ЧАСТЬ 1

ОСНОВЫ АЛГОРИТМИЗАЦИИ

Глава 1
Структурная организация данных

Обработка информации на персональных электронных вычислительных машинах (ПЭВМ) требует, чтобы ее структура была определена и точно представлена в ПЭВМ. Информация, представленная в формализованном виде, пригодном для автоматизированной обработки, называется ***данными***. Структура данных определяет их семантику, а также способы организации данных и управления ими. При использовании компьютера для хранения и обработки данных необходимо точно определить тип и структуру данных, а также найти способ наиболее естественного их представления. Структуры данных и алгоритмы служат тем материалом, из которого строятся программы.

Без понимания структур данных и алгоритмов невозможно создать серьезный программный продукт — они служат базовыми элементами любой программы. В организации структур данных и процедур их обработки заложена возможность проверки правильности работы программы.

1.1. Основные понятия структур данных

Компьютер оперирует только с одним видом данных — с отдельными битами, или двоичными цифрами, и работает с этими данными только в соответствии с неизменным набором алгоритмов,

которые определяются системой команд центрального процессора. Задачи, которые решаются с помощью компьютера, редко выражаются на языке битов. Как правило, данные имеют форму чисел, литер, текстов, символов и более сложных структур типа последовательностей, списков и деревьев. Еще разнообразнее алгоритмы, применяемые для решения различных задач; фактически алгоритмов не меньше, чем вычислительных задач.

Структура данных, рассматриваемая без учета ее представления в машинной памяти, называется ***абстрактной***, или ***логической***. Понятие «***физическая структура данных***» отражает способ физического представления данных в машинной памяти. Вследствие различия между логической и соответствующей ей физической структурами в вычислительной системе существуют процедуры, осуществляющие отображение логической структуры в физическую, и наоборот.

Например, доступ к элементу двумерного массива на логическом уровне реализуется указанием номеров строки и столбца в прямоугольной таблице, на пересечении которых расположен соответствующий элемент. На физическом же уровне к элементу массива доступ осуществляется с помощью функции адресации, которая при известном начальном адресе массива в машинной памяти преобразует номера строки и столбца в адрес соответствующего элемента массива. Таким образом, каждую структуру данных можно характеризовать ее логическим (абстрактным) и физическим (конкретным) представлениями, а также совокупностью операций на этих двух уровнях представления структуры (рис. 1.1).

Очень часто, говоря о той или иной структуре данных, имеют в виду ее логическое представление, так как физическое представление обычно скрыто от программиста. Так как физическая структура данных реализуется в машинной памяти, имеющей ограниченный объем, то при изучении такой структуры должна учитываться проблема распределения и управления памятью.

Структуры данных, применяемые в алгоритмах, могут быть чрезвычайно сложными. В результате выбор правильного представления данных часто служит ключом к удачному программированию и может в большей степени сказываться на производительности программы, чем детали используемого алгоритма.

Под ***структурой данных*** в общем случае понимают множество элементов данных и множество связей между ними. Такое определение охватывает все возможные подходы к структуризации данных, но

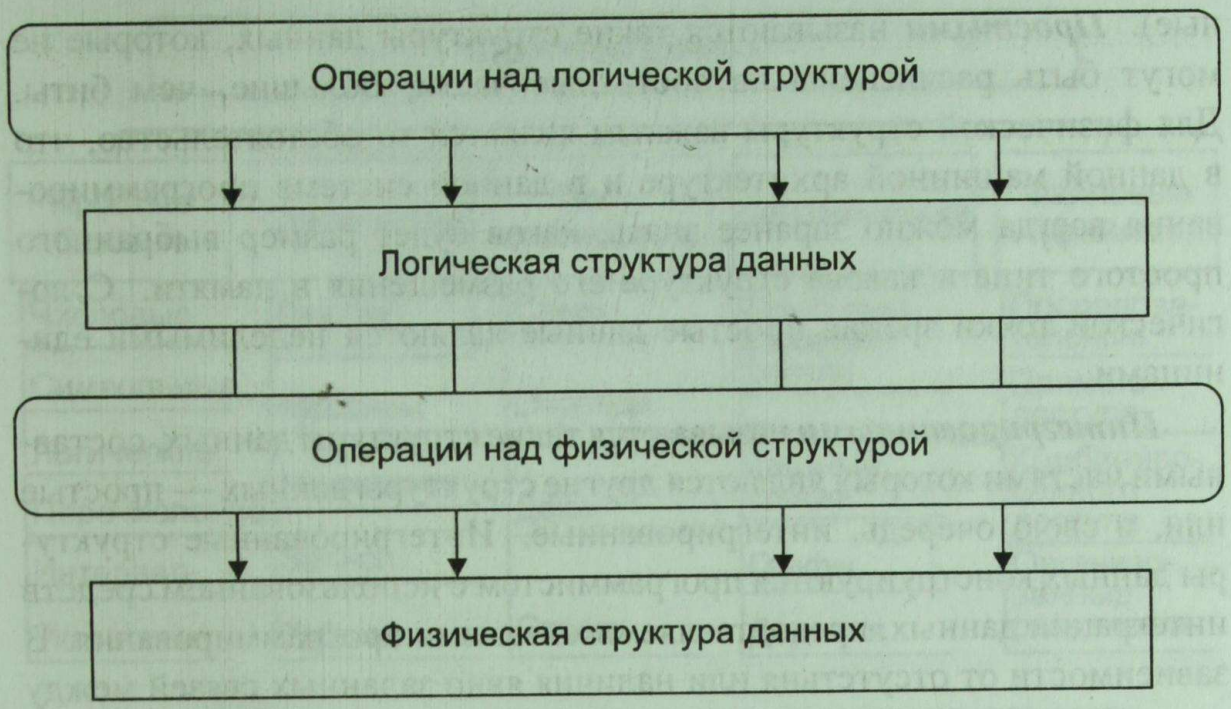


Рис. 1.1. Уровни представления структуры данных

в каждой конкретной задаче используются те или иные его аспекты. Поэтому вводится дополнительная классификация структур данных, направления которой соответствуют различным аспектам их рассмотрения. Прежде чем приступать к изучению конкретных структур данных, дадим их общую классификацию по нескольким признакам. Понятие «физическая структура данных» отражает способ физического представления данных в памяти ПЭВМ и называется еще структурой хранения, внутренней структурой или структурой памяти. Рассмотрение структуры данных без учета ее представления в памяти компьютера называется абстрактной, или логической, структурой. В общем случае между логической и соответствующей ей физической структурами существует различие, степень которого зависит от самой структуры и особенностей той среды, в которой она должна быть отражена. Вследствие этого различия существуют процедуры, осуществляющие отображение логической структуры в физическую и, наоборот, физической структуры в логическую. Эти процедуры обеспечивают, кроме того, доступ к физическим структурам и выполнение над ними различных операций, причем каждая операция рассматривается применительно к логической или физической структуре данных.

Различают простые (базовые, примитивные) структуры (типы) данных и интегрированные (структурированные, композитные, слож-

ные). ***Простыми*** называются такие структуры данных, которые не могут быть расчленены на составные части, бóльшие, чем биты. Для физической структуры важным является то обстоятельство, что в данной машинной архитектуре и в данной системе программирования всегда можно заранее знать, каков будет размер выбранного простого типа и какова структура его размещения в памяти. С логической точки зрения простые данные являются неделимыми единицами.

Интегрированными называются такие структуры данных, составными частями которых являются другие структуры данных — простые или, в свою очередь, интегрированные. Интегрированные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования. В зависимости от отсутствия или наличия явно заданных связей между элементами данных следует различать ***несвязные структуры*** (векторы, массивы, строки, стеки, очереди) и ***связные структуры*** (связные списки).

1.2. Классификация структур данных по признаку изменчивости

Важный признак структуры данных — ее изменчивость, т.е. изменение числа элементов и (или) связей между элементами структуры. В определении изменчивости структуры не отражен факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости.

По признаку изменчивости различают структуры ***базовые***, ***статические***, ***полустатические***, ***динамические*** и ***файловые***. Классификация структур данных (СД) по признаку изменчивости приведена на рис. 1.2. Базовые структуры данных, статические, полустатические и динамические характерны для оперативной памяти и часто называются оперативными структурами. Файловые структуры соответствуют структурам данных для внешней памяти.

Вектор (одномерный массив) — структура данных с фиксированным числом элементов одного и того же типа.

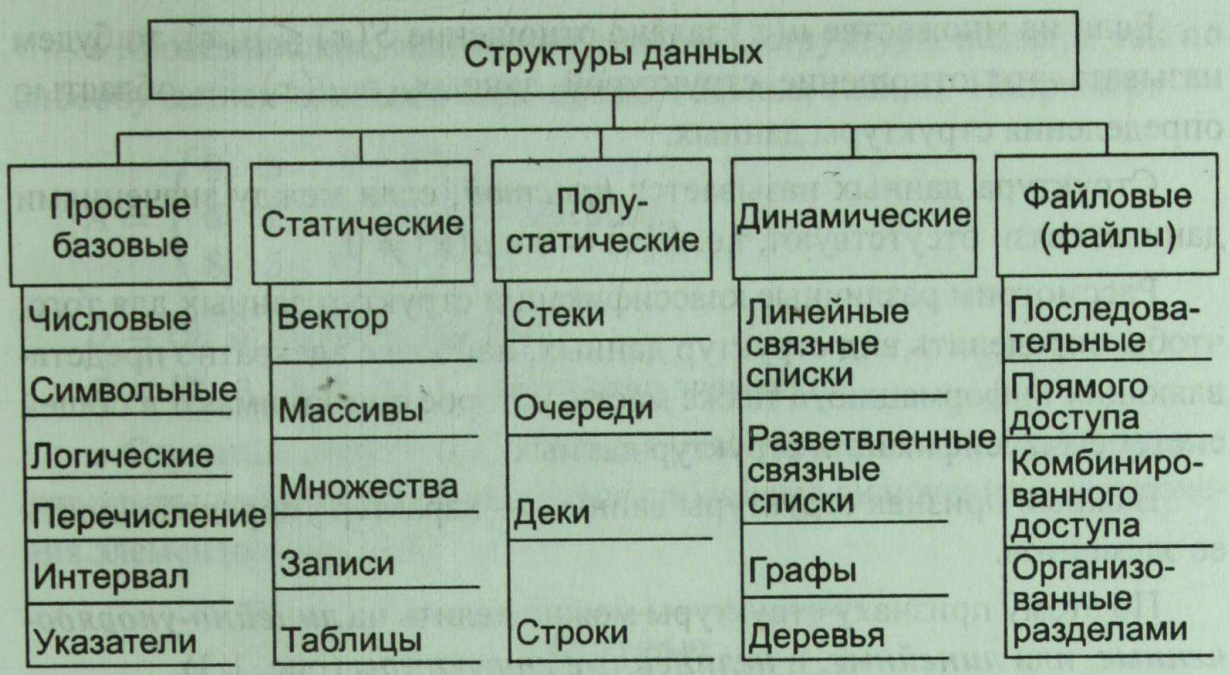


Рис. 1.2. Классификация структур данных по признаку изменчивости

Массив — последовательность элементов одного типа, называемого базовым.

Множество — такая структура, которая представляет собой набор неповторяющихся данных одного и того же типа.

Запись — конечное упорядоченное множество полей, характеризующихся различным типом данных.

Таблица — последовательность записей, которые имеют одну и ту же организацию.

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется линейным.

1.3. Линейные и нелинейные структуры данных

Напомним, что структуры данных представляют собой отношения, заданные на множествах данных.

Пусть X — множество данных, Y — множество значений данных:

$$X \xrightarrow{\mu} Y \quad \text{и при этом} \quad \mu(x) = \{\overline{x}_i : x_i \in X,\ \overline{x}_i \in Y\}.$$

Если на множестве $\mu(x)$ задано отношение $S(x) \leq \mu(x)$, то будем называть это отношение структурой данных, а $\mu(x)$ — областью определения структуры данных.

Структура данных называется ***простой***, если между значениями данных связи отсутствуют, т.е. $S(x) = 0$ и $\mu(x) \neq 0$.

Рассмотрим различные классификации структур данных для того, чтобы определить вид структур данных, наиболее адекватно представляющих информацию, а также место, которое они занимают в общей системе классификации структур данных.

Важный признак структуры данных — характер упорядоченности ее элементов.

По этому признаку структуры можно делить на ***линейно-упорядоченные***, или ***линейные***, и ***нелинейные структуры*** (рис. 1.3).

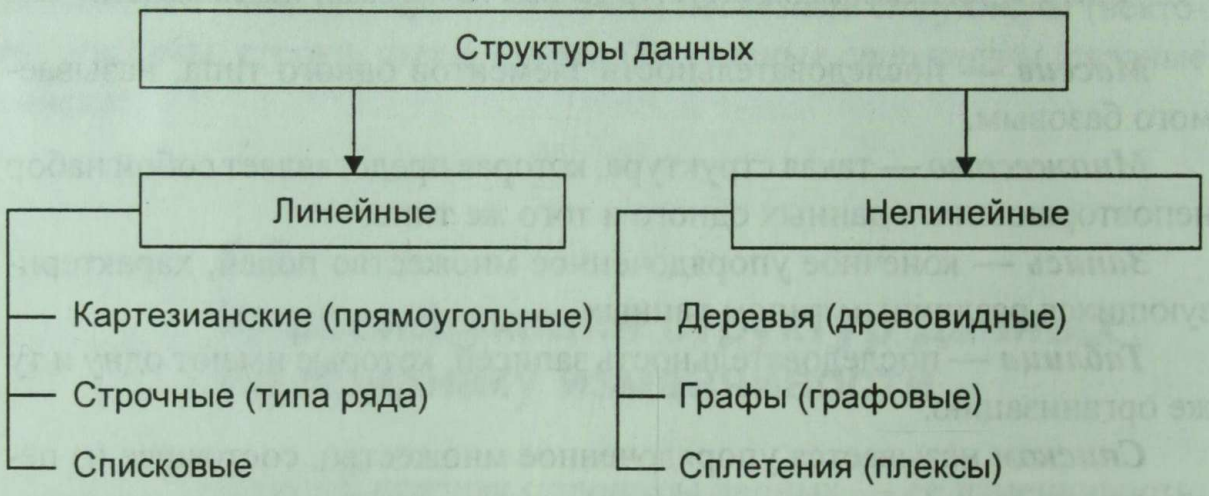


Рис. 1.3. Линейные и нелинейные структуры данных

В зависимости от характера взаимного расположения элементов в памяти ПЭВМ линейные структуры можно разделить на структуры с последовательным распределением их элементов в памяти (векторы, строки, массивы, стеки, очереди) и структуры с произвольным связанным распределением элементов в памяти (односвязные, двухсвязные и прочие списки).

