например, если есть ребро (uv), нет ребра (vu) и $f(uv) = f_{uv} > 0$. Тогда по определению потока $f(vu) = -f(uv)$. Стало быть, остаточная пропускная способность на ребре (vu) определится как $c_f(vu) = c(vu) - f(vu) = 0 - (-f_{uv}) = f_{uv} > 0$, т.е. станет положительной. Смысл возникновения таких несуществующих ребер заключается в следующем. Допустим, мы пропустили по ребру слишком большой поток, а в оптимальном потоке эта величина должна быть меньше. Тогда такие дополнительные ребра позволяют нам исправить ошибки, пропуская поток в обратном направлении, что эквивалентно сокращению потока в прямом направлении.

Определение 5. *Остаточной сетью называется сеть, в которой ребра есть между теми вершинами, где остаточная пропускная способность положительна* $c_f(uv) > 0$*:*

$$c_f = \{V, E_f = \{(uv) \in V \times V : c_f(uv) > 0\}, c_f\}$$

В основе метода Форда-Фалкерсона лежит следующая

Лемма 1. *Лемма. Пусть* G *сеть,* f *— поток* G' *— остаточная сеть и* f' *— поток в остаточной сети. Тогда* $f + f'$ *является потоком в сети* G *и его величина равна* $|f + f'| = |f| + |f'|$.

Доказательство. Для доказательства надо проверить, что для $f + f'$ выполняются свойства потока:

- Кососимметричность. Поскольку $f(vu) = -f(uv)$ и $f'(vu) = -f'(uv)$, то $f(vu) + f'(vu) = -f(uv) + f'(uv)$
- Закон сохранения. Поскольку $\sum_v f(uv) = 0$ и $\sum_v f'(uv) = 0$, то $\sum_v (f(uv) + f'(uv)) = 0$
- Ограничение на пропускную способность. Действительно $f'(uv) < c_f(uv)$. Поскольку по определению $c_f(vu) = c(vu) - f(vu)$, то $f'(uv) \leq c(vu) - f(vu)$, или $f(uv) + f'(uv) \leq c(uv)$
- Величина потока $|f + f'| = \sum_v (f(sv) + f'(sv)) = |f| + |f'|$

□

Если остаточная сеть имеет путь из источника s в сток t, то по сети можно пропустить дополнительный поток так же, как мы это делали в первый раз. Процедуру можно повторять до тех пор, пока в остаточной сети не останется путей из источника в сток. На каждом шаге такой процедуры мы увеличиваем поток. Всего возможных способов увеличения потока конечно. Поэтому процедура сходится за конечное число шагов. Правда число шагов может оказаться очень большим. В таблице 4.7.2 показан пример работы алгоритма.

Вообще говоря, из того, что в дополнительной сети нет пути из источника в сток, не следует, что мы нашли максимальный поток. Может быть, если бы мы выбирали другие пути из s в t для пропуска дополнительного потока, мы смогли достичь в конце концов большего потока? Есть теорема, доказывающая, что этот алгоритм дает в конце максимальный поток. Она будет доказана позже.

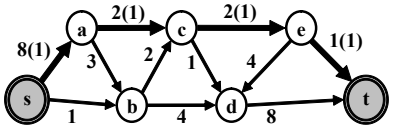
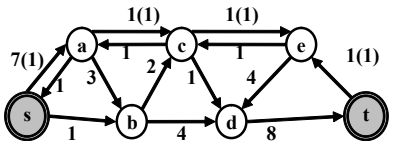
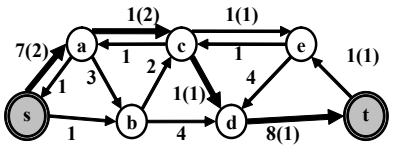
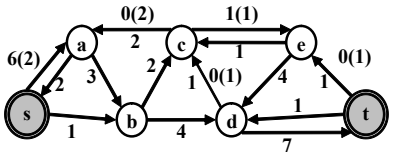
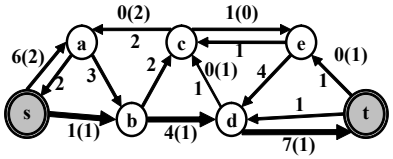
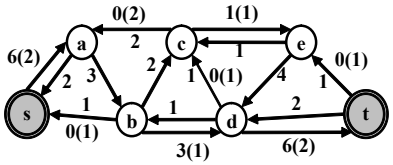
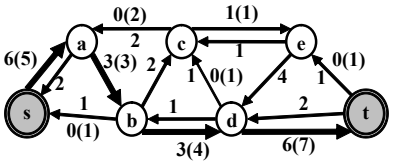
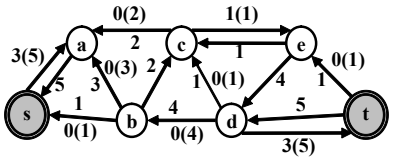
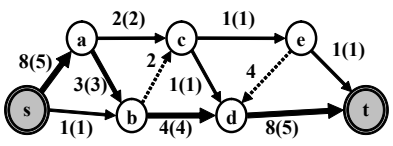
4.7.3 Разрезы в сетях

Определение 6. *Разрезом сети $G = \{V, E, c\}$ называется разбиение множества вершин на два подмножества S и T, таких, что:*

1. $V = S \cup T$
2. $S \cap T = \emptyset$
3. $s \in S, \quad t \in T$

Говорят, что ребро $e = (vu)$ принадлежит разрезу, если вершина u принадлежит подмножеству S, а вершина v — подмножеству T. Просто говоря, ребро принадлежит разрезу, если мы его "разрезали разделив вершины между S и T. Принадлежность ребра e к разрезу обозначается $\in E_c$. Отметим, что ребра, идущие из T в S разрезу не принадлежат (рис.4.14).

Таблица 4.9: Поиск максимального потока (числа в скобках — величина потока).

сеть	поток	пояснения
	1	Находим путь из источника в сток (выделено)
	1	Строим дополнительную сеть. ребро et исчезает.
	2	Находим путь по дополнительной сети
	2	исчезает ребро cd
	3	Находим путь
	3	Строим остаточную сеть исчезло ребро sb
	6	Поток увеличился на 3
		В дополнительной сети нет пути из s в t. Поток нельзя увеличить.
		Результирующий поток

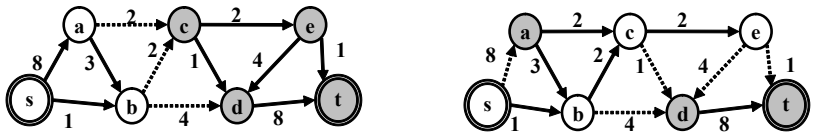


Рис. 4.14: Примеры разрезов в сетях. Белым показаны вершины, принадлежащие множеству S, серым — множество T. Пунктиром показаны ребра, принадлежащие разрезу.

Определение 7. *Пропускной способностью разреза называется сумма пропускных способностей всех ребер разреза:*$C(S,T) = \sum_{u \in S, v \in T} c(uv)$. *Разрез минимальной пропускной способности называют минимальным разрезом сети. Поток из S в T равен:* $f(S,T) = \sum_{u \in S, v \in T} f(uv)$. *Суммирование ведется по всем вершинам u, принадлежащим S и по всем вершинам v, принадлежащим T. Максимальный поток через разрез равен пропускной способности разреза.*

Свойства потока между множествами вершин сети $G = \{V, E, c\}$: 1. $f(X,X) = 0$; 2.f(X,Y)=-f(Y,X); 3. Пусть $X, Y, Z \subseteq V$ и $X \cap Y = \emptyset$, тогда $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$; $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$, т.е. поток дистрибутивен. 4. Пусть $X \subseteq V \setminus \{s,t\}$ (в подмножество X могут входить любые вершины сети, кроме источника и стока). Тогда $f(X,V) = 0$. Поток из любого множества вершин, не включающего s и t, в множество вершин, равен нулю.