Линейные структуры данных. ***Линейные структуры данных*** (СД) — это структуры, в которых связи между элементами не зависят от выполнения какого-либо условия. Линейные структуры подразделяются на три типа: картезианские, строчные и списковые.

• Картезианские, или прямоугольные, структуры названы так по способу записи данных в виде прямоугольных таблиц. Например:

$$A = \begin{pmatrix} 0 & 3 & 7 & 9 \\ 6 & 2 & 1 & 0 \\ 8 & 5 & 10 & 6 \end{pmatrix}$$ — матрица;

$B = (9, 3, 6, 5)$ — вектор;

$Z = \{7, 6, 0, 2, 3\}$ — множество элементов.

• Строчные структуры — одномерные, динамически изменяемые структуры данных, различающиеся способами включения и исключения элементов (рис. 1.4).

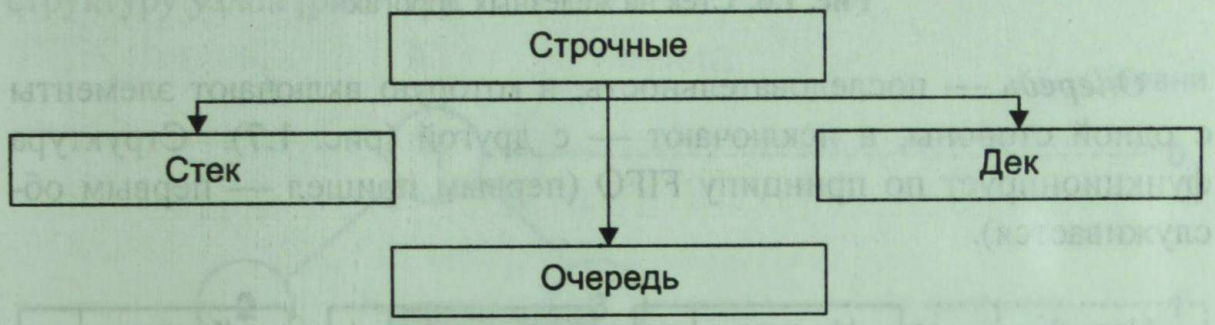


Рис. 1.4. Строчные структуры данных

Стек — это последовательность, в которой включение и исключение элемента осуществляется с одной стороны последовательности (рис. 1.5).

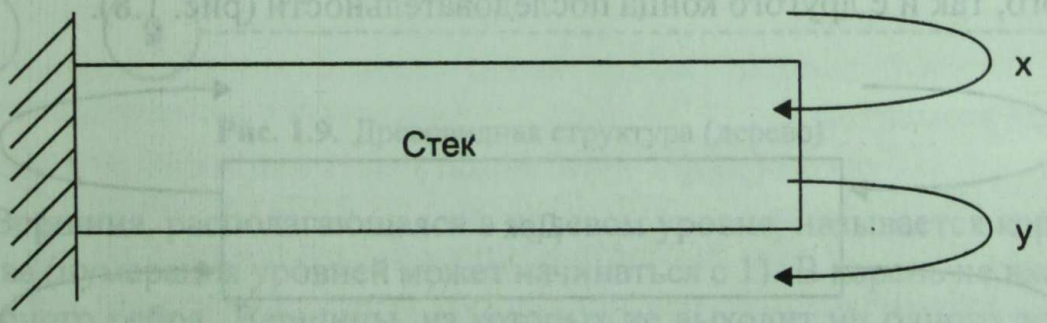


Рис. 1.5. Схема доступа к элементам стека

Пусть, например, дано множество элементов {2, 5, 7, 1,9,3} и задан алгоритм работы стека (X, X, X, Y, X, Y, Y). Тогда после работы алгоритма в стеке останется число 2.

Известные примеры стека — винтовочный патронный магазин, железнодорожный разъезд для сортировки вагонов (рис. 1.6).

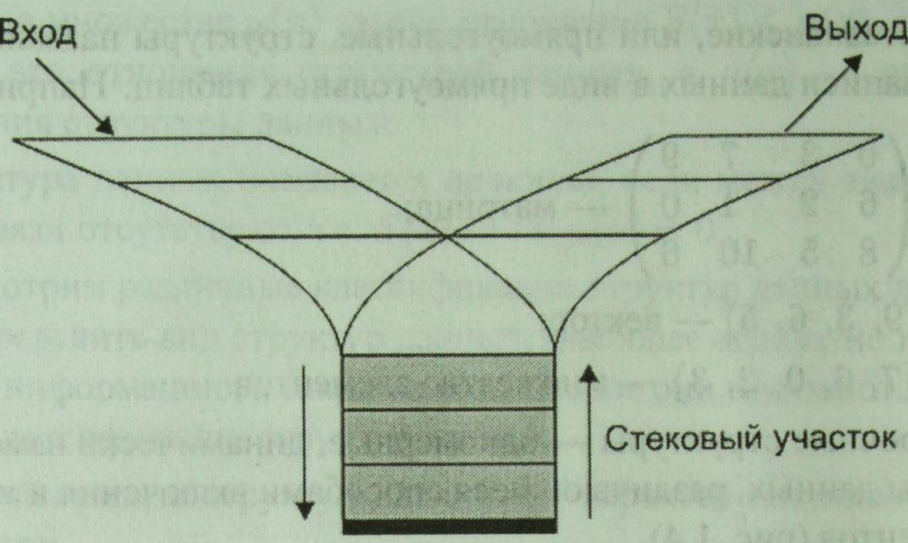


Рис. 1.6. Стек на железных дорогах

Очередь — последовательность, в которую включают элементы с одной стороны, а исключают — с другой (рис. 1.7). Структура функционирует по принципу FIFO (первым пришел — первым обслуживается).

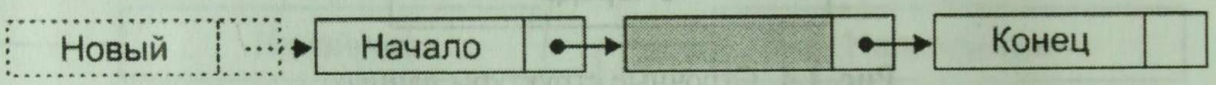


Рис. 1.7. Схема доступа к элементам очереди

Дек — линейная структура (последовательность), в которой операции включения и исключения элементов могут выполняться как с одного, так и с другого конца последовательности (рис. 1.8).

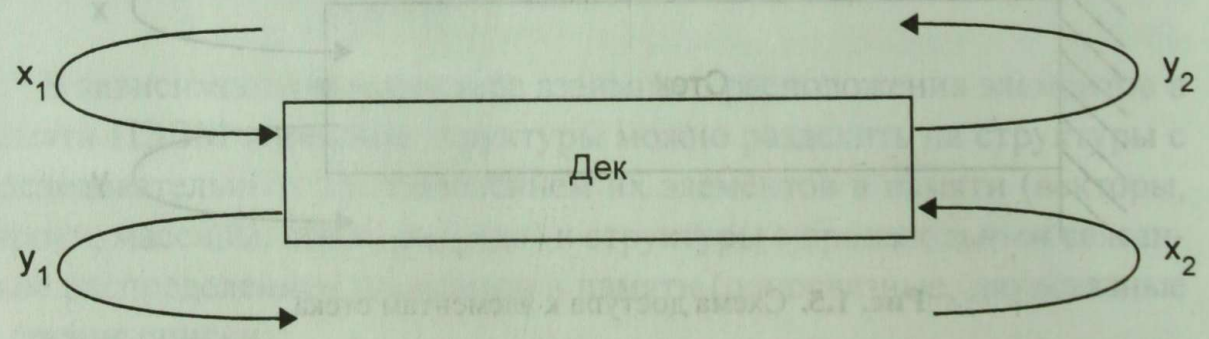


Рис. 1.8. Схема доступа к элементам дека

В ***списковых структурах*** логический порядок данных определяется указателями. Любая списковая структура представляет собой набор элементов, каждый из которых состоит из двух полей: в одном

из них размещен элемент данных или указатель на него, а в другом — указатель на следующий элемент списка.

Нелинейные структуры данных. ***Нелинейные структуры данных*** — это СД, у которых связи между элементами зависят от выполнения определенного условия. Примеры нелинейных структур — деревья, графы, многосвязные списки.

• ***Древовидные структуры*** — это иерархические структуры, состоящие из набора вершин и ребер, каждая вершина содержит определенную информацию и ссылку на вершину нижнего уровня. ***Дерево*** — это совокупность элементов, называемых ***узлами*** (один из которых определен как ***корень***), и отношений, образующих иерархическую структуру узлов (рис. 1.9).

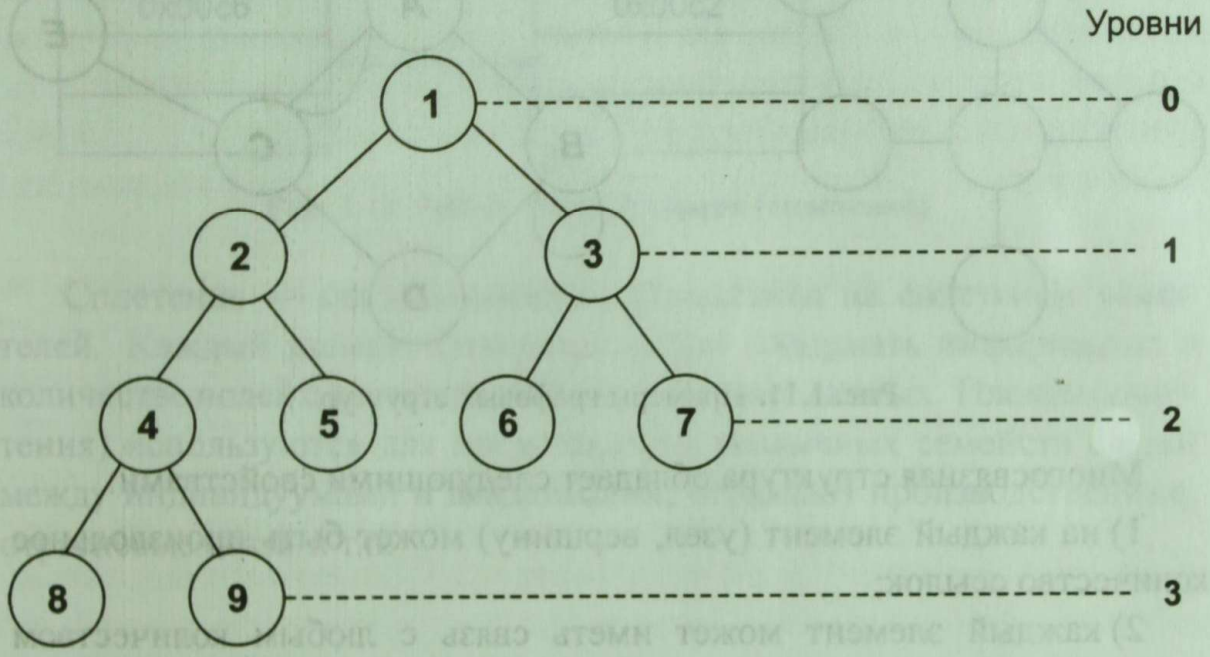


Рис. 1.9. Древовидная структура (дерево)

Вершина, располагающаяся в нулевом уровне, называется корнем дерева (нумерация уровней может начинаться с 1). В корень не входит ни одного ребра. Вершины, из которых не выходит ни одного ребра, называются ***листьями*** (вершины 8, 9, 5, 6, 7). Дерево, из каждой вершины которого выходит только по два ребра, называется ***бинарным*** (рис. 1.10).

• ***Графы*** представляют собой совокупность двух множеств: вершин и ребер. Граф — это сложная нелинейная многосвязная динамическая структура, отображающая свойства и связи сложного объекта (рис. 1.11).

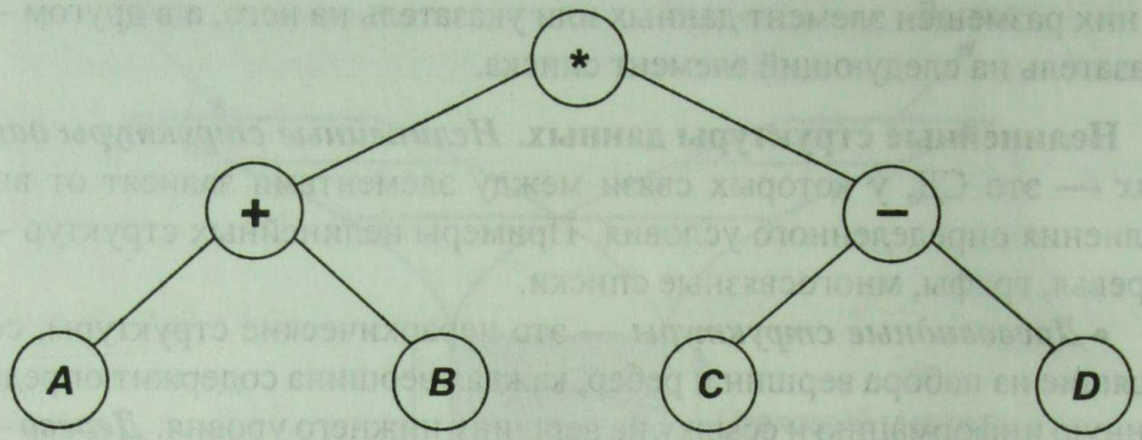


Рис. 1.10. Бинарное дерево

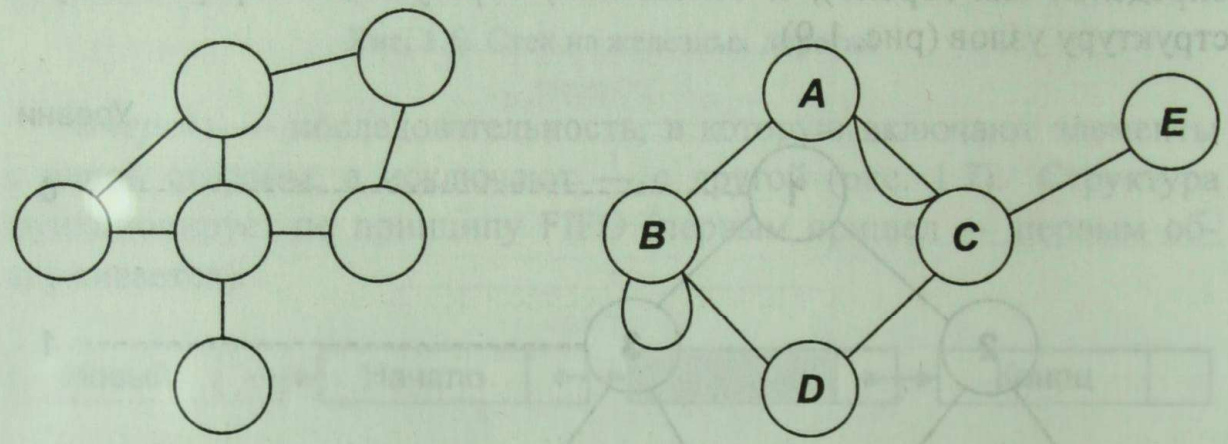


Рис. 1.11. Примеры графовых структур

Многосвязная структура обладает следующими свойствами:

1) на каждый элемент (узел, вершину) может быть произвольное количество ссылок;

2) каждый элемент может иметь связь с любым количеством других элементов;

3) каждая связка (ребро, дуга) может иметь направление и вес.