Упражнение 31. *Докажите свойства 1-4.*

Лемма 2. *Имеется некая сеть $G = \{V, E, c\}$, некий ее разрез $S \cup T$ и некий поток через эту сеть f. Тогда для любого разреза выполняется $f(S,T) = |f|$, т.е. поток через любой разрез равен потоку между источником и стоком.*

Доказательство. Очевидно, что любой поток обязательно проходит через ребра разреза (поскольку любая вершина из V принадлежит либо S, либо T). Более строгое доказательство выглядит так:

$$f(S,T) = f(S,S) + f(S,T) = f(S, S \cup t) = f(S,V) = f(S \setminus s) + f(s,V) = |f|$$

Здесь использовано свойство 4: множество $S \setminus s$ не содержит ни источника, ни стока, поэтому $f(S \setminus s) = 0$. □

Лемма 3. *для любого разреза $S \cup T = V$ и любого потока f выполнено $|f| \leq c(ST)$.*

Доказательство. следует непосредственно из первого свойства потока и леммы 1. □

4.7.4 Теорема о максимальном потоке и минимальном разрезе

Теорема 3 (Фолкерсона)**.** *Следующие утверждения эквивалентны.*

1. *Поток f — максимальный*
2. *Остаточная сеть не содержит путей из s в t*
3. *Существует разрез $S \cup T = V$, такой, что $|f| = c(S,T)$, в частности, это значит, что если вершины $u \in S$, $v \in T$ то $f(uv) = c(uv)$, то есть существует такой разрез, поток через который равен пропускной способности разреза. Иными словами, существует хотя бы один разрез сети, поток через который максимален и равен пропускной способности разреза.*

Доказательство. **1** $\to$ **3** очевидно. От противного: если бы остались пути из s в t, поток можно было бы еще увеличить.

2 $\to$ **3** Пусть в остаточной сети нет путей из s в t. Рассмотрим в качестве S' множество вершин, достижимых из источника s по остаточной сети, в качестве T' — остальные вершины. Мы получили разрез, поскольку $S' \cap T' = \emptyset, S' \cup T' = V, s \in S', t \in T'$. По построению ни одно ребро остаточной сети не принадлежит разрезу (мы собрали в множестве все ребра, достижимые по остаточной сети из источника). Поэтому остаточный вес ребер разреза равен нулю. Формула для получения остаточного веса ребра (uv) $c_f(uv) = c(uv) - f(uv)$, следовательно, для ребер, принадлежащих разрезу $c(uv) = f(uv)$, и поток через разрез равен пропускной способности разреза, что и требовалось доказать.

3 $\to$ **1** Для любого разреза $S \cup T = V$ выполнено $|f| \leq c(S,T)$, поэтому из равенства $|f| = c(S,T)$ следует максимальность потока.

□

4.7.5 Случай многих источников и стоков

Разумеется, Вы владеете не одной качалкой для нефти и заводов у Вас несколько. Как найти оптимальное распределение нефти по трубопроводам? Очень просто. Надо ввести дополнительные вершины — общий источник и общий сток и провести ребра из универсального источника в Ваши качалки и от Ваших заводов к универсальному стоку. Если Ваше месторождение имеет ограничение по дебету (мощности), то эти дополнительные ребра должны иметь пропускную способность, равную дебету скважин. Аналогично для ограничений мощности заводов (рис.4.15).

4.7.6 Паросочетания в двудольном графе

Двудольный граф позволяет разбить множество вершин на два подмножества, так, что ребер внутри этих подмножеств нет. Паросочетания — множество ребер в графе, такое, что нет ребер, имеющих общую вершину.

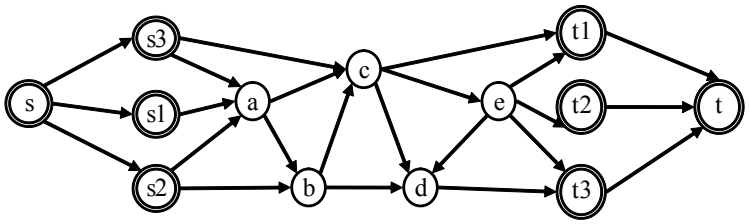


Рис. 4.15: Случай многих источников и стоков.

Добавляем к двудольному графу источник и сток, пропускные способности устанавливаем на всех ребрах графа =1. Ищем максимальный целочисленный поток. Ребра, которые будут иметь поток, равный 1 обеспечат максимальное паросочетание. Таким способом может решиться задача о браке: есть n юношей и n девушек, нужно составить максимальное число супружеских пар. Пусть юноши составляют одну "долю"графа, девушки другую (все они являются вершинами графа). Между двумя вершинами есть ребро, если между соответствующими юношей и девушкой есть симпатия. Найдя максимальное паросочетание в таком графе, мы решим задачу о максимальном количестве браков. В биоинформатике поиск потока на двудольном графе — сети с вершинами — последовательностями используется можно использовать для поиска ортологов. Вершины графа одной доли — гены в одном геноме, другая доля графа соответствует второму геному. Задача о паросочетаниях, а также другие задачи, связанные с потоками встречаются в качестве вспомогательных задач весьма часто, в том числе и в биоинформатике.

Глава 5

Свойства задач и NP-полные задачи.

5.1 Нижняя оценка времени сортировки массива

До сих пор мы рассматривали разные задачи и для них строили алгоритмы, которые их решают. При этом мы делали оценку времени работы алгоритма. Например, когда мы рассматривали задачу сортировки массива, мы сначала построили наивный алгоритм, который сортировал массив за время $T = O(N^2)$. Потом мы построили алгоритм, который выполнял сортировку за время $T = O(N \log N)$. Может быть если еще подумать, то можно построить алгоритм сортировки за время, скажем, $T = O(N)$, или даже $T = O(\log N)$?

Итак, задача сортировки массива. Сортировка массива предполагает, что элементы массива можно сравнивать, т.е. для любых двух элементов массива можно сказать верно одно из следующих утверждений $\forall a, b$ $a > b$ или $b > a$ или $a = b$. При этом выполняется условие транзитивности: если $a > b$ и $b > c$, то $a > c$. Аналогично для равенства. Задача: переставить элементы массива так, чтобы $\forall 0 < i < N$ выполнялось $a[i] > a[i-1]$.

Теорема 1. *Не существует алгоритма, который решал бы задачу сортировки массива быстрее, чем за $O(N \log N)$, иначе говоря, быстрее, чем за $c \cdot N \log N$, где c — некая константа.*

Доказательство. Любому алгоритму сортировки соответствует дерево решений: надо сделать сравнение и в зависимости от результата что-то сделать (например, переставить два элемента) и затем сделать следующее сравнение. В результате алгоритму соответствует дерево решений. Листья этого дерева соответствуют отсортированному массиву. Если зафиксировать длину массива, то есть взаимно-однозначное соответствие дерева решений и

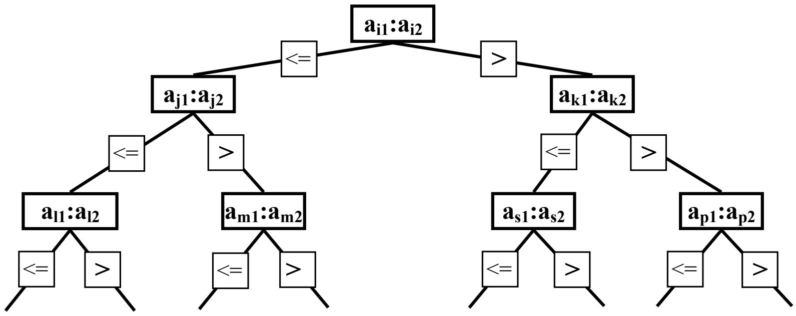


Рис. 5.1: Дерево решений в задаче сортировки. Узлы — сравнения. В зависимости от результата сравнения происходят какие-то действия, определяемые алгоритмом и происходят следующие сравнения.

алгоритма (если не учитывать такие "мелочи как действия, связанные с перестановкой элементов). Количество листьев дерева решений равно числу возможных перестановок n элементов. Действительно, каждому отсортированному массиву соответствует набор индексов исходного массива в том порядке, в каком элементы будут лежать в отсортированном массиве.

Время работы алгоритма сортировки равно высоте h дерева решений. Максимальное число листьев в дереве высоты h равно 2^h. Число листьев в дереве решений равно $n!$, поскольку каждому листу соответствует одна перестановка элементов массива. Поэтому для любого алгоритма должно выполняться

$$N! \leq 2^h \text{ или } h \geq \log_2 N$$

Применяя оценку для факториала $N! \geq (N/e))^N$, получаем окончательно:

$$T(N) = c \cdot h \geq c \cdot N \log N - N \cdot \log e = O(N \cdot \log N)$$

□

Таким образом, $O(N \cdot \log N)$ — нижняя оценка времени работы любого алгоритма сортировки массива. Отметим, что эта оценка является оценкой для наихудшего случая. Не исключено, что дерево решений может иметь достаточно короткие пути, и для лучшего случая время работы алгоритма будет меньше. На самом деле есть алгоритмы сортировки, которые это делают за линейное время (!). Это алгоритмы целочисленной сортировки, которые учитывают не только результат сравнения $a > b$, но и то насколько одни элемент больше другого. Это позволяет сделать ветвление в дереве решений не на две ветки, а на количество ветвей, сравнимое с длиной массива. Для целых чисел такое сравнение возможно, но, например для строк, это сделать затруднительно, а для многих других случаев просто невозможно. Из этой теоремы следует важный вывод. Оказывается можно до создания алгоритма оценить минимально достижимое время работы, и это время является *свойством задачи*, а не алгоритма, которого мы не знаем.

5.2 Формальные языки. Допускающий и распознающий алгоритмы

Задачи бывают разные. В том числе, бывают и те, в которых ответ надо дать в виде "да"или "нет". Для решения задач такого плана (а их еще иногда называют задачами разрешения) удобно использовать терминологию теории формальных языков. Понятие "формальный язык"определяет множество конечных слов (строк) над конечным алфавитом. Вход любого алгоритма сводится к строке (любая программа имеет на входе двоичный файл). Поэтому время работы алгоритма, вообще говоря, правильно оценивать как функцию длины входной строки. Итак:

Алфавит — некое (конечное) множество Σ.

Слово длины n — последовательность из n символов алфавита Σ.

Σ^* — Множество всех слов всех длин

Язык L **над алфавитом** Σ — подмножество Σ^*: $L \subseteq \Sigma^*$ — (некоторое множество строк символов из алфавита).

Поскольку алфавит конечен, то есть отображение множества всех слов в алфавите $\Sigma = \{a_i\}$ на множество всех слов в алфавите $\Sigma_{0,1} = \{0, 1\}$. Далее будем рассматривать языки над алфавитом $\{0, 1\}$. Так, например, можно рассматривать алфавит $\Sigma_{0,1} = \{0, 1\}$ и язык $L = \{10,\ 11,\ 101,\ 111,\ 1011,\ 1101,\ 10001\cdots\}$, состоящий из двоичных записей простых чисел.

Мы будем рассматривать класс алгоритмов, результатом которых является ответ либо "да"либо "нет". Т.е. алгоритмы, которые реализует отображение $\{0, 1\}^* \to \{0, 1\}$. Эти алгоритмы называются решающими алгоритмами. Таким образом, алгоритм A определяет подмножество $L \subseteq \Sigma^*$, такое, что $A(x \in L) = 1, A(x \notin L) = 0$. Иными словами, алгоритм определяет некоторый язык. Теперь становится интуитивно ясным, что есть соответствие между формальными языками и задачами (ведь алгоритм решает задачу). Разумеется, класс решающих алгоритмов гораздо уже, чем все мыслимые алгоритмы. В частности, алгоритм сортировки затруднительно представить в виде решающего алгоритма. Однако, алгоритмы для задач оптимизации (например, задача поиска кратчайшего пути в графе) можно представить как решающие алгоритмы. Соответствующий решающий алгоритм будет отвечать на вопрос "имеет ли данный граф путь, короче, чем заданная величина?".

Введем несколько важных определений.

Определение 1. *Алгоритм A допускает слово x, если $A(x) = 1$.*

Определение 2. *Алгоритм A отвергает слово x, если $A(x) = 1$.*

Заметим, что алгоритм может не допускать и не отвергать некоторые слова. Случается и так, что алгоритм не дает никакого ответа (например, зацикливается или аварийно завершается).

Определение 3. *Алгоритм A допускает язык L, если он допускает те и только те слова, которые принадлежат L.*

Понимать следует и то, что алгоритм A, допускающий некоторый язык L, вовсе не обязан отвергать всякое слово, не входящее в L. Пусть, например, язык L это все Эйлеровы графы. Ясно, что граф можно описать в виде двоичного кода. Однако есть (в зависимости от двоичного представления) двоичные слова, которые не представляют никакого графа. В случае, если алгоритму представлено недопустимое слово, алгоритм может аварийно завершить работу, т.е. допускающий алгоритм для языка L может не дать ответа.

Определение 4. *Алгоритм A распознает язык L, если он допускает все слова из L и отвергает все другие слова.*

Определение 5. *Язык L допускается за полиномиальное время, если существует допускающий алгоритм A, причем всякое слово x допускается алгоритмом за время порядка $O(n^k)$ (т.е. время работы алгоритма A порядка $O(n^k)$), где n — длина слова, а k — некоторое не зависящее от x число.*

Определение 6. *Язык распознается за полиномиальное время, если существует распознающий алгоритм A и число k такие, что его время работы порядка $O(n^k)$, где n — длина слова.*

Определение 7. *Класс языков, для которых существуют полиномиальные алгоритмы называется классом P.*

5.3 Проверяющие алгоритмы. Класс языков NP

Определение 8. *Назовем* проверяющим алгоритмом *A алгоритм с двумя аргументами. Первый аргумент будем называть* входной строкой *(слово $x \in L$), а второй —* сертификатом*. Проверяющий алгоритм допускает слово x, если существует сертификат $y \in \{0,1\}^*$, такой, что $A(x,y) = 1$ для $x \in L$ и не существует сертификата для других слов. Отметим, что сертификат может не иметь никакого отношения к языку L.*