Типичными графами являются схемы авиалиний и схемы метро, а на географических картах — изображение железных или автомобильных дорог. Выбранные точки графа называются его ***вершинами***, а соединяющие их линии — ***ребрами***.

• ***Сплетение*** (многосвязный список, плекс) — это нелинейная структура данных, объединяющая такие понятия, как дерево, граф и списковая структура.

Основное свойство сплетений, отличное от других типов структур, — наличие у каждого элемента сплетения нескольких полей с указателями на другие элементы того же сплетения (рис. 1.12).

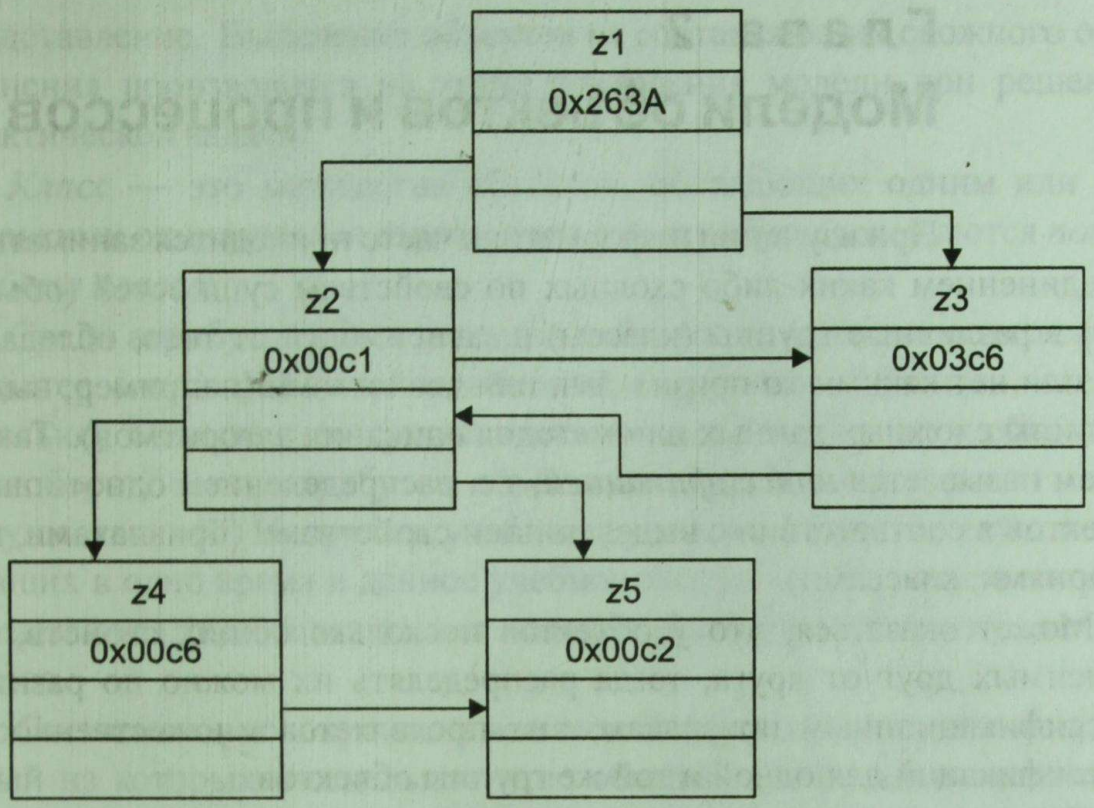


Рис. 1.12. Многосвязный список (сплетение)

Сплетение — связь элементов, основанная на сплетении указателей. Каждый элемент сплетения может содержать информацию о количестве полей с указателями и формате поля данных. Плексы (сплетения) используются для представления различных семейств связей между индивидуумами и владельцами, отражают производственные, отраслевые связи и т.п.

Контрольные вопросы

1. Что называется структурой данных?
2. Что такое логическая и физическая структура данных?
3. Чем различаются простые и интегрированные структуры данных?
4. Назовите основные особенности статических, полустатических и динамических структур.
5. Перечислите основные типы связных списков.
6. Дайте определение линейных и нелинейных структур.
7. Чем различаются стек, очередь и дек?
8. В чем заключается основная особенность древовидных структур?
9. Приведите примеры графовых структур.

Глава 2
Модели объектов и процессов

При изучении информатики часто приходится заниматься объединением каких-либо сходных по свойствам сущностей (*объектов*) в различные группы (классы) в зависимости от того, обладают они или нет какими-то признаками или свойствами (например, выделяя типы сложных данных или методов описания алгоритмов). Такой прием называется ***классификацией***, т.е. распределением однотипных объектов в соответствии с выделенными свойствами (признаками, категориями, классами).

Может оказаться, что у объектов несколько общих свойств, не зависимых друг от друга, тогда распределять их можно по разным классификационным признакам, т.е. проявляется множественность классификаций для одной и той же группы объектов.

Объект — простейшая составляющая сложного объединения, обладающая следующими качествами:

- в рамках данной задачи он не имеет внутреннего устройства и рассматривается как единое целое;
- у него имеется набор свойств (атрибутов), которые изменяются в результате внешних воздействий;
- он идентифицирован, т.е. имеет имя (название).

Например, отдельное предприятие в рамках экономики государства может считаться простым элементом, т.е. объектом, с некоторым набором параметров, для государства существенных: характер и объем выпускаемой продукции, местонахождение, потребности в ресурсном обеспечении, количество работников и т.д. При этом не включаются в рассмотрение структура производства, количество производственных помещений, личности руководителей, цвет зданий и пр.

В другой задаче, где требуется найти оптимальную схему производства конкретного предприятия, рассматривать его как простой элемент, безусловно, нельзя. Проникновение вглубь устройства чего-либо, вообще говоря, безгранично, поэтому всегда приходится останавливаться на некотором «уровне простоты», который приемлем для данной задачи. Таким образом, отнесение каких-то составляющих сложного объединения к объектам есть не что иное, как упрощение реальной ситуации, т.е. моделирование. Объект — модельное

представление. Выделение объектов из составляющих сложного объединения производится на этапе построения модели при решении практической задачи.

Класс — это множество объектов, обладающих одним или несколькими одинаковыми атрибутами; эти атрибуты называются *полем свойств класса.*

Среди атрибутов объекта всегда имеются те, которые определяют характер его связей (взаимодействия) с другими объектами, следовательно, оказываются существенными для объединения объектов, и наоборот, часть атрибутов объекта для объединения может быть несущественной. Например, учебная группа объединяет людей, поступивших в одно время в данное учебное заведение; несущественными оказываются рост, цвет глаз и волос и прочие индивидуальные качества.

Система — совокупность взаимодействующих компонентов, каждый из которых в отдельности не обладает свойствами системы в целом, но является ее неотъемлемой частью.

Человек издавна использует моделирование для исследования объектов, явлений и процессов в различных областях. Моделирование помогает принимать обоснованные и продуманные решения, предвидеть последствия своей деятельности. Понятие «компьютерное моделирование» введено для того, чтобы отразить использование в этом процессе мощного современного средства переработки информации — компьютера.

Модель — упрощенное представление о реальном объекте, процессе или явлении. Моделирование — построение моделей для исследования и изучения объектов, процессов или явлений.

Для чего создавать модель, а не исследовать сам оригинал?

Во-первых, можно моделировать оригинал (прототип), которого уже не существует или его нет в действительности.

Во-вторых, оригинал может иметь много свойств и взаимосвязей. Для изучения какого-либо свойства иногда полезно отказаться от менее существенных, вовсе не учитывая их.

Любая классификация начинается с выделения общих свойств (признаков) и, возможно, определения их значений, по которым объекты будут распределяться в различные группы (классы). Представлена классификация может быть в графической форме (в виде графа) и в текстовой форме (в виде таблиц или списков).

2.1. Модели структурные и функциональные

Состояние прототипа — это совокупность свойств его составных частей, а также его собственных. Состояние — «моментальная» фотография прототипа для выбранного момента времени. С течением времени состояние может изменяться, тогда говорят о существовании ***процесса***. В соответствии со сказанным возможно построение ***модели состояния*** и ***модели процессов***. Модели первого типа называются ***структурными***, второго типа — ***функциональными моделями***.

Примерами структурных моделей являются чертеж какого-либо устройства, схема компьютера, блок-схема алгоритма и пр. Примером функциональных моделей является макет, демонстрирующий работу чего-либо. Важнейшим классом функциональных моделей являются ***имитационные модели***.

Имитационное моделирование — метод исследования, основанный на том, что изучаемый прототип заменяется его имитатором (натурной или информационной моделью), с которым и проводятся эксперименты с целью получения информации об особенностях прототипа.

Примером натурной имитационной установки может служить аэродинамическая труба, позволяющая исследовать воздушные потоки, обтекающие транспортное средство (самолет, автомобиль, тепловоз и пр.) при движении. В качестве имитаторов могут выступать и математические модели, реализованные на компьютере. В настоящее время именно имитационное моделирование оказывается важнейшим методом исследования и прогнозирования в науке, экономике и других отраслях знаний. Примерами являются моделирование последствий ядерной войны, прогноз погоды, экономические прогнозы и пр.

2.2. Модели натурные и информационные

Модели можно отнести к одной из двух групп: натурные (материальные) и информационные (нематериальные) — рис. 2.1.

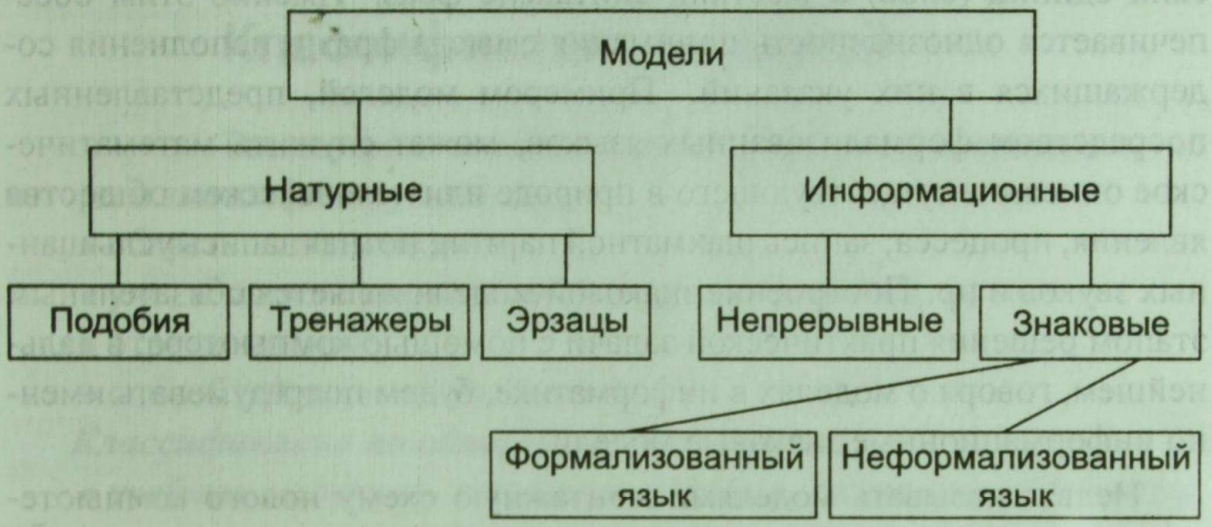


Рис. 2.1. Классификация моделей

Примерами ***натурных моделей*** подобия являются: игрушка, манекен, фотография и т.п. К натурным моделям-тренажерам следует отнести различные устройства, применяемые при подготовке летчиков, водителей и других, которые имитируют некоторую ситуацию, требующую принятия решений и действий, и позволяют отрабатывать методы ее разрешения.

Модели-эрзацы — это в первую очередь протезы, заменяющие и частично выполняющие функции настоящих органов.

Материальные модели иначе можно назвать предметными, физическими. Они воспроизводят геометрические и физические свойства оригинала и всегда имеют реальное воплощение. ***Информационная модель*** — совокупность информации, характеризующая свойства и состояние объекта, процесса, явления, а также взаимосвязь с внешним миром. Информационные модели в свою очередь подразделяются на *знаковые* и *непрерывные*.

Примером непрерывной информационной модели могут служить математическая функция и ее график.

Знаковые информационные модели можно было бы назвать также дискретными, подчеркивая то обстоятельство, что информация в

них представлена в дискретной форме, т.е. посредством некоторого алфавита и языка. Язык может быть обычным разговорным — *с неформализованным синтаксисом*. Примерами моделей, построенных посредством такого языка, являются различные описания, характеристики, мнения, толкования и пр.

Формализованные языки имеют фиксированный набор лексических единиц (слов) и жесткий синтаксис фраз. Именно этим обеспечивается однозначность понимания смысла фраз и исполнения содержащихся в них указаний. Примером моделей, представленных посредством формализованных языков, может служить математическое описание существующего в природе или человеческом обществе явления, процесса; запись шахматной партии; нотная запись услышанных звуков и пр. Построение знаковой модели является обязательным этапом решения практической задачи с помощью компьютера; в дальнейшем, говоря о моделях в информатике, будем подразумевать именно информационные знаковые модели.