Приведем пример. Гамильтоновым циклом в графе является такой цикл, который проходит через все вершины графа ровно по одному разу. Графы, имеющие Гамильтонов цикл называются Гамильтоновыми графами. Пусть язык Ham — множество Гамильтоновых графов. Мы не умеем проверить непосредственно, является ли предъявленный граф гамильтоновым, т.е. у нас нет распознающего алгоритма, такого, что $A(x) = 1$, если слово x принадлежит языку Ham и $A(x) = 0$ в противном случае. Однако, если кто-то нам предъявит последовательность вершин и заявит, что эта последовательность является Гамильтоновым циклом, мы можем это проверить, т.е. можем проверить, что предъявленное двоичное слово действительно представляет последовательность вершин исследуемого графа, что

эта последовательность является циклом и что она содержит все вершины графа. Эта последовательность может использоваться в качестве сертификата. Отметим, что последовательность вершин данного графа не является вообще каким-либо графом. Тем не менее, эта последовательность вершин представляется в виде двоичного слова, т.е. $y \in \{0,1\}^*$. Алгоритм, который проверяет, действительно ли y представляет гамильтонов цикл в графе x довольно легко реализовать. Ясно, что этот алгоритм должен зависеть от двух аргументов — собственно слова x, представляющего граф, и слова y, представляющего последовательность вершин — $A(x,y)$.

Определение 9. *Язык принадлежат классу NP, если существует проверяющий алгоритм $A(x,y)$, причем:*

- *Существуют такие константы C_p и p_2 , что длина сертификата ограничена полиномом: $|y| \leq C_p \cdot |x|^{p_1}$;*
- *Время работы проверяющего алгоритма ограничено полиномом от длины входного слова: $T(A(x,y)) \leq C_A \cdot |x|^{p_1+p_2}$, где C_A, p_1, p_2 — константы, не зависящие от входного слова.*

NP расшифровывается как "полиномиально неопределенный".

Язык (или задача) тогда принадлежит классу NP, когда он легко проверяем проверяющим алгоритмом. Заметим, что мы не говорим, что умеем решать такую задачу — умеем быстро проверять, является ли представленное решение правильным. Отметим, что мы здесь нигде не конкретизируем, а как задачу представить в виде двоичных слов. Для записи тех же графов можно придумать разнообразные способы записи, т.е. разнообразные отображения *ЗАДАЧА*$\rightarrow \{0,1\}^*$. Однако можно считать, что все мыслимые отображения в некотором смысле эквивалентны, т.е. длины слов в разных представлениях одной и той же задачи отличаются друг от друга не более, чем полиномиально.

Упражнение 32. *Докажите, что все языки, для которых существует полиномиальный распознающий алгоритм, принадлежат классу NP.*

5.4 Сводимость языков. NP-полные задачи (NPC)

Говоря о сводимости задач вспоминается старый анекдот. **Задача1.** Дано: чайник, кран, газовая плита и спички. Требуется вскипятить воду. Решение: наливаем в чайник воду, зажигаем газ, ставим чайник на газ, ждем 10 минут. **Задача 2.** Дано чайник с водой, кран, газовая плита и спички. Требуется вскипятить воду. Решение. Выливаем воду из чайника и сводим задачу к предыдущей. В этом анекдоте заложена идея сводимости одних задач к другим.

Определение 10. *Язык L_1 сводится к языку L_2, если существует отображение $f : \{0,1\}_* \rightarrow \{0,1\}_*$ такое, что $x \in L_1 \Leftrightarrow f(x) \in L_2$. Отметим,*

что отображение f не является взаимно-однозначным, т.е. может не существовать обратного отображения (рис.5.2). Частным случаем такого отображения является допускающий алгоритм (если существует). В этом случае язык L_2 состоит только из одного слова $\{1\}$, а отображение есть алгоритм: $f(x) = A(x)$. Если сводящее отображение $f(x)$ вычисляется за полиномиальное время от длины слова x, то говорят, что язык L_1 сводится к языку L_2 за полиномиальное время, и обозначается $L_1 <_p L_2$.

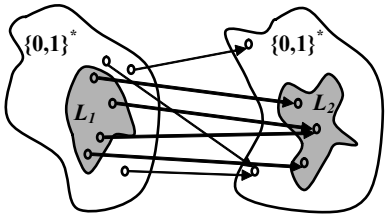


Рис. 5.2: К понятию сводимости.

Определение 11. *Язык L принадлежит классу NP-полных (NP-complete, NPC), если:*

1. *$L \in NP$ — язык принадлежит класс NP (полиномиально проверяемые задачи)*
2. *$L' <_p L$ для любого языка $L' \in NP$, любой такой язык сводим к L,*

Иными словами, NP-полные задачи это полиномиально проверяемые задачи, которые полиномиально не менее сложные, чем любая NP-задача.

Благодаря второму свойству (свойству сводимости) существует ОДНА И ТОЛЬКО ОДНА NP-полная задача. Если для хотя бы одной NP-полной задачи найдется полиномиальный алгоритм, то ВСЕ NP-полные задачи можно будет решить за полиномиальное время.

Есть гипотеза: для NP-полных задач не существует полиномиального алгоритма.

Некоторое количество NP-полных задач надо знать наизусть. Это полезно, поскольку если вы сидите и решаете какую-то задачу, стараясь найти полиномиальный алгоритм, а к этой задаче сводится NP-полная задача, не надо биться — полиномиального алгоритма решения нет. И если Вы смогли построить полиномиальный алгоритм, то, скорее всего, Вы придумали неправильный алгоритм, который, по крайней мере, в некоторых случаях не решает Вашу задачу. Надо понимать, что для NP-полных задач нет эффективных полиномиальных алгоритмов, а есть, по-видимому, экспоненциальные. Но если Ваша практическая задача небольшая (например граф имеет немного — порядка 10 вершин), то вполне может устроить и неэффективный алгоритм полного перебора. Впрочем даже для достаточно большого

размера задачи есть выход — искать приближенное решение, или, опираясь на особенности задачи решать не универсальную задачу, а суженную задачу. Эта ситуация будет рассмотрена позже.

Впрочем, если использовать вычислительные средства, в которых количество процессоров *сравнимо* с размером задачи, то можно строить эффективные алгоритмы решения таких задач. Кроме того, квантовые вычислители также могут решать сложные задачи (правда, в некотором *вероятностном* смысле). Впрочем рассмотрение этих вычислителей выходит далеко за рамки курса.

5.5 Некоторые NP-полные задачи

Поиск максимальной клики в графе. Поиск максимальной клики в графе — это поиск максимального полного подграфа (где все ребра связаны со всеми). Поиск любой клики тривиален — кликой мощности 2 является любое ребро. Задача поиска клики в графе часто встречается, в том числе и в биоинформатике. К примеру, программа CluD, используемая для поиска гидрофобных ядер в белках, основана на поиске клики в специальном графе "гидрофобных групп"белка, или поиск ортологов — по большому счету это поиск клик в многодольных графах. Мы уже упоминали, что задача на максимум заменяется задачей о том есть ли в графе клика больше заданного размера. При формализации прикладной задачи (например,из области биоинформатики) надо иметь в виду, что часто максимальная клика не единственна!

Поиск гамильтонова пути. Поиск гамильтонова пути — в отличие от поиска эйлерова пути так просто не решается. Выше мы уже говорили (см. раздел про применение эйлеровости графа в биоинформатике) о возможном применении поиска гамильтонова пути и замене его поиском эйлерова пути на другом графе.

Задача коммивояжера. Задача коммивояжера основана на предыдущей задаче о гамильтоновом пути. Это интересная прикладная задача. Коммивояжер (бродячий торговец) ездит по городам и предлагает свои товары. Понятно, что чем больше городов он объедет, тем больше денег заработает, при этом мы учитываем стоимость переезда из одного города (вершины графа) в другой — не каждая дорога (ребро графа) коммивояжеру выгодна. Сложность задачи еще и в том, что коммивояжер нещадно обманывает и обсчитывает своих покупателей, поэтому появляться дважды в одном городе ему нельзя. Наша задача — помочь наглому коммивояжеру заработать как можно больше денег.

Незацикливающаяся программа. Незацикливающаяся программа — задача о том, как, смотря на любую программу, сказать, будет ли он за-

цикливаться, или нет (такую задачу каждый из вас "решал вручную"на занятиях по программированию).

Задача о рюкзаке. Задача о рюкзаке знакома каждому туристу, путешественнику и студенту ФББ (как завсегдатаю ЗБС и ББС). Есть большой рюкзак и много вещей разного заданного размера. Нужно положить в рюкзак максимально количество вещей. Заметим, что жадный алгоритм — хватать самое большое и запихивать в рюкзак — здесь не подходит. Эта задача (несколько модифицированная) имеет приложения в биоинформатике. Речь идет о расшифровке аминокислотных последовательностей по данным масс-спектрометрии. В весьма упрощенной постановке задача выглядит следующим образом. Есть белок с неизвестной последовательностью. Его случайным образом порезали на фрагменты. Обычно это делают с использованием различных протеаз. Далее, с помощью масс-спектрометра определяют массы фрагментов. Мы знаем массы аминокислотных остатков. Вопрос: из каких остатков могут состоять фрагменты. Дальнейшая, весьма сложная работа заключается в восстановлении аминокислотной последовательности. К сожалению, этот подход непосредственно применить очень трудно еще из-за того, что есть неоднозначность в определении массы фрагментов (некоторые фрагменты могут при себе нести дополнительные химические группы, например, OH и поэтому эти модификации дают дополнительные пики в спектре).

Задача о выполнимости булевой формулы (SAT от satisfaction). Задача о выполнении булевой формулы. Имеется некая (любая) булева формула от любого конечного числа булевых переменных, к примеру, такая: . Задача заключается в том, чтобы убедиться, что существует набор переменных, при которых эта формула принимает значение истина.

Для приведенных задач очевидно, что существуют быстрые проверочные алгоритмы — их очень просто написать. Для того, чтобы доказать, что какая-либо задача принадлежит классу NP-полных задач необходимо, во-первых, показать существование полиномиального проверяющего алгоритма с сертификатом полиномимальной длины. А во вторых, свести к этой задаче какую-нибудь NP-полную задачу. Обратим внимание, на то что к чему надо сводить. Чтобы доказать NP-полноту задачи L надо *к ней* свести известную NP-полную задачу, а не наоборот.

Для задачи SAT доказана NP-полнота полностью, т.е. доказано, что любая задача из класса NP сводится к ней. Эта теорема достаточно сложна и мы не будем ее здесь доказывать. Однако мы покажем, как производится сводимость одних NP-полных задач к другим.

5.5.1 Задачи выполнимости SAT и 3CNF.

3-Conjunctive Normal Form или 3-конъюктивная нормальная форма, сокращенно 3CNF — это комбинация из троек переменных, внутри каждой трой-

ки между переменными стоит ИЛИ, а перед некоторыми переменными стоит отрицание ¬, а между тройками стоит И. Пример 3CNF:

$$(x_1|x_2|\neg x_3)\&(x_1|\neg x_2|x_3).$$

Комбинацию вида x или $\neg x$ называется термом, Совокупность трех термов, разделенных знаком |, называется группой. Следовательно, 3CNF — это конечное число групп, разделенных знаком &. 3CNF сводима к SAT. Это очевидно, поскольку 3CNF является частным случаем SAT, и если у нас есть алгоритм решения задачи SAT, то автоматически мы умеем решать задачу 3CNF. Но для доказательства NP-полноты 3CNF надо наоборот свести задачу SAT к 3CNF!

Докажем, что SAT сводится к 3CNF. Для этого покажем, что для любой SAT можно написать эквивалентную 3CNF, быть может, для этого нам придется ввести дополнительные переменные. Рассмотрим пример. Пусть нам дана булева формула:

$$\varphi = ((x_1\&(x2|x3))\&(x_2|x_4))|x_1 \tag{5.1}$$

Построим для этого выражения так называемое дерево разбора. Чтобы вычислить это выражение нам надо вычислить выражения в скобках, и сделать соответствующие подстановки. Введем дополнительную переменную $y_1 = x_2|x_3$. Тогда мы сможем вычислить следующую скобку $y_2 = x_1\&y_1$. Продолжая вводим еще переменные $y_3 = x_2|x_4)$, $y_4 = y_2\&y_3$, $y_5 = y_4|x_1$. Эту процедуру можно представить в виде дерева разбора (см. рис.5.3). Листья дерева — переменные. Узлы — бинарные операции. Теперь (5.1) можно

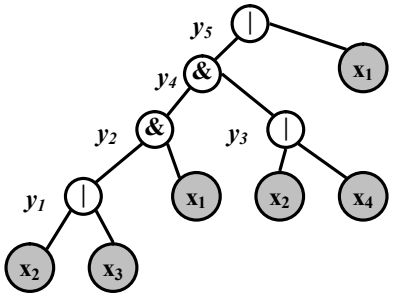


Рис. 5.3: Дерево разбора для (5.1).

переписать в виде:

$$\varphi = y_5\&(y_5 \equiv y4|x_1)\&(y_4 \equiv y_2\&y_3)\&(y3 \equiv x_2|x_4)\&(y_2 \equiv y_1\&x_1)\&(y_1 \equiv x_2|x_3)$$

Отметим, что, во-первых, эта формула эквивалентна исходной формуле. Во-вторых, она почти 3CNF , поскольку она является комбинацией из групп по 3 терма. Единственное, вместо необходимых операций "или"там встречаются разные другие операции — проверка тождественности, операции "и".

Кроме того, есть группа, состоящая из одного терма — y_5. Самое простое — представить эту группу в виде тройки: $(y_5|y_5|y_5)$, которая вполне является 3-группой.

Упражнение 33. *Докажите, что приведенная формула эквивалентна исходной, т.е. она принимает значение true на тех и только на тех наборах переменных x, на которых принимает значение true исходная формула.*

Приведем полученные тройки к 3CNF. Для каждой тройки вида $x \equiv y|z$ либо $x \equiv y\&z$ существует всего 8 разных наборов булевых переменных, обращающих тройку в "true"либо "false". К примеру, если все три переменные равны нулю, тройка обращается в равенство (0=0&0). Это равенство верное, ему соответствует "true"(1).

Перечислим все возможные тройки булевых переменных и соответствующие им элементы решения тройки $(x \equiv y\&z)$:

тройка	Значение
0 0 0	1
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	0
1 0 1	0
1 1 0	0
1 1 1	1

Рассмотрим сочетание (0, 1, 1). Попробуем представить это значение в виде комбинации термов, объединенных операцией "так, чтобы оно было равно 0 при значениях переменных (0, 1, 1) и только при них. Эта комбинация будет такой: $(x \equiv y\&z) = (x|\neg y|\neg z)$. Действительно, чтобы получить 0 нам надо, чтобы все термы были раны 0. Ясно, что там, где в исходных данных была 1 нам надо применить отрицание, а там, где был 0 надо принять переменную как есть. Заметим, что тройка $(x|\neg y|\neg z)$ является группой 3CNF. Аналогично для любой тройки, результатом вычисления которой есть 0 мы определяем свой терм. Это процедура называется приведением термов.