Нельзя называть моделями монтажную схему нового компьютера, блок-схему разрабатываемой программы или план создаваемого литературного или научного произведения — правильнее было бы использовать какой-то иной термин, например *проект*. Если же продукт творчества обладает прототипом (например, это портрет или пейзаж с натуры), то он может считаться моделью, причем это могут быть как натурные, так и информационные модели.

Рассмотрим графическую форму модели (рис. 2.2), соответствующей следующему словесному описанию: «A учится в одной группе с B и C, но не с D и E, которые учатся в другой группе». Здесь $M = \{A, B, C, D\}$, а отношением R будет «учиться в одной группе».

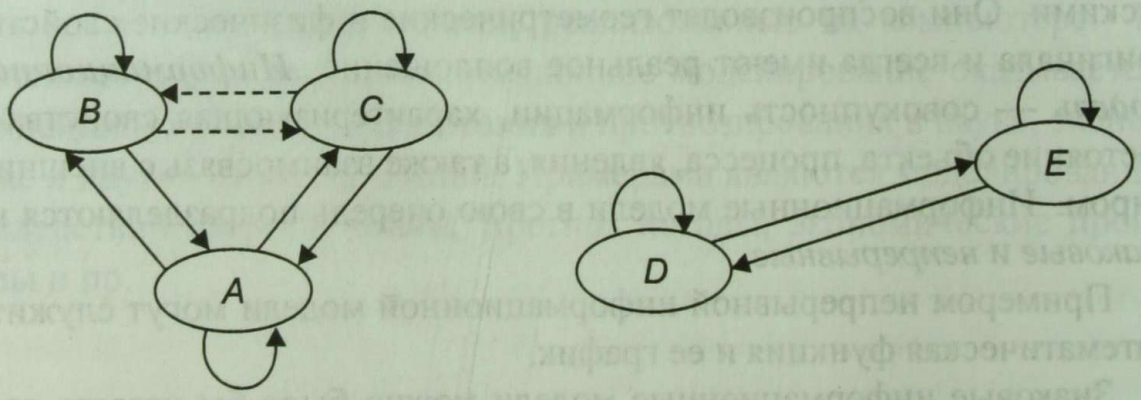


Рис. 2.2. Графическая форма модели

Вершинами графа являются элементы несущего множества, а его дугами — отношения.

Очевидно, рассматриваемое отношение транзитивно, что отражено парными пунктирными дугами, связывающими B и C.

2.3. Классификация моделей

Рассмотрим наиболее распространенные признаки, по которым классифицируются модели:

- область использования;
- учет в модели временного фактора (динамики);
- отрасль знаний;
- способ представления моделей.

Классификация по области использования:

- *учебные модели* — наглядные пособия, различные тренажеры, обучающие программы;
- *научно-технические модели* создают для исследования процессов и явлений;
- *игровые модели* — это военные, экономические, спортивные и деловые игры;
- *имитационные модели* не просто отражают реальность, а имитируют ее. Эксперимент либо многократно повторяется, либо проводится одновременно со многими другими похожими объектами, но поставленными в разные условия.

Классификация с учетом фактора времени. Модели можно разделить на *статические* (это как бы одномоментный срез информации по объекту) и *динамические*. Динамическая модель позволяет увидеть изменения объекта во времени.

Рассмотрим схему классификации моделей по способу представления (рис. 2.3).

Вербальная модель — информационная модель в мысленной или разговорной форме. К таким моделям можно отнести и идею, возникшую у изобретателя, и музыкальную тему, и рифму, прозвучавшую пока еще в сознании у автора.

Знаковая модель — информационная модель, выраженная специальными знаками, т.е. средствами любого формального языка (ри-

Рис. 2.3. Классификация моделей по способу представления

сунки, тексты, графики, схемы). По форме представления можно выделить следующие виды информационных моделей:

- *геометрические* — графические формы и объемные конструкции;
- *словесные* — устные и письменные описания с использованием иллюстраций;
- *математические* — математические формулы, отображающие связь различных параметров объекта или процесса;
- *структурные* — схемы, графики, таблицы и т.п.;
- *логические* — модели, в которых представлены различные варианты выбора действий на основе умозаключений и анализа условий;
- *специальные* — ноты, химические формулы и т.п.;
- *компьютерные* и *некомпьютерные модели.*

2.4. Этапы моделирования

Моделирование является одним из ключевых видов деятельности человека. Оно всегда в той или иной форме предшествует любому делу. Моделирование занимает центральное место в исследовании объекта. Оно позволяет обоснованно принимать решение. Решение любой задачи разбивается на несколько этапов. Моделирование — творческий процесс и заключить его в формальные рамки очень трудно. В общем виде его можно представить поэтапно (рис. 2.4).

Все этапы определяются поставленной задачей и целями моделирования. Рассмотрим основные этапы моделирования подробнее.

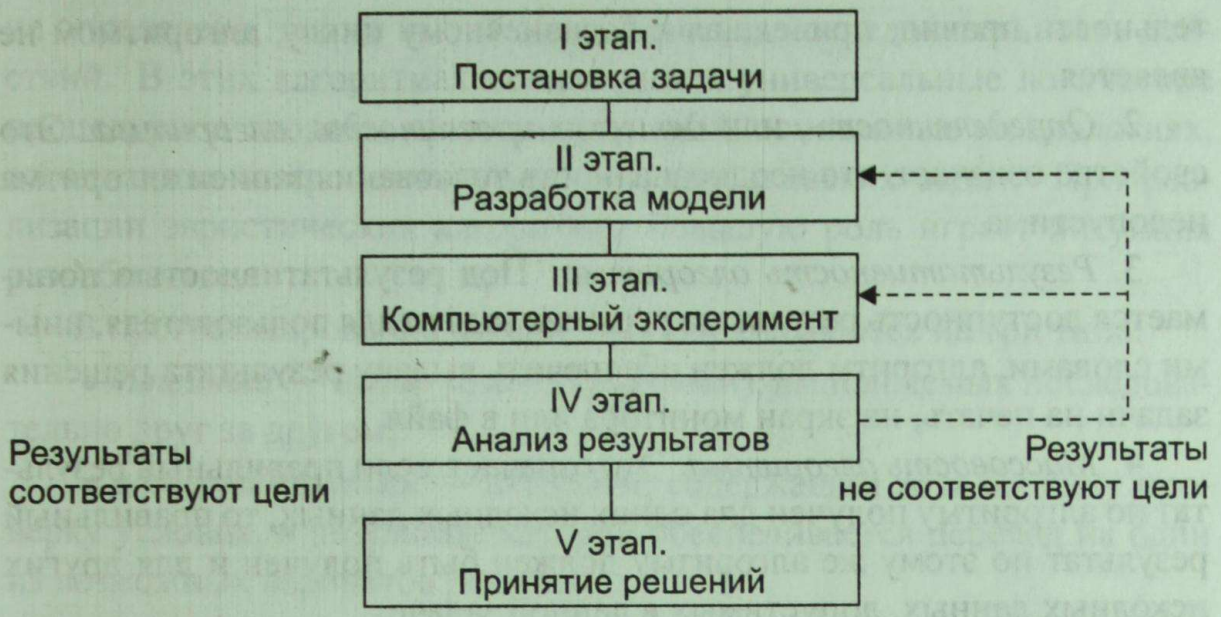


Рис. 2.4. Этапы моделирования

Этап I. *Постановка задачи.* Описание задачи. Определение цели моделирования. Анализ объекта моделирования.

Этап II. *Разработка модели.* На этом этапе выясняются свойства, состояния, действия и другие характеристики элементарных объектов. Формируется представление об элементарных объектах. Выбор наиболее существенной информации при создании информационной модели и ее сложность обусловлены целью моделирования.

Этап III. *Компьютерный эксперимент.* Тестирование — процесс проверки правильности модели.

Этап IV. *Анализ результатов моделирования.* Принятие решения, которое должно быть выработано на основе всестороннего анализа полученных результатов.

Этап V. *Конечная цель моделирования.*

2.5. Свойства алгоритма

Алгоритмом называется система формальных правил, четко и однозначно определяющая процесс решения поставленной задачи в виде конечной последовательности действий или операций.

Свойства, которыми должен обладать алгоритм:

1. *Конечность* (*финитность*) *алгоритма.* Алгоритм должен приводить к решению задачи обязательно за конечное время. Последова-

тельность правил, приведшая к бесконечному циклу, алгоритмом не является.

2. *Определенность*, или *детерминированность*, *алгоритма*. Это свойство означает, что неоднозначность толкования записи алгоритма недопустима.

3. *Результативность алгоритма*. Под результативностью понимается доступность результата решения задачи для пользователя, иными словами, алгоритм должен обеспечить выдачу результата решения задачи на печать, на экран монитора или в файл.

4. *Массовость алгоритма*. Это означает, если правильный результат по алгоритму получен для одних исходных данных, то правильный результат по этому же алгоритму должен быть получен и для других исходных данных, допустимых в данной задаче.

5. *Эффективность алгоритма*. Под эффективностью алгоритма будем понимать такое его свойство (качество), которое позволяет решить задачу за приемлемое для разработчика время. К параметру, характеризующему эффективность алгоритма, следует отнести также объем памяти компьютера, необходимый для решения задачи.

2.6. Виды алгоритмов и их реализация

В зависимости от цели, начальных условий задачи, путей ее решения, определения действий разработчика алгоритмы подразделяются на механические, или детерминированные (жесткие), и гибкие, или стохастические (вероятностные и эвристические).

Механический алгоритм задает определенные действия, обозначая их в единственной последовательности, обеспечивающей однозначный требуемый (искомый) результат в том случае, если выполняются условия процесса, для которых разработан алгоритм. К таким алгоритмам относятся алгоритмы работы машин, станков, двигателей и т.п.

Вероятностный (*стохастический*) *алгоритм* предлагает программу решения задачи несколькими путями или способами, приводящими к достижению результата.

Эвристический алгоритм (от греческого слова «эврика») — это такой алгоритм, в котором достижение конечного результата однозначно

не определено, так же как не обозначена вся последовательность действий. В этих алгоритмах используются универсальные логические процедуры и способы принятия решений, основанные на аналогиях, ассоциациях и прошлом опыте решения похожих задач. При реализации эвристических алгоритмов большую роль играет интуиция разработчика.

В программировании алгоритмы подразделяются на три типа:

• *линейный* — набор команд (указаний), выполняемых последовательно друг за другом;

• *разветвляющийся* — алгоритм, содержащий хотя бы одну проверку условия, в результате которой обеспечивается переход на один из возможных вариантов решения;

• *циклический* — алгоритм, предусматривающий многократное повторение одного и того же действия (одних и тех же операций) над новыми исходными данными. К циклическим алгоритмам сводится большинство методов вычислений и перебора вариантов.

Вспомогательный (подчиненный) алгоритм — это алгоритм, ранее разработанный и целиком используемый при алгоритмизации конкретной задачи.

В зависимости от степени детализации, поставленных целей, методов и технических средств решения задачи используются различные формы представления алгоритмов. Все варианты представления алгоритмов могут быть объединены в единую классификационную схему, изображенную на рис. 2.5.

На практике наиболее распространены следующие способы:

• словесный;
• формульно-словесный;
• блок-схемный;
• псевдокод;
• структурные диаграммы;
• языки программирования.

Словесный способ — содержание этапов вычислений задается на естественном языке в произвольной форме с требуемой детализацией.

Например, пусть задан массив чисел. Требуется проверить, все ли числа принадлежат заданному отрезку, который задается границами A и B.

Шаг 1. Выбираем первый элемент массива — к шагу 2.

Рис. 2.5. Классификация способов представления алгоритмов

Шаг 2. Сравниваем: выбранный элемент массива принадлежит интервалу — если «да», то к шагу 3, если «нет» — к шагу 6.

Шаг 3. Все элементы массива просмотрены? Если «да» — то к шагу 5, если «нет» — то к шагу 4.

Шаг 4. Выбираем следующий элемент — к шагу 2.

Шаг 5. Печать сообщения: все элементы принадлежат интервалу — на шаг 7.

Шаг 6. Печать сообщения: не все элементы принадлежат интервалу — на шаг 7.

Шаг 7. Конец.

При этом способе записи алгоритма отсутствует наглядность вычислительного процесса, так как нет достаточной формализации.

Формульно-словесный способ — задание инструкций с использованием математических символов и выражений в сочетании со словесными пояснениями.

Например, вычислить площадь треугольника по трем сторонам a, b, c. Данный алгоритм может быть представлен следующим образом:

Шаг 1. Вычислить полупериметр треугольника $p = (a+b+c)/2$.

Шаг 2. Вычислить $S = \sqrt{p(p-a)(p-b)(p-c)}$.

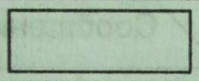
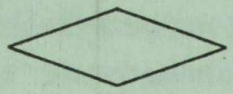
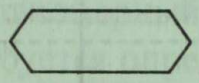
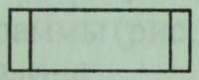
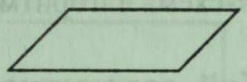
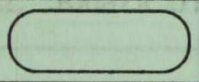
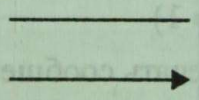
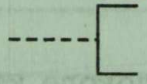
Шаг 3. Напечатать результат S и прекратить вычисления.

При использовании этого способа может быть достигнута любая степень детализации более наглядно, но не строго формализованно.

Блок-схемный способ — это графическое изображение алгоритма, в котором каждый этап процесса обработки данных представляется в виде геометрических фигур (блоков), имеющих определенную конфигурацию в зависимости от характера выполняемых операций. В табл. 2.1 приведены наиболее часто употребляемые блоки.

Таблица 2.1

Геометрические фигуры блок-схем

Название символа	Обозначение	Пояснение
Процесс		Вычислительное действие или последовательность действий
Решение		Блок проверки условия, имеющий один вход и ряд альтернативных выходов, один из которых может быть активизирован после выполнения условий
Модификация		Модификация команды или группы команд с целью воздействия на некоторую последующую функцию
Предопределенный процесс		Процесс, состоящий из одной или нескольких операций (шагов) подпрограммы или модуля
Ввод-вывод		Ввод-вывод информации, при котором отсутствует необходимость в описании фактического носителя данных
Пуск-останов		Начало или конец алгоритма, вход (выход) подпрограммы
Линии потока данных		Отображает поток данных или управления (при необходимости могут быть добавлены стрелки-указатели)
Соединитель		Используется для обрыва линии и продолжения ее в другом месте блок-схемы
Комментарий		Символ используют для добавления комментариев или пояснительных записей

Рассмотрим пример блок-схемы задачи, для которой ранее представлен словесный алгоритм (рис. 2.6).

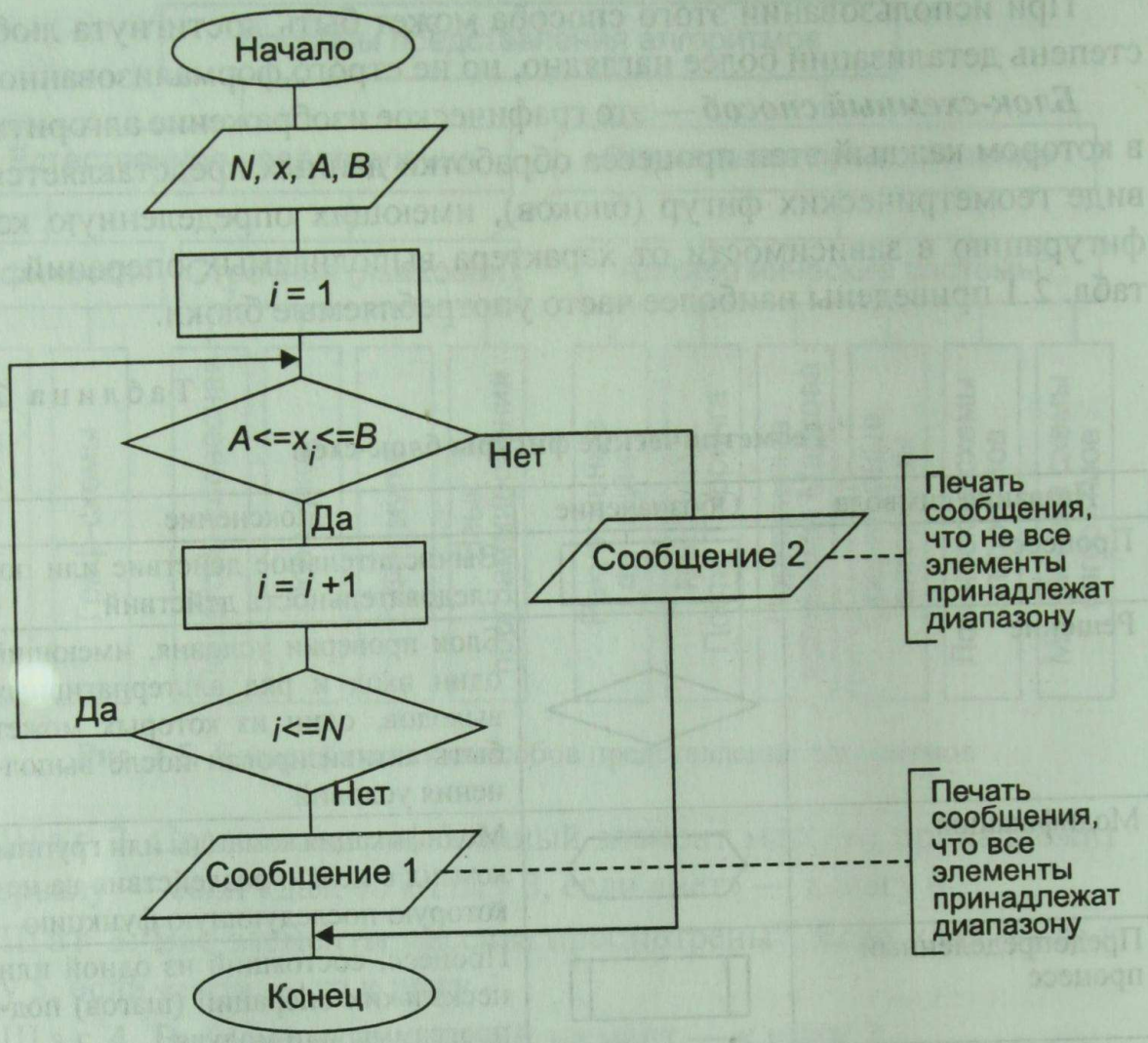


Рис. 2.6. Блок-схема алгоритма

Псевдокод представляет собой совокупность операторов языка программирования и естественного языка. Запись алгоритма в виде псевдокода представлена ниже:

Выбираем первый элемент ($i = 1$)
IF $A > x_i$ или $x_i > B$ **THEN** печать сообщения и выход на конец
ELSE переход к следующему элементу
IF массив не кончился **THEN** переход на проверку интервала
ELSE печать сообщения, что все элементы входят в интервал
Конец

При записи алгоритма на псевдокоде каждое отдельное предложение может начинаться со звездочки (*). Алгоритм строится таким образом, что разбиение продолжается до тех пор, пока каждый шаг алгоритма не станет достаточно понятным.

Пример. Вычислить при заданном x

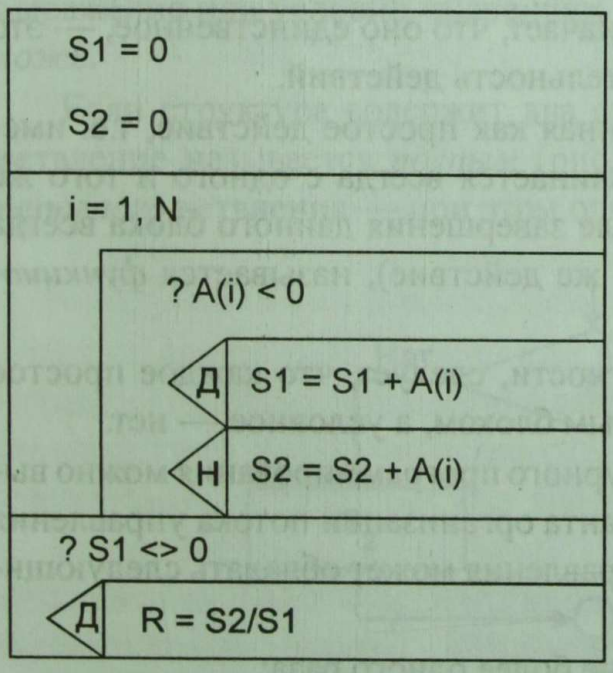
$$y = \begin{cases} x^2, & \text{если} \quad x < 0; \\ x + 1, & \text{если} \quad x \leq 0. \end{cases}$$

Решение.

```
* Ввод (x)
* Если x < 0 то
* * y := x * x
* иначе
* * y := x + 1
* конец проверки условия
* вывод (y)
* конец алгоритма
```

Структурные диаграммы могут использоваться в качестве структурных блок-схем, для показа межмодульных связей, для отображения структур данных и систем обработки данных. Существуют следующие структурные диаграммы: диаграммы Насси — Шнейдермана, Варнье, Джексона, МЭСИД и др.

Пример. Задан одномерный массив из положительных и отрицательных чисел. Требуется определить частное от деления суммы положительных элементов на сумму отрицательных элементов этого массива. Справа от диаграммы (рис. 2.7) приводятся соответствующие операторы языка Turbo Pascal.




```
... S1 := 0;
    S2 := 0;
    FOR I := 1 TO N DO
        IF A[I] < 0 THEN
            S1 := S1 + A[I]
        ELSE
            S2 := S2 + A[I];
    IF S1 <> 0 THEN
        R := S2/S1; ...
```

Рис. 2.7. Диаграмма МЭСИД

2.7. Базовые канонические структуры алгоритмов

Для реализации алгоритмов на ПЭВМ используется алгоритмический язык — набор символов и правил образования и истолкования конструкций из этих символов для записи алгоритмов.

Любую программу можно написать, используя комбинации трех базовых структур:

- следования, или последовательности операторов;
- ветвления, или условного оператора;
- повторения, или оператора цикла.

Программа, составленная из канонических структур, называется ***регулярной программой***, т.е. имеющей один вход и один выход.

Поскольку алгоритм определяет порядок обработки информации, он должен содержать, с одной стороны, действия по обработке, а с другой — порядок их следования, называемый ***потоком управления***.

Рассмотренные выше блоки, связанные с обработкой данных, делятся на *простые* и *условные*. Особенность простого действия в том, что оно имеет один вход и один выход, в отличие от условного, обладающего двумя выходами в зависимости от того, истинным ли окажется условие. Простое действие не означает, что оно единственное, — это может быть некоторая последовательность действий.

Часть алгоритма, организованная как простое действие, т.е. имеющая один вход (выполнение начинается всегда с одного и того же действия) и один выход (т.е. после завершения данного блока всегда начинает выполняться одно и то же действие), называется *функциональным блоком*.

Из этого определения, в частности, следует, что каждое простое действие является функциональным блоком, а условное — нет.

Согласно положениям структурного программирования можно выделить всего три различных варианта организации потока управления действиями алгоритма. Поток управления может обладать следующими свойствами:

- каждый блок выполняется не более одного раза;
- выполняется каждый блок.

Поток управления, в котором выполняются оба эти свойства, называется ***линейным***; в нем несколько функциональных блоков выполняются последовательно. Линейному потоку на языке блок-схем соответствует структура, показанная на рис. 2.8.

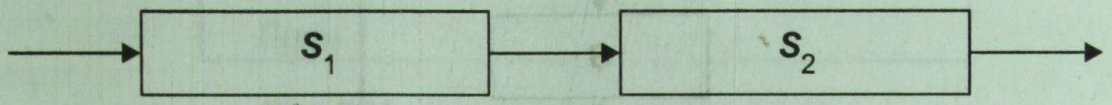


Рис. 2.8. Линейный поток управления

Очевидно несколько блоков, связанных линейным потоком управления, могут быть объединены в один функциональный блок (рис. 2.9).

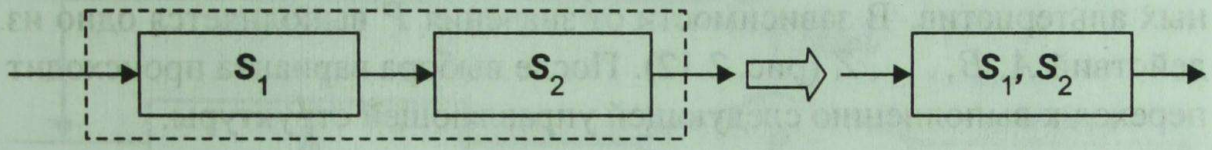


Рис. 2.9. Функциональный блок

Второй тип потока управления называется ***ветвлением*** — он организует выполнение одного из двух функциональных блоков в зависимости от значения проверяемого логического условия. Проверка P представляется *предикатом*, т.е. функцией, задающей логическое выражение или условие, значением которого может быть *истина* или *ложь*.

Если структура содержит два функциональных блока (S_1 и S_2), ветвление называется *полным* (рис. 2.10); возможно существование *неполного* ветвления — при этом один блок отсутствует (рис. 2.11).

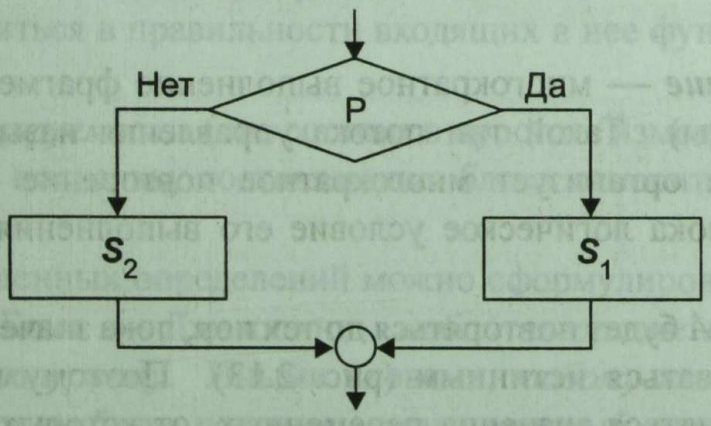


Рис. 2.10. Полное ветвление

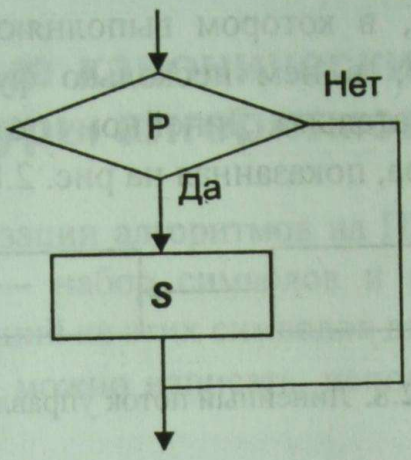


Рис. 2.11. Неполное ветвление

Переключатель — выбор одного варианта из множества возможных альтернатив. В зависимости от значения P выполняется одно из действий $A, B, \ldots, Z$ (рис. 2.12). После выбора варианта происходит переход к выполнению следующей управляющей структуры.

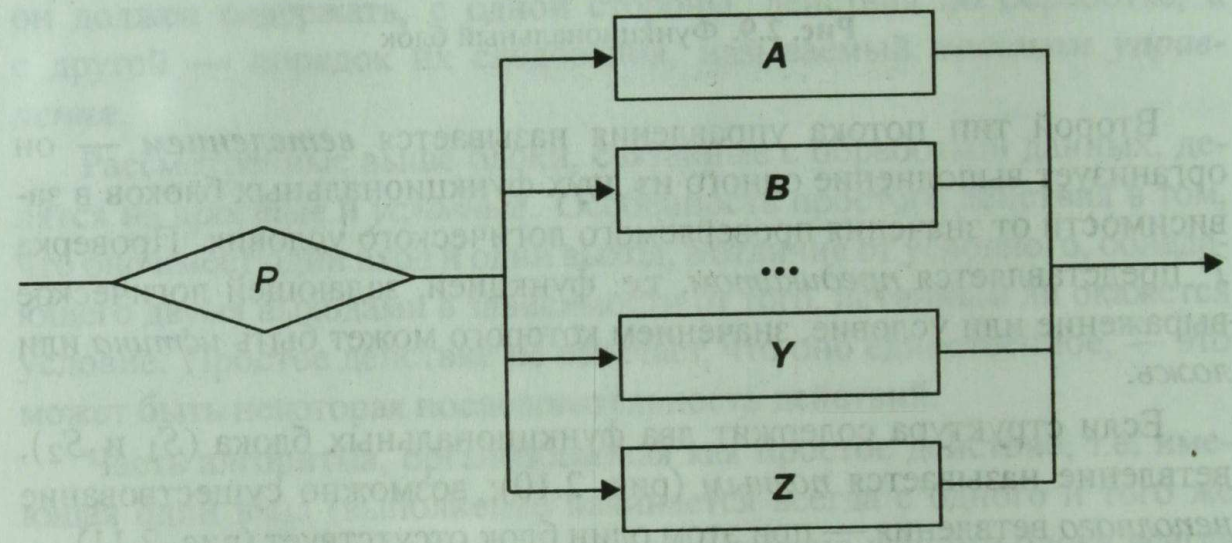


Рис. 2.12. Схема переключателя

Повторение — многократное выполнение фрагментов алгоритма (программы). Такой тип потока управления называется *циклическим* — он организует многократное повторение функционального блока, пока логическое условие его выполнения является истинным.