Для тех, комбинаций, которые в результате дают 1 приведение термов не возможно. Но это не важно. Просто мы в финальную 3CNF не будем писать соответствующие группы. Эти группы равны 1, а операция &, которая объединяет группу, нейтральна по отношению к этому значению.

В результате мы каждую комбинацию вида $x \equiv y \diamond z$ заменяем некоторым количеством групп (не более 8), объединенных операцией "&". А эти комбинации, в свою очередь, также объединены операцией "&". Итак, мы получили 3CNF, причем ее длина не более, чем в 16 раз больше, чем длина исходной формулы.

Алгоритм приведения произвольной SAT к 3CNF:

1. Строим дерево разбора

2. Расширяем термы (вводим новые переменные)

3. Приводим термы и строим 3CNF

Итак, мы показали (не доказали), как свести SAT к 3CNF. Если рассуждения, приведенные в примере обобщить, то будет доказано, что задача SAT не сложнее задачи 3CNF. Но задача SAT является NP-полной, поэтому и задача 3CNF также является NP-полной. Обратите внимание на то, в какую сторону мы делали сводимость. Мы сводили NP-полную задачу к нашей (не наоборот!).

5.5.2 Задача о клике (CLIQUE).

Дан граф $G = \{V, E\}$. Выяснить есть ли в нем клика, содержащая более k вершин. Проверка решения задачи тривиальна (достаточно предъявить список из вершин), поэтому это задача класса NP. Оценим ее полноту. Для этого сведем к этой задаче другую NP-полную задачу. Покажем, что 3CNF-SAT $<_p$ CLIQUE.

Пусть дана некая 3CNF:

$$(x_1|x_2|\neg x_3)\&(x_1|\neg x_2|x_4)\&(x_2|x_5|\neg x_3)\&(\neg x_3|x_4|\neg x_5)\&(x_3|\neg x_3|x_3)$$

Построим по ней многодольный граф, причем такой, что если в нем есть k-клика, то эта формула выполнима. Для каждой группы рисуем 3 вершины (по одной на терм). Вершины u и v соединяем ребром, если они принадлежат разным тройкам и совместимы. Несовместимыми считаются вершины, несущие — одна некий терм, другая — его отрицание. К примеру, несовместимы вершины, несущие термы x_3 и $\neg x_3$. Граф получается очень запутанный (на рис.5.4 показаны не все ребра).

Лемма 1. *Если в графе, построенном по некой 3CNF по указанному правилу, есть клика размером в число групп 3CNF, эта 3CNF разрешима.*

Доказательство. Если формула разрешима, то существует набор переменных, когда все группы истинны. Тогда в каждой группе есть истинный терм, и, следовательно, группа тоже истинна. Соответствующие вершины образуют клику. Обратное тоже верно — если есть клика размера k, то формула разрешима. □

В нашем случае для разрешимости 3CNF нужно требовать существования клики мощностью 5. Такая клика в графе есть (выделено жирным). Для получения набора переменных, удовлетворяющего 3CNF, взяли термы, которые образуют клику в графе, и присвоили им значение 1("true"), значения остальных переменных не важно. При существовании показанной клики наша 3CNF разрешима с таким набором переменных: $x_1 = true, x_2 = true, x_3 = false$, так как $\neg x_3 = true$, x_4 — любое $x_5 = false$.

Итак, NP-полная задача 3CNF сводится к задаче о клике в многодольном графе. Поэтому задача о клике в многодольном графе также является

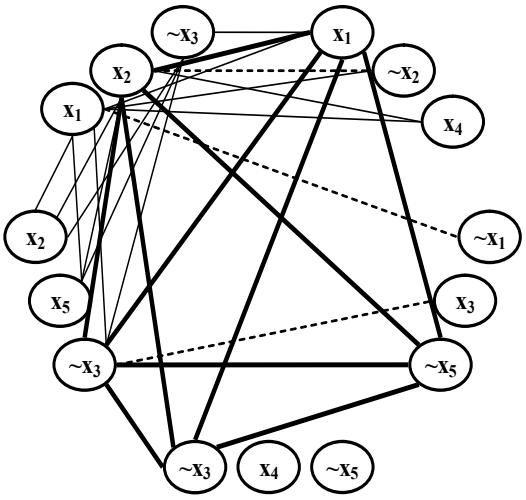


Рис. 5.4: Многодольный граф, соответствующий 3CNF. *Показаны не все ребра.* Пунктиром отмечены пары несовместимых вершин. Жирным выделены ребра, принадлежащие клике.

NP-полной. С другой стороны, задача о клике в многодольном графе является частным случаем (следовательно, тривиально сводится) к задаче о клике в произвольном графе. Следовательно, задача о клике в произвольном графе также является NP-полной. Опять обратите внимание на направление сводимости.

5.5.3 Решение NP-полных задач

Для NP-полных задач нет полиномиальных алгоритмов. Это в настоящее время не доказано, но является предметом почти повсеместной *веры*. Это не значит, что для этих задач вообще нет алгоритмов. Алгоритмы есть! только они требуют полного перебора вариантов, и поэтому их время работы очень быстро растет с размером задачи. Однако, не все реальные задачи большие. Часто на практике встречаются задачи достаточно небольшой размерности (десятки вершин в графе). Поэтому вопрос об алгоритмах для NP-полных задач не является бессмысленным. Итак, решать NP-полные задачи надо! Мы упоминали выше, какими важными являются, к примеру, задачи о клике и гамильтоновом пути, как много применений имеют эти задачи.

Во-первых, не все NP-полные задачи для общего случая являются таковыми для некоторых специальных случаев. Например, задача о максимальной клике является NP-полной для общего вида графов, но является тривиальной для деревьев. Дерево по определению ациклический граф, а в любой клике мощностью больше трех всегда есть циклы. Поэтому, даже не глядя на дерево, можно сказать, что мощность максимальной клики равна

двум.

Во вторых, для NP-полных задач существуют эвристические алгоритмы решения NP-полных задач. Эвристическими называются методы, сокращающие полный перебор при решении задачи. Такие методы дают не точные, а приближенные решения. Например, ищут не максимальную клику, а достаточно большую, или ищут не совсем клику, а очень плотный подграф. Для эвристических алгоритмов нужно делать оценку качества их работы.

Часто можно сделать некоторые предварительные действия для того, чтобы сократить перебор. Например, в задаче о поиске клики хорошо бы сначала выделить связные компоненты, поскольку клики по определению являются связными компонентами. Далее, можно попробовать в графе выделить в графе Другие структуры. Ясно, что клика размером более двух вершин обязательно содержит циклы. При декомпозиции графа мы ищем в графе определенные точки и составные части, по которым граф можно было бы разделить на подграфы. Такое деление часто облегчает задачу, к примеру, задачу поиска клики. Неориентированный граф можно разбить на узлы, мосты, деревья и двусвязные компоненты (см.рис.5.5):

Двусвязная компонента. Подграфы, в которых две любые вершины принадлежат общему циклу

Узел. Вершина, соединяющая две двусвязные компоненты графа, или, что то же самое связные компоненты , между которыми нет ребер.

Мост. Цепочки ребер, соединяющие две двусвязные компоненты, между которыми больше нет ребер.