Действие A будет повторяться до тех пор, пока значение предиката P будет оставаться истинным (рис. 2.13). Поэтому в действии A должны изменяться значения переменных, от которых зависит P, в противном случае произойдет зацикливание.

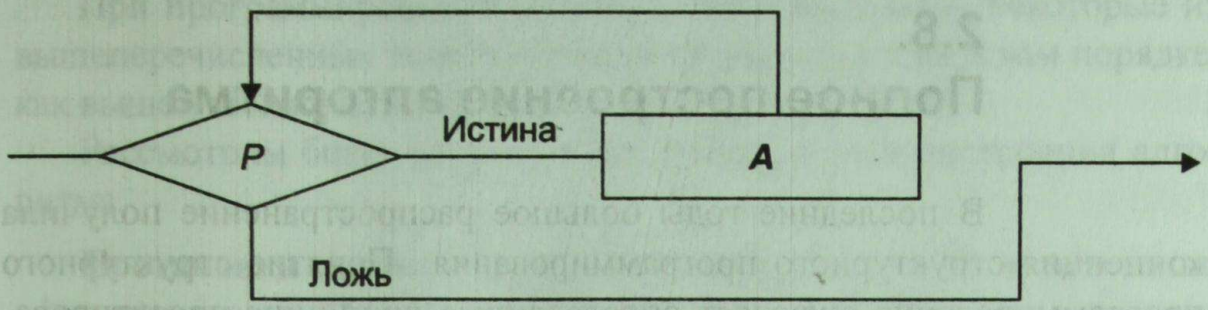


Рис. 2.13. Схема цикла с предусловием

Вычисление предиката P может производиться после выполнения действия A, в этом случае действие A будет выполняться хотя бы один раз (рис. 2.14).

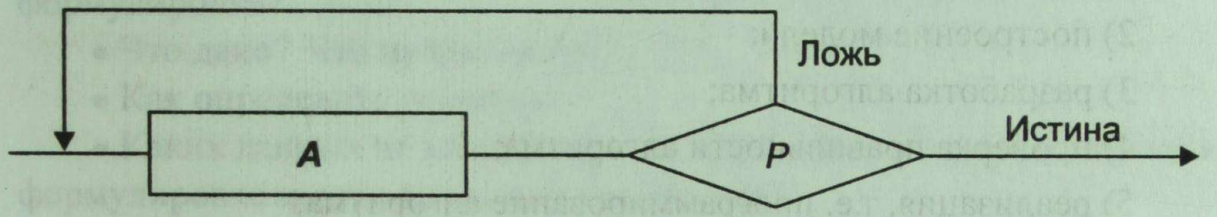


Рис. 2.14. Схема цикла с постусловием

Алгоритм называется ***структурным***, если он представляет собой комбинацию из трех рассмотренных выше структур (они называются *базовыми алгоритмическими структурами*). Безусловно, не все алгоритмы являются структурными. Однако именно структурные алгоритмы обладают рядом замечательных преимуществ по сравнению с неструктурными:

- *понятность* и *простота* восприятия алгоритма (поскольку невелико число исходных структур, которыми он образован);
- *проверяемость* (для проверки любой из основных структур достаточно убедиться в правильности входящих в нее функциональных блоков);
- *модифицируемость* (она состоит в простоте изменения структуры алгоритма, поскольку составляющие блоки относительно независимы).

После введенных определений можно сформулировать структурную теорему Бома — Джакопини: *любой алгоритм может быть сведен к структурному*. Иными словами, *любому неструктурному алгоритму может быть построен эквивалентный ему структурный алгоритм*.

2.8.
Полное построение алгоритма

В последние годы большое распространение получила концепция структурного программирования. Понятие структурного программирования включает определенные принципы проектирования, кодирования, тестирования и документирования программ в соответствии с заранее определенной жесткой дисциплиной.

Полное построение алгоритма предусматривает последовательное выполнение следующих этапов:

1) постановка задачи;

2) построение модели;

3) разработка алгоритма;

4) проверка правильности алгоритма;

5) реализация, т.е. программирование алгоритма;

6) анализ алгоритма и его сложности;

7) проверка (отладка) программы;

8) составление документации.

Не все эти этапы четко различимы между собой, особенно эта различимость делается мало заметной при программировании простых задач (рис. 2.15). При программировании простых задач некоторые этапы могут вообще не выполняться — настолько очевидны их результаты.

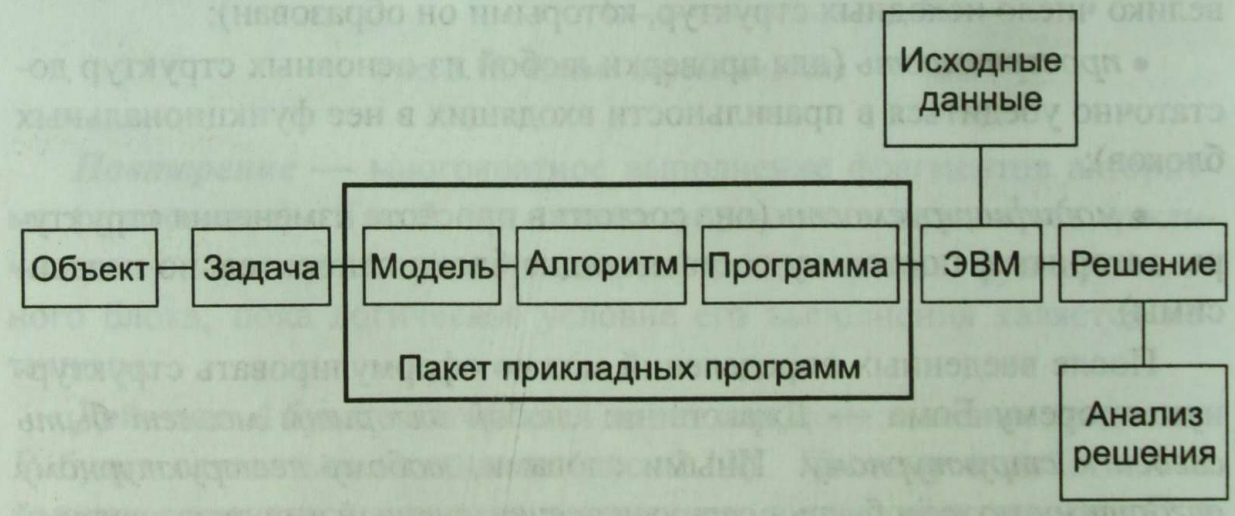


Рис. 2.15. Этапы решения задач на ЭВМ

При программировании сложных, объемных задач некоторые из вышеперечисленных этапов приходится выполнять не в том порядке, как выше указано, или выполнять их не один раз.

Рассмотрим более подробно каждый из этапов построения алгоритма.

Постановка задачи. Прежде чем понять задачу, ее нужно точно сформулировать. Это условие само по себе не является достаточным для понимания задачи, но оно абсолютно необходимо.

Обычно процесс точной формулировки задачи сводится к постановке правильных вопросов. Перечислим некоторые полезные вопросы для плохо сформулированных задач:

- Понятна ли терминология, используемая в предварительной формулировке?
- Что дано? Что нужно найти?
- Как определить решение?
- Каких данных не хватает или, наоборот, все ли перечисленные в формулировке задачи данные используются?
- Какие сделаны допущения?

Возможны и другие вопросы, возникающие в зависимости от конкретной задачи.

Построение модели. Задача четко поставлена, теперь нужно сформулировать для нее математическую модель. Выбор модели существенно влияет на остальные этапы в процессе решения.

Выбор модели — в большей степени искусство, чем наука. Правда, если вы будете встречаться с типовой задачей, то опыт, приобретенный ранее, не потребует от вас особого творческого подхода, и в этом случае как раз будет удобно и целесообразно воспользоваться ранее наработанными правилами. Поэтому изучение удачных моделей — это наилучший способ приобрести опыт в моделировании.

Приступая к разработке модели, следует задать, по крайней мере, два основных вопроса:

1. Какие математические структуры больше всего подходят для задачи?
2. Существуют ли решенные аналогичные задачи?

Второй вопрос, возможно, самый полезный во всей математике. В контексте моделирования он часто дает ответ на первый вопрос. Действительно, большинство решаемых в математике задач, как правило, являются модификациями ранее решенных.

Сначала нужно рассмотреть первый вопрос. Мы должны описать математически, что мы знаем и что хотим найти. На выбор соответствующей структуры будут оказывать влияние следующие факторы:

- ограниченность наших знаний относительно небольшим количеством структур;
- удобство представления;
- простота вычислений;
- полезность различных операций, связанных с рассматриваемой структурой или структурами.

Сделав пробный выбор математической структуры, задачу следует переформулировать в терминах соответствующих математических объектов. Это будет одна из возможных моделей, если мы можем утвердительно ответить на вопросы:

- Вся ли важная информация задачи хорошо описана математическими объектами?
- Существует ли математическая величина, ассоциируемая с искомым результатом?
- Выявили ли мы какие-нибудь полезные отношения между объектами модели?
- Можем ли мы работать с моделью? Удобно ли с ней работать?

Разработка алгоритма. Выбор метода разработки алгоритма зачастую сильно зависит от выбора модели и может в значительной степени повлиять на эффективность алгоритма решения. Два различных алгоритма могут быть правильными, но очень сильно отличаться по эффективности.

Правильность алгоритма. Доказательство правильности алгоритма — это один из наиболее трудных, а иногда и особенно утомительных этапов создания алгоритма. Вероятно, наиболее распространенный прием доказательства правильности программы — это прогон ее на разных тестах. Если выданные программой ответы могут быть подтверждены известными или вычисленными вручную данными, возникает искушение сделать вывод, что программа «работает» правильно. Однако этот метод редко исключает все сомнения; может существовать случай, в котором программа не работает.

Рассмотрим следующую общую методику доказательства правильности алгоритма. Предположим, что алгоритм описан в виде последовательности шагов, допустим, от шага 0 до шага m. Постараемся предложить некое обоснование правомерности для каждого

шага. В частности, может потребоваться формулировка утверждения об условиях, действующих до и после пройденного шага. Затем постараемся предложить доказательство конечности алгоритма, при этом будут проверены *все* подходящие входные данные и получены *все* подходящие выходные данные.

Другой метод доказательства правильности алгоритма, который не имеет специального названия, состоит в следующем. Для каждого цикла, который имеется в программе (алгоритме), вручную (например, на калькуляторе) подсчитываются две контрольные точки. Если контрольные точки совпадают со значениями, выданными программой, можно быть уверенным, что все циклы в программе работают правильно. Почему речь идет о двух контрольных точках? Дело в том, что первое контрольное значение в программе может быть вычислено правильно, а затем в этом цикле будут произведены некоторые некорректные действия, которые приведут к искажению всех последующих результатов. Совпадение второго контрольного значения как раз и подтверждает, что в данном цикле некорректности нет. Таким образом, два контрольных вычисления должны быть сделаны для каждого цикла программы.

Следует подчеркнуть и тот факт, что правильность алгоритма еще ничего не говорит о его эффективности. В этом смысле исчерпывающие алгоритмы, или, как их еще называют, алгоритмы полного перебора, редко бывают хорошими во всех отношениях.

Реализация алгоритма. Как только алгоритм выражен, допустим, в виде последовательности шагов и мы убедились в его правильности, настает черед реализации алгоритма, т.е. написания программы для компьютера.

При этом возникают следующие проблемы.

- Очень часто отдельно взятый шаг алгоритма может быть выражен в форме, которую трудно перевести непосредственно в конструкции языка программирования. Например, один из шагов алгоритма может быть записан в виде, требующем целой подпрограммы для своей реализации.

- Реализация может оказаться трудным процессом потому, что перед тем, как написать программу, необходимо построить целую систему структур данных для представления важных аспектов используемой модели.

Чтобы сделать это, необходимо ответить, например, на такие вопросы:

Каковы основные переменные?

Каких они типов?

Сколько нужно массивов и какой размерности?

Имеет ли смысл пользоваться связными списками?

Какие нужны подпрограммы (возможно, уже записанные в памяти)?

Каким языком программирования пользоваться?

Конкретная реализация может существенно влиять на требования к памяти и на скорость алгоритма.

Заметим, что одно дело — доказать правильность конкретного алгоритма, описанного в словесной форме, другое — доказать, что данная машинная программа, предположительно являющаяся реализацией этого алгоритма, также правильна. Поэтому необходимо очень тщательно следить, чтобы процесс преобразования правильного алгоритма (в словесной форме или форме схемы алгоритма) в программу, написанную на алгоритмическом языке, заслуживал доверия.

2.9. Главные принципы создания эффективных алгоритмов

Каждый, кто занимается разработкой алгоритмов, должен овладеть некоторыми основными методами и понятиями. Перед тем, кто когда-то столкнулся с трудной задачей, вставал вопрос: «С чего начать?». Один из возможных путей — просмотреть свой запас общих алгоритмических методов для того, чтобы проверить, нельзя ли с помощью одного из них сформулировать решение новой задачи. Ну, а если такого запаса нет, то как все-таки разработать хороший алгоритм? Рассмотрим три общих метода решения задач, полезных для разработки алгоритмов.

Первый метод связан со сведением трудной задачи к последовательности более простых задач. Конечно, мы надеемся на то, что более простые задачи легче поддаются обработке, чем первоначальная задача, а также на то, что решение первоначальной задачи может быть получено из решений этих более простых задач. Такая процедура называется ***методом частных целей***.

Этот метод выглядит очень разумно. Но, как и большинство общих методов решения задач или разработки алгоритмов, его не всегда легко перенести на конкретную задачу. Осмысленный выбор более простых задач — скорее, искусство или интуиция, чем наука. Не существует общего набора правил для определения класса задач, которые можно решать с помощью такого подхода. Размышление над любой конкретной задачей начинается с постановки вопросов. Частные цели могут быть установлены, когда получены ответы на следующие вопросы:

1. Можем ли мы решить часть задачи? Можно ли, игнорируя некоторые условия, решить оставшуюся часть задачи?

2. Можем ли мы решить задачу для частных случаев? Можно ли разработать алгоритм, который дает решение, удовлетворяющее всем условиям задачи, но входные данные которого ограничены некоторым подмножеством всех входных данных?

3. Есть ли что-то, относящееся к задаче, что мы недостаточно хорошо поняли? Если попытаться глубже вникнуть в некоторые особенности задачи, сможем ли мы что-то узнать, что поможет нам подойти к решению?

4. Встречались ли мы с похожей задачей, решение которой известно? Можно ли видоизменить ее решение для решения нашей задачи? Возможно ли, что эта задача эквивалентна известной нерешенной задаче?

Второй метод разработки алгоритмов известен как ***метод подъема***. Алгоритм подъема начинается с принятия начального предположения или вычисления начального решения задачи. Затем начинается насколько возможно быстрое движение «вверх» от начального решения по направлению к лучшим решениям. Когда алгоритм достигнет такой точки, из которой больше невозможно двигаться наверх, алгоритм останавливается. К сожалению, мы не можем всегда гарантировать, что окончательное решение, полученное с помощью алгоритма подъема, будет оптимальным. Эта ситуация часто ограничивает применение метода подъема.

Вообще методы подъема являются «грубыми». Они запоминают некоторую цель и стараются сделать все, что могут и где могут, чтобы подойти ближе к цели. Это делает их несколько недальновидными. Недальновидность метода подъема хорошо иллюстрируется следующим примером. Пусть требуется найти максимум функции $y = f(x)$, представленной графиком (рис. 2.16).

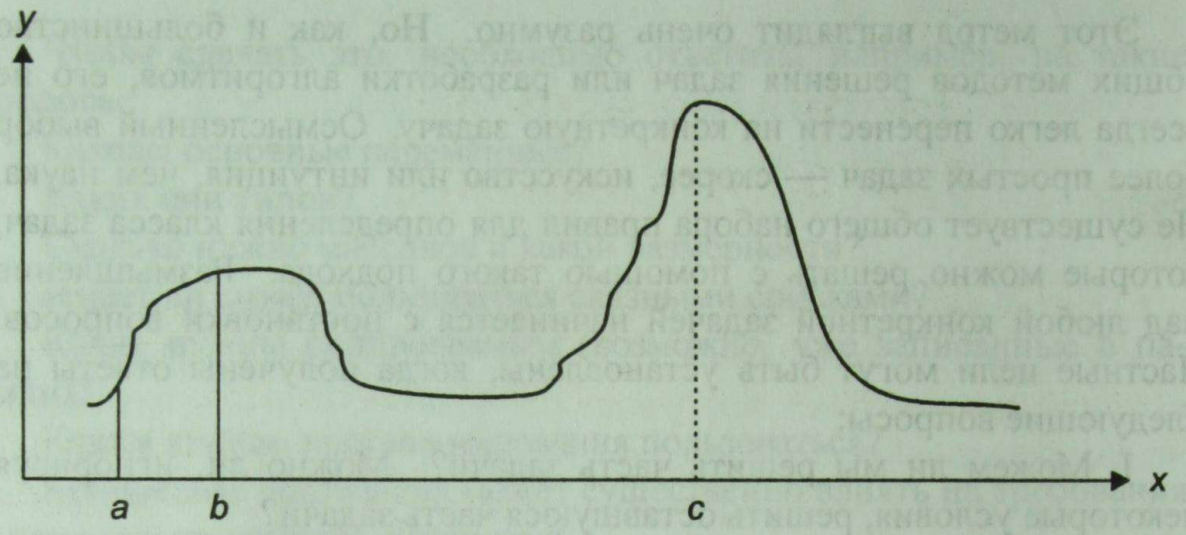


Рис. 2.16. Иллюстрация метода подъема

Если начальное значение аргумента x равно a, то метод подъема даст устремление к ближайшей цели, т.е. к значению функции в точке $x = b$, тогда как подлинный максимум этой функции находится при $x = c$. В данном случае метод подъема находит локальный максимум, но не глобальный. В этом и заключается «грубость» метода подъема.

Третий метод известен как ***отрабатывание назад***, т.е. работа этого алгоритма начинается с цели или решения задачи и затем осуществляется движение к начальной постановке задачи. Затем, если эти действия обратимы, производится движение обратно от постановки задачи к решению.

Контрольные вопросы

1. Дайте определение объекта, класса, системы, модели.
2. Назовите основные типы моделей.
3. Что такое имитационное моделирование?
4. Какие классификации моделей существуют?
5. Укажите основные этапы моделирования.
6. Что такое алгоритм?
7. Перечислите свойства алгоритма.
8. Какие этапы выполняются при полном построении алгоритма?
9. Что такое блок-схема алгоритма?
10. Дайте определение функционального блока.
11. Какой алгоритм называется структурным?
12. Назовите главные принципы, лежащие в основе создания эффективных алгоритмов.

Глава 3
Эволюция языков программирования

Для преобразования текста в последовательность машинных команд необходима промежуточная программа, называемая ***компилятором***. На этапе компиляции производится также распределение данных в оперативном запоминающем устройстве (ОЗУ) ПЭВМ; при этом вместо имен переменных подставляются относительные адреса ячеек, в которых располагаются данные.

Абсолютные адреса данным присваивает операционная система при размещении программы в ОЗУ компьютера перед ее использованием.

Для всех языков высокого уровня общим является то, что они ориентированы не на систему команд той или иной машины, а на систему операторов, характерных для записи определенного класса алгоритмов.

3.1. Классификация языков программирования по функциональному назначению

По функциональному назначению языки программирования высокого уровня делят на ***проблемно-ориентированные*** и ***универсальные***. Первые предназначены для решения специфических задач из некоторой отрасли знаний. Примерами являются: *FORTRAN* — язык решения сложных научных и инженерных задач; *COBOL* — язык для решения экономических и коммерческих задач; *LISP* — язык, используемый в решении задач искусственного интеллекта. К универсальным языкам относятся *PASCAL*, *BASIC*, *C* (*C++*), *Java*, а также современные среды визуального программирования *DELPHI*, *VISUAL BASIC* и др. Эти языки позволяют решить любую задачу, хотя трудоемкость решения конкретной задачи в разных языках будет различаться.

3.2. Классификация языков программирования по парадигме (концепции) и методологии программирования

Другой признак, в соответствии с которым возможна классификация языков программирования, является ***парадигма*** (концепция) программирования (табл. 3.1), т.е. совокупность основополагающих идей и подходов, определяющих модель представления данных и их обработки, а также методологии программирования.

Таблица 3.1

Основные различия в парадигмах

Парадигма программирования	Представление программ и данных	Исполнение программы	Связь частей программы между собой
Процедурное	Программа и данные представляют собой отдельные, не связанные друг с другом элементы	Последовательное выполнение операторов	Возможна только через совместно обрабатываемые данные
Объектно-ориентированное	Данные и методы их обработки инкапсулированы в рамках единого объекта	Последовательность событий и реакций объектов на эти события	Отдельные части программы могут наследовать методы и элементы данных друг у друга
Логическое	Данные и правила их обработки объединены в рамках единого логического структурного образования	Преобразование логического образования в соответствии с логическими правилами	Разбиение программы на отдельные независимые части затруднительно

Известно, что информационные технологии являются одной из наиболее быстро развивающихся областей современной жизни. Новые технологии, проекты, названия и аббревиатуры появляются едва ли не каждый день. В настоящее время в распоряжении программиста имеется довольно обширный спектр языков-инструментов, из которых для любой конкретной задачи можно выбрать тот, что позволит решить ее оптимальным путем.

Язык программирования — набор ключевых слов (словарь) и система правил (грамматических и синтаксических) для конструирования операторов, состоящих из групп или строк чисел, букв, знаков

препинания и других символов, с помощью которых люди могут сообщать компьютеру набор команд. Хронология создания языков программирования представлена в табл. 3.2.

Таблица 3.2

Хронология создания языков программирования

Язык	Год создания	Вид	Автор	География создания
Фортран (*Fortran*)	1954	*A*	Джон Бэкус	Америка
Лисп (*LISP*)	1958	*F*	Джон Маккарти	Америка
Алгол-60 (*Algol 60*)	1960	*A*	Питер Наур+	Международный
Кобол (*Cobol*)	1960	*A*	Группа авторов	Международный
Симула (*Simula*)	1962	*B*	Кристен Нигаард+	Европа
Бейсик (*Basic*)	1963	*A*	Джон Кемени+	Америка
ПЛ/1 (*PL/I*)	1964	*A*	Джордж Радин	Америка
Алгол-68	1968	*A*	Адван Вайнгартен+	Международный
Паскаль (*Pascal*)	1971	*C*	Никлаус Вирт	Европа
Форт (*FORTH*)	1970	*A**	Чарльз Мур	Америка
Си (*C*)	1972	*C**	Деннис Ритчи	Америка
Smalltalk	1972	*B*	Алан Кей	Америка
Пролог (*Prolog*)	1973	*E*	Алан Кольмеро+	Европа
Ада (*Ada*)	1980	*H**	Джин Ишбиа+	Америка
Си++	1984	*H**	Бьорн Страуструп	Америка
Java	1995	*H*	Джеймс Гослинг	Америка
АПЛ (*APL*)	1957	*I*	Кеннет Айверсон	Америка
Снобол (*Snobol*)	1962	*I*	Ральф Грисуолд	Америка
Сетл (*SETL*)	1969	*I*	Джек Шварц	Америка
Scheme	1975	*F*	Гай Стил+	Америка
Icon	1977	*I*	Ральф Грисуолд	Америка
Модула-2 (*Modula-2*)	1979	*D**	Никлаус Вирт	Европа
Оккам (*Occam*)	1982	*G**	Дэвид Мэй+	Европа
Common *Lisp*	1984	*F*	Гай Стил+	Америка
Objective C	1986	*H**	Брэд Кокс	Америка
Оберон (*Oberon*)	1988	*D**	Никлаус Вирт	Европа
Модула-3 (*Modula-3*)	1988	*H**	Билл Калсов+	Америка
Limbo	1996	*D**	Деннис Ритчи	Америка
C#	2000	*H**	Андерс Хейльсберг+	Америка

Условные обозначения:
A — процедурное программирование;
B — объектно-ориентированное программирование;
C — структурное программирование;
D — модульное (компонентное) программирование;
E — логическое (реляционное) программирование;
F — функциональное программирование;
G — параллельное программирование;
H — смесь парадигм: $B + C + D + G$;
I — специализированные языки;
* — поддержка системного программирования;
\+ — язык программирования создан несколькими авторами.

3.3. Классификация языков программирования по типам задач

Языки программирования можно классифицировать по типам задач следующим образом:

• ***Задачи искусственного интеллекта*** — *Lisp*, *Prolog*, *Multilisp*, *Commonlisp*, *Рефал*, *Planner*, *QA4*, *FRL*, *KRL*, *Qlisp*;

• ***Параллельные вычисления*** — *Fun*, *Apl*, *Alfl*, *PARAlfl*, *ML*, *SML*, *PPL/1*, *Hope*, *Miranda*, *Occam*, *PFOR*, *Glypnir*, *Actus*, параллельный *Cobol*, ОВС-ЛЯПИС, ОВС-Мнемокод, ОВС-Алгол, ОВС-Фортран, РА(1), РА(G);

• ***Задачи вычислительной математики и физики*** — *Occam*, *PFOR*, *Glypnir*, *Actus*, параллельный *Cobol*, ОВС-ЛЯПИС, ОВС-Мнемокод, ОВС-Алгол, ОВС-Фортран, РА(1), РА(G);

• ***Разработка интерфейса*** — *Forth*, C, C++, Ассемблер, Макроассемблер, *Simula-67*, OAK, *Smalltalk*, *Java*, РПГ;

• ***Разработка программ-оболочек, разработка систем*** — *Forth*, C, C++, Ассемблер, Макроассемблер, Simula-67, *OAK*, *Smalltalk*, *Java*, РПГ;

• ***Задачи вычислительного характера*** — *Algol*, *Fortran*, *Cobol*, *Ada*, *PL/1*, Фокал, *Basic*, *Pascal*;

• ***Оформление документов, обработка больших текстовых файлов, организация виртуальных трехмерных интерфейсов в Интернете, разработка баз данных*** — HTML, *Perl*, *Tcl/Tk*, *VRML*, *SQL*, *PL/SCL*, *Informix 4GL*, *Natural*, *DDL*, *DSDL*, *SEQUEL*, *QBE*, *ISBL*.

* * *

В настоящее время существуют программно-аппаратные комплексы, позволяющие организовать параллельное выполнение различных частей одного и того же вычислительного процесса.

В основе функционального программирования лежит представление программы в виде математических функций. Существуют языки с ленивой и с энергичной семантикой: в языках с энергичной семантикой вычисления производятся в том же месте, где они описаны, а в случае ленивой семантики вычисление производится только тогда, когда оно необходимо. Программы на языках логического программи-

рования выражены как формулы математической логики, а компилятор получает следствия из них.

В последнее время в связи с развитием интернет-технологий получили распространение скриптовые языки. Характерными особенностями данных языков являются их интерпретируемость, простота синтаксиса и легкая расширяемость. Они идеально подходят для использования в часто изменяемых программах или в случаях, когда для выполнения операторов языка затрачивается время, несопоставимое с временем их разбора.

Контрольные вопросы

1. Дайте определение алгоритмического языка.
2. Что такое компилятор?
3. Классифицируйте языки программирования.
4. Что такое парадигма?
5. Назовите первый алгоритмический язык.
6. Перечислите парадигмы языков программирования.
7. В чем заключена основная особенность объектно-ориентированных языков?
8. Назовите языки для обработки данных.
9. Назовите Pascal- и C-подобные языки.
10. Какие алгоритмические языки используются для решения задач математики и физики? Для решения задач вычислительного характера? Для решения задач искусственного интеллекта?
11. Чем различаются языки с ленивой и энергичной семантикой?
12. Назовите характерные особенности скриптовых языков.

Глава 4
Функция сложности алгоритма

Существует ряд важных причин для анализа алгоритмов. Одной из них является необходимость получения оценок или границ для объема памяти или времени работы, которое потребуется алгоритму для успешной обработки конкретных данных.