Дерево. Ациклический подграф.

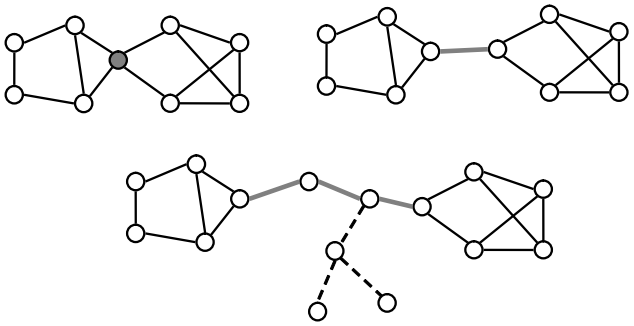


Рис. 5.5: Двусвязные компоненты (показаны обычными линиями), узлы (серые вершины), мосты (толстые серые ребра), деревья (показаны штриховой линией).

Если в графе несколько двусвязных компонент, соединенных ребрами, мостами и т.п., гораздо удобней искать клику в каждой компоненте по отдельности, поскольку клики мощностью больше, чем количество вершин в

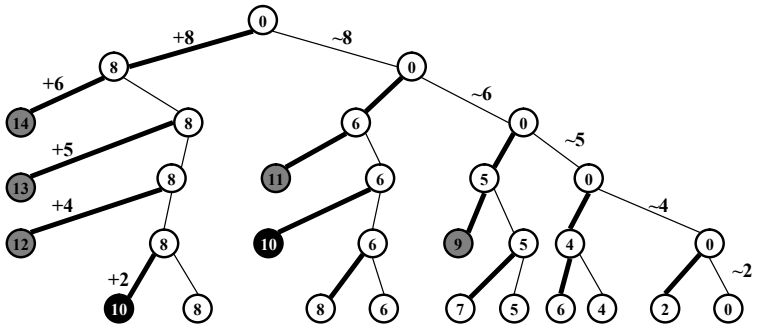


Рис. 5.6: Дерево поиска оптимальной загрузки рюкзака. Множество весов: 8,6,5,4,2, порог равен 10. толстые ребра отвечают включению элемента в список, а тонкие — исключению из списка. Серые вершины отвечают бесперспективным путям, черные вершины — найденным решениям.

самой большой двусвязной компоненте, в таком графе нет. Кстати, в отличие от клик, для двусвязных компонент существует эффективный алгоритм поиска.

5.5.4 Переборные алгоритмы

Обычно перебор ведется с помощью обхода некоторого дерева. Рассмотрим задачу сумме чисел. Дан набор положительных чисел $S = \{x_1, x_2, \cdots, x_n\}$ и некоторое положительное число t. Надо найти такое подмножество $T = \{x_{i_1}, x_{i_2}, \cdots, x_{i_k}\} \subseteq S$, чтобы их сумма $s = \sum_j^k x_{i_j}$ была максимальной, но не превышала t:

$$s \to \max | s \leq t$$

Пусть множество S упорядочено по убыванию (пусть первым будет самый тяжелый элемент). На первом шаге у нас есть выбор: брать первый элемент или нет. Если он тяжелее ограничения, то мы заведомо его не берем (и не рассматриваем вариантов, содержащих его). На каждом следующем шаге мы стоим перед выбором: или брать очередной элемент или не брать. При этом проверяем перспективность дальнейшего. Простейшее правило остановки — если добавление самого легкого элемента переводит за порог, то дальнейшее продвижение неперспективно. Если он превышает ранее зафиксированный результат, то запоминаем его в качестве максимума.

Здесь возможна еще одна эвристика. Если мы в каждой позиции будем помнить сумму весов последующих элементов, то мы сможем отсекать больше неперспективных путей. В этом алгоритме мы ветвимся и отсекаем неперспективные пути. Поэтому метод называется методом ветвей и границ.

Упражнение 34. *Как можно использовать эвристику суммы остатков?*

5.6 Стохастические алгоритмы

Допустим, нам нужно посчитать такой замечательный интеграл:

$$\int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 f(x,y,z,t,u,v,w)dxdydzdtdudvdw$$

Функция определена в семимерном пространстве, и считать точное значение интеграла очень долго. При любом регулярном методе подсчета нам нужно семь вложенных циклов. Для разумной точности надо в каждом цикле сделать порядка ста шагов. Итого получаем примерно 10^14 операций. Можно предложить более эффективный способ подсчета. Воспользуемся знанием математической статистики и попробуем посчитать значение интеграла как среднее по большой выборке частных значений. Для этого берем какие-то конкретные случайные значения всех семи переменных из равномерного распределения на отрезке $[0,1]$: $rx, ry \cdots rw$ и подставляем в функцию f, получаем некое значение этой S. Так делаем много — 10000 уже вполне достаточно — раз. Берем все 10^4 значений S и считаем среднее. Получаем значение интеграла с хорошим приближением. При подсчете этого интеграла на сетке с шагом 10^-2 объем вычислений составляет 10^14, что на много порядков больше, чем стохастический подсчет.

Идея использования случайных чисел широко применяется для решения многих сложных задач оптимизации. Эта группа алгоритмов имеет общее название методы Монте-Карло.

5.6.1 Искусственный отжиг

Пусть нам надо оптимизировать (допустим, минимизировать) некую функцию на некотором множестве. Дано множество S (конечное или бесконечное), функция $f : S \to R$. Задача: найти элемент множества $s^* \in S$ такой, что $f(s^*) \to \min$. Пусть, например, мы хотим найти Гамильтонов цикл в графе. На самом деле, мы заранее не знаем есть ли такой цикл в графе, но понимаем, что хотелось бы найти такой цикл, в котором вершины не слишком часто повторялись и чтобы поменьше было не пройденных вершин. Получаем задачу оптимизации, в которой S — множество циклов в графе, а функция f — число "неправильных"вершин.

Множество S мы в дальнейшем будем называть множеством состояний, а функцию f энергией. На множестве состояний S строится неориентированный граф: вершины соединяются ребром, если эти состояния (в некотором смысле) не сильно отличаются. Например, в случае поиска Гамильтонова цикла, множество S — это множество циклов, а два цикла называются соседними, если они имеют общее множество ребер. Важно, чтобы граф состояний был связным, т.е., чтобы из любого состояния в другое состояние был бы путь через некоторые промежуточные состояния. Надо понимать, что граф состояний очень большой и не описан явно. Теперь мы начинаем блуждать по графу, просматривая соседние состояния. Для этого нам не нужно полное описание графа состояний, но для каждого состояния надо

уметь находить множество соседей $\{s_i\}_{neighbor}$. Важно, чтобы множество соседей было не очень большим.

Находясь в состоянии s выбираем произвольное состояние s_{new}, принадлежащие множеству соседних состояний $\{s_i\}_{neighbor}$. Рассмотрим, насколько изменится энергия, если мы перейдем в новое состояние s_{new} : $\Delta f = f(s_{new}) - f(s)$. Чем меньше это значение, тем полезнее будет переход системы в состояние s_{new} с точки зрения минимизации энергии f. Если переход в какое-то соседнее состояние оказался энергетически выгоден ($\Delta f < 0$), то мы переходим в новое состояние. Казалось бы мы так будем всегда уменьшать энергию, пока не дойдем до минимума. Однако этот минимум будет локальным, а движение только в сторону уменьшения энергии исключает возможность нащупать глобальный минимум. Кроме того, если несколько соседних состояний лучше текущего, то, выбрав неправильное направление, мы можем попасть в тупик. Поэтому нам надо оставить возможность выбираться из локального минимума и переваливать через высокие барьеры. Граф состояний не ориентирован, следовательно, если состояние s_{new} является соседом состояния s, то и состояние s является соседом состояния s_{new}. Поэтому, введя обратимость переходов, мы всегда можем вернуться в исходное состояние.

Вспоминая статистическую физику, мы знаем, что отношение вероятности перехода из состояния a в состояние b к вероятности обратного перехода определяется законом:

$$\frac{p(a \to b)}{p(b \to a)} = \exp(-\frac{G(b) - G(a)}{kT})$$

Поскольку одна из вероятностей равна 1 (всегда переходим в сторону уменьшения энергии), то другая вероятность должна определяться из этого соотношения. Получаем формулу для определения вероятностей переходов:

$$p(s \to s_{new}) = \begin{cases} 1 & \Delta f < 0 \\ \exp\left(-\frac{\Delta f}{T}\right) & \Delta f \geq 0 \end{cases}$$

Теперь можно описать весь алгоритм.

1. Выбираем произвольное допустимое состояние $s \in S$
2. Для текущего состояния выбираем произвольное соседнее состояние $s_{new} \in S$
3. Вычисляем вероятность перехода $p(s \to s_{new})$
4. Разыгрываем случайную величину r, распределенную равномерно на $[0, 1]$. Если $p(s \to s_{new}) < r$, то переходим в новое состояние.
5. Переходим к п. 2.

Здесь надо заметить, что если температура T очень велика по сравнению с Δf, то мы будем просто свободно блуждать по пространству состояний. Если же $T \ll \Delta f$, то вероятности невыгодных переходов будут очень малы и мы будем быстро приходить в локальный минимум. Поэтому в этом алгоритме используют процедуру понижения температуры. Обычно температуру полагают равной

$$T = \frac{1}{\alpha\sqrt{i}},$$

где α — некоторый параметр, i — номер итерации.

Существует теорема, что при достаточно плавном понижении температуры процедура сходится к глобальному минимуму. Однако скорость сходимости сильно зависит от стратегии понижения температуры, от определения функции энергии, от определения пространства состояний и множества соседних состояний. Поскольку в процессе работы алгоритма мы постоянно понижаем температуру, то эта процедура напоминает отжиг. Поэтому алгоритм называется алгоритмом искусственного отжига (Annealing). Авторство такого подхода принадлежит Метрополису и Гастингсу (1954). Другой алгоритм, имеющий аналогию в статистической физике, устроен точно также, но без понижения температуры. В процессе блуждания по пространству мы в каких-то состояниях бываем чаще, а в каких-то реже. Наиболее посещаемые состояния на самом деле являются оптимальными. Этот алгоритм называется Гиббс самплер. Оба подхода имеют достаточно широкое применение в биоинформатике.

5.6.2 Генетические алгоритмы

Генетические алгоритмы — стохастические, эвристические оптимизационные методы, впервые предложенные Холландом (1975). Предложенная Ч. Дарвином в 1859г. эволюционная теория сильно влияла на мировоззрение людей с самого ее появления. Дарвин выявил главный механизм развития живого: отбор в сочетании с изменчивостью. В общем-то, неудивительно, что ученые, занимающиеся компьютерными исследованиями, обратились к теории эволюции в поисках вдохновения. Итак, идея генетического алгоритма заимствована у живой природы и состоит в организации эволюционного процесса, конечной целью которого является нахождение оптимального решения в сложной комбинаторной задаче. Разработчик генетических алгоритмов выступает, в данном случае, как "создатель устанавливающий законы "эволюции для наиболее быстрого достижения желаемой цели.

Введем основные понятия, важные для генетических алгоритмов. Допустимое решение описывается как массив значений (например, битов или байтов) и называется **генотипом** или **особью** — в этой науке это одно и тоже. Значение каждого признака (бита, байта) называется **аллельным состоянием**. Допустимое решение должно обладать свойством: Если мы возьмем часть значений из одного допустимого решения, а остальные значения из другого, то получим снова допустимое решение.

Функция приспособленности f (от слова fitness) — вычисляется по генотипу и характеризует качество соответствующего допустимого решения.

Мутация — изменение аллельного состояния одного из признаков

Кроссинговер — порождение новой особи в результате скрещивания двух родительских особей. Новая особь получает часть генов от одного родителя, остальные — от другого. Обычно используется схема случайного и независимого выбора (сцепленности генов здесь нет)

Популяция — совокупность из некоторого количества особей (обычно размер популяций составляет сотни особей)

Генетический алгоритм:

1. Создается популяция случайных особей.
2. Выбирается несколько произвольных особей (это количество — параметр алгоритма) и проводится мутагенез по некоторому количеству случайных аллелей.
3. Выбирается произвольная пара особей и осуществляется скрещивание. В результате появляется новая особь.
4. С вероятностями, зависящими от функции приспособленности f (например $p_i = exp(-f_i)/Z, Z = \sum_i exp(-f_i)$) выбирается жертва естественного отбора, и соответствующая особь удаляется из популяции.
5. Переход к п.2.

Эволюция популяции приводит к появлению оптимальных особей (набор аллелей становится примерно одинаковым у каждой). О сходимости можно судить по двум признакам — по среднему по популяции значению функции приспособленности и однородности популяции, т.е. по дисперсии функции приспособленности. Успех или не успех применения генетического алгоритма в большой степени зависит от искусства. Надо правильно построить генотипы, правильно сформировать функцию приспособленности, правильно подобрать параметры мутирования. Обычно хорошие результаты можно получить уже через 40 "поколений"популяции (каждое поколение означает выполнение действий, описанных в 1-3 пунктах алгоритма).

Правильно запрограммированные генетические алгоритмы могут быть очень эффективными. Применение их весьма широко: это и разнообразные задачи на графах (задача коммивояжера, например), и оптимизация запросов в БД, и биоинформатика (свертывание белков, докинг) и многое другое.

Упражнение 35. *Постройте схему генетического алгоритма для поиска максимальной клики в графе.*

Глава 6

Заключение

Описанные в этой книге структуры, идеи, алгоритмы являются лишь небольшой частью айсберга, имя которому теоретическая информатика. Существует и используется гораздо более широкий арсенал подходов, и методов решения разного рода алгоритмических задач. Искусство применения теоретической информатики заключается, прежде всего, в том, чтобы для прикладной задачи найти правильную формализацию и разглядеть черты уже известных задач теоретической информатики. В большинстве удается новую прикладную задачу свести к уже известной алгоритмической задаче. Однако, одна и та же прикладная задача может быть формализована разными путями, что может привести к разным не эквивалентным алгоритмическим задачам.

Хочется предупредить читателя, что изобретательство велосипеда дело достойное и интересное, но мало продуктивное. Поэтому прежде, чем пытаться построить новый алгоритм для решения какой-либо прикладной задачи, хорошо бы посмотреть, а нет ли уже известных подходящих алгоритмов. Если же не удалось найти подходящего алгоритма решения задачи, и Вы построили новый алгоритм, то надо озаботиться тем, чтобы попробовать доказать теоретически, что предложенный алгоритм действительно решает поставленную задачу. Строить собственные алгоритмы необходимо не с нуля, а опираясь на уже существующие алгоритмы.

В заключении можно порекомендовать следующую литературу:

Предметный указатель

Предметный указатель