Для оценки качества алгоритма вводится понятие ***сложность алгоритма***, или обратное понятие — ***эффективность алгоритма***. Чем

большее время и объем памяти требуются для реализации алгоритма, тем больше его сложность и соответственно ниже эффективность. Сложность алгоритма делится на временну́ю и емкостную. Временна́я сложность — это критерий, характеризующий временны́е затраты на реализацию алгоритма. Емкостная сложность — критерий, характеризующий затраты памяти на те же цели. В зависимости от конкретной формы этих критериев сложность алгоритма в свою очередь подразделяется на *практическую* и *теоретическую*. Практическая временна́я сложность обычно оценивается во временных единицах (секунды, милисекунды, количество временных тактов процессора, количество выполнения циклов и т.п.). Практическая емкостная сложность выражается, как правило, в битах, байтах, словах и т.п. Способы представления теоретической сложности алгоритма будут рассмотрены далее.

Перечислим основные факторы, от которых может зависеть сложность алгоритма:

- быстродействие компьютера и его емкостные ресурсы (в первую очередь — объем оперативной памяти). В самом деле, чем ниже тактовая частота процессора и меньше объем оперативного запоминающего устройства, тем медленнее выполняются арифметические и логические операции, тем чаще (для больших задач) приходится обращаться к медленно действующей внешней памяти, и, следовательно, больше времени уходит на реализацию алгоритма;
- выбранный язык программирования. Задача, запрограммированная, например, на языке Ассемблера, в общем случае решится быстрее, чем по тому же самому алгоритму, но запрограммированному на языке более высокого уровня, например на Паскале;
- выбранный математический метод формулирования задачи;
- искусство и опыт программиста. В общем случае по одному и тому же алгоритму опытный программист напишет более эффективно работающую программу, чем его начинающий коллега.

Для решения многих проблем существует множество различных алгоритмов. Какой из них выбрать для решения конкретной задачи? Этот вопрос очень тщательно прорабатывается в программировании. Эффективность программы является очень важной ее характеристикой. Эффективность программы имеет две составляющие: память (или пространство) и время. Пространственная эффективность измеряется количеством памяти, требуемой для выполнения программы.

Компьютеры обладают ограниченным объемом памяти. Если две программы реализуют идентичные функции, то та, которая использует меньший объем памяти, характеризуется большей пространственной эффективностью. Иногда память становится доминирующим фактором в оценке эффективности программ. Однако в последние годы в связи с быстрым ее удешевлением эта составляющая эффективности постепенно теряет свое значение. Временна́я эффективность программы определяется вре́менем, необходимым для ее выполнения.

Лучший способ сравнения эффективностей алгоритмов состоит в сопоставлении их порядков сложности. Этот метод применим как к временно́й, так и пространственной сложности. Порядок сложности алгоритма выражает его эффективность обычно через количество обрабатываемых данных. Например, некоторый алгоритм может существенно зависеть от размера обрабатываемого массива. Если, например, время обработки удваивается с удвоением размера массива, то порядок временно́й сложности алгоритма определяется как размер массива. Порядок алгоритма — это функция, доминирующая над точным выражением временно́й сложности. Функция $f(n)$ имеет порядок $O(g(n))$, если имеется константа k и счетчик n_0, такие, что $f(n) \leq kg(n)$ для $n > n_0$. Действительное время есть функция, зависящая от длины массива.

Например, известно, что точное время обработки массива определяется из уравнения

Действительное время (Длина массива) =
$= \text{Длина массива}^2 + 5 \cdot \text{Длина массива} + 100.$

Грубое значение определяется вспомогательной функцией:

Оценка времени (Длина массива) $= 1{,}1 \cdot \text{Длина массива}^2.$

Функция сложности O выражает относительную скорость алгоритма в зависимости от некоторой переменной (или переменных). Существуют три важных правила для определения функции сложности:

1. $O(kf) = O(f)$.
2. $O(fg) = O(f)O(g)$ или $O(f/g) = O(f)/O(g)$.
3. $O(f + g)$ равна доминанте $O(f)$ и $O(g)$.

Здесь k обозначает константу, а f и g — функции.

Первое правило декларирует, что постоянные множители не имеют значения для определения порядка сложности, например:

$$O(1{,}5N) = O(N).$$

Из второго правила следует, что порядок сложности произведения двух функций равен произведению их сложностей, например::

$$O(17N \cdot N) = O(17N)\,O(N) = O(N)\,O(N) = O(N \cdot N) = O(N^2).$$

Из третьего правила следует, что порядок сложности суммы функций определяется как порядок доминанты первого и второго слагаемых, т.е. выбирается наибольший порядок, например:

$$O(N^5 + N^2) = O(N^5).$$

4.1. Виды функции сложности алгоритмов

Функция сложности $O(1)$. В алгоритмах константной сложности большинство операций в программе выполняется один или несколько раз. Любой алгоритм, всегда требующий независимо от размера данных одного и того же времени, имеет константную сложность.

Функция сложности $O(N)$. Время работы программы линейно, обычно когда каждый элемент входных данных требуется обработать лишь линейное число раз.

Функция сложности $O(N^2)$, $O(N^3)$, $O(N^a)$ — полиномиальная функция сложности.

Функция сложности $O(\log_2 N)$, $O(N \log_2 N)$. Такое время работы имеют те алгоритмы, которые делят большую проблему на множество небольших, а затем, решив их, объединяют решения.

Функция сложности $O(2^N)$. Экспоненциальная сложность. Такие алгоритмы чаще всего возникают в результате подхода, именуемого «метод грубой силы».

4.2. Временна́я функция сложности

Программист должен уметь проводить анализ алгоритмов и определять их сложность. Временна́я сложность алгоритма может быть посчитана исходя из анализа его управляющих структур.

Алгоритмы без циклов и рекурсивных вызовов имеют константную сложность. Если нет рекурсии и циклов, то все управляющие структуры могут быть сведены к структурам константной сложности. Следовательно, и весь алгоритм также характеризуется константной сложностью. Определение сложности алгоритма в основном сводится к анализу циклов и рекурсивных вызовов.

Например, рассмотрим алгоритм обработки элементов массива:

```
For i:=1 to N do
Begin
   ...
End;
```

Сложность этого алгоритма $O(N)$, так как тело цикла выполняется N раз, и сложность тела цикла равна $O(1)$. Если один цикл вложен в другой и оба цикла зависят от размера одной и той же переменной, то вся конструкция характеризуется квадратичной сложностью:

```
For i:=1 to N do
For j:=1 to N do
Begin  }
   ...  } тело цикла
End;   }
```

Сложность этой программы $O(N^2)$.

4.3. Анализ функции сложности по программе

Существуют два способа анализа сложности алгоритма: восходящий (от внутренних управляющих структур к внешним) и нисходящий (от внешних структур к внутренним).

Как правило, около 90% времени работы программы требует выполнение повторений и только 10% составляют непосредственно

вычисления. Анализ сложности программ показывает, на какие фрагменты выпадают эти 90% — это циклы наибольшей глубины вложенности. Повторения могут быть организованы в виде вложенных циклов или вложенной рекурсии. Эта информация может использоваться программистом для построения более эффективной программы следующим образом. Прежде всего можно попытаться сократить глубину вложенности повторений. Затем следует рассмотреть возможность сокращения количества операторов в циклах с наибольшей глубиной вложенности. Если 90% времени выполнения составляет выполнение внутренних циклов, то тридцатипроцентное сокращение этих небольших секций приводит к 27-процентному снижению времени выполнения всей программы.

Оценим сложность программы «Тройки Пифагора»:

```
D{ A{ Writeln(`Введите ограничение`);
      Readln(N);
      small:=1;
   E{ While small < N do
      begin
      B{ next:=small;
         G{ While next <= N do
            begin
            C{ last:=next;
               I{ While last <=N do
                  begin
                  F{ if (last<=small*2) and (next<=small*2) and
                     (last*last=small*small+next*next) then
                     begin
                     H{ writeln(small);
                        writeln(next);
                        writeln(last);
                     end;
                  J{ inc(last);
                  end;
            K{ inc(next);
            end;
      L{ inc(small);
      end;
```

$$O(H)=O(1)+O(1)+O(1)=O(1);$$

$$O(I)=O(N)\cdot(O(F)+O(J))=O(N)\cdot O(\text{доминанты условия})=O(N);$$

$$O(G)=O(N)\cdot(O(C)+O(I)+O(K))=O(N)\cdot(O(1)+O(N)+O(1))=O(N^2);$$

$$O(E)=O(N)\cdot(O(B)+O(G)+O(L))=O(N)\cdot O(N^2)=O(N^3);$$

$$O(D)=O(A)+O(E)=O(1)+O(N^3)=O(N^3).$$

Сложность данного алгоритма $O(N^3)$.

4.4. Оценка алгоритма бинарного поиска

Произведем оценку алгоритма бинарного поиска в массиве по приведенной программе:

```
function search(low, high, key: integer): integer;
var
    mid, data: integer;
begin
    while low <= high do
    begin
        mid:=(low+high) div 2;
        data:=a[mid];
        if key=data then
            search:=mid
        else
            if key < data then
                high := mid-1
            else
                low:=mid+1;
    end;
    search:=-1;
end;
```

Первая итерация цикла имеет дело со всем списком. Каждая последующая итерация делит пополам размер массива. Так, размерами списка для алгоритма являются

$$n,\quad \frac{n}{2^1},\quad \frac{n}{2^2},\quad \frac{n}{2^3},\quad \frac{n}{2^4},\quad \ldots,\quad \frac{n}{2^m}.$$

В конце концов будет такое целое m, что $\frac{n}{2^m} < 2$ или $n < 2^{m+1}$. Так как m — это первое целое, для которого $\frac{n}{2^m} < 2$, то должно быть верно $\frac{n}{2^{m-1}} \geq 2$ или $2^m \leq n$. Из этого следует, что $2^m \leq n < 2^{m+1}$. Возьмем логарифм каждой части неравенства и получим $m \leq \log_2 n < m + 1$. Отсюда $O(\log_2 n)$.

4.5. Теоретическая и практическая функции сложности

Для оценки эффективности алгоритмов используется функция сложности алгоритма, которая обозначается заглавной буквой «О», в круглых скобках записывается аргумент. Например, функция сложности $O(n^2)$ читается как функция сложности порядка n^2. Функция сложности алгоритма — это функция, которая определяет количество сравнений, перестановок, а также временные и ресурсные затраты на реализацию алгоритма (рис. 4.1).

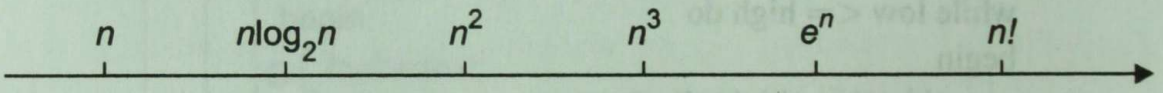


Рис. 4.1. Ряд значений функции сложности

Функция $f(n)$ в ряде случаев может иметь достаточно сложную аналитическую форму. Поскольку для временно́й теоретической сложности большее значение имеет не столько вид функции, сколько порядок ее роста, то во многих математических дисциплинах, в том числе и в теории алгоритмов, функцию $f(n)$ определяют как $O(g(n))$ и говорят, что она порядка $g(n)$ для больших n, если

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{const} \neq 0,$$

где $f(n)$ и $g(n)$ — экспериментальная и теоретическая функции сложности. При этом если $f(n) = O(g(n))$, то предел отношения равен константе и тогда функция $f(n)$ имеет порядок $O(g(n))$ («О большое»).

Пример 1. Полином $f(n) = 2n^5 + 6n^4 + 6n^2 + 18$ имеет $O(n^5)$, так как

$$\lim_{n\to\infty} \frac{2n^5 + 6n^4 + 6n^2 + 18}{n^5} = 2.$$

Пример 2. Определить функцию сложности алгоритма по результатам эксперимента:

N	Количество сравнений
6	54

Решение. Вначале найдем экспериментальную функцию сложности $O_э$.

Экспериментальная функция сложности алгоритма принимает следующий ряд значений:

$$an, \quad an\log_2 n, \quad an^2, \quad an^3, \quad ae^n, \quad an!$$

Для правильно подобранной экспериментальной функции $a = 1$, однако на практике допускается $1 \le a \le 2$;

а) допустим, $O_э = an$, тогда $an = 54$, $6a = 54$, $a = \frac{54}{6}$ (не удовлетворяет условию $1 \le a \le 2$).

При $a = 1$ значение экспериментальной функции совпадает со значением теоретической функции сложности;

б) допустим, $O_э = an\log_2 n$, тогда $an\log_2 n = 54$, $a6\log_2 6 = 54$, $a = \frac{54}{6\log_2 6} = \frac{9}{\log_2 6}$ ($a > 2$, не удовлетворяет условию);

в) допустим, $O_э = an^2$, тогда $an^2 = 54$, $a \cdot 36 = 54$, $a = \frac{54}{36} = 1{,}5$ ($a < 2$ — удовлетворяет условию).

Таким образом, экспериментальная функция сложности имеет вид $O_э(1{,}5n^2)$.

Найдем теоретическую функцию сложности:

$$\lim_{n\to\infty} \frac{O_э(n)}{O_т(n)} = \text{const} \neq 0, \qquad \lim_{n\to\infty} \frac{1{,}5n^2}{X} = \text{const} \neq 0,$$

$$\lim_{n\to\infty} \frac{1{,}5n^2}{n^2} = \text{const} \neq 0.$$

Отсюда теоретическая функция сложности — $O(n^2)$.

Контрольные вопросы

1. Дайте определение функции сложности.
2. Укажите виды функций сложности алгоритмов.
3. Что включает понятие сложности алгоритма?
4. Укажите правила для определения функции сложности.
5. Какие виды функции сложности существуют?
6. Каким образом определяется временная функция сложности?
7. Назовите способы анализа функции сложности по программе.
8. Укажите уравнение, по которому определяется функция сложности.
9. Какие значения принимает экспериментальная функция сложности?
10. Каков смысл коэффициента a в экспериментальной функции сложности?

ЧАСТЬ 2

АЛГОРИТМЫ ОБРАБОТКИ СТРУКТУР ДАННЫХ

Глава 5
Методы сортировки

Упорядочение элементов множества в возрастающем или убывающем порядке называется ***сортировкой***.

С упорядоченными элементами проще работать, чем с произвольно расположенными: легче найти необходимые элементы, исключить, вставить новые. Сортировка применяется при трансляции программ, при организации наборов данных на внешних носителях, при создании библиотек, каталогов, баз данных и т.д.

Алгоритмы сортировки можно разбить на следующие группы (рис. 5.1).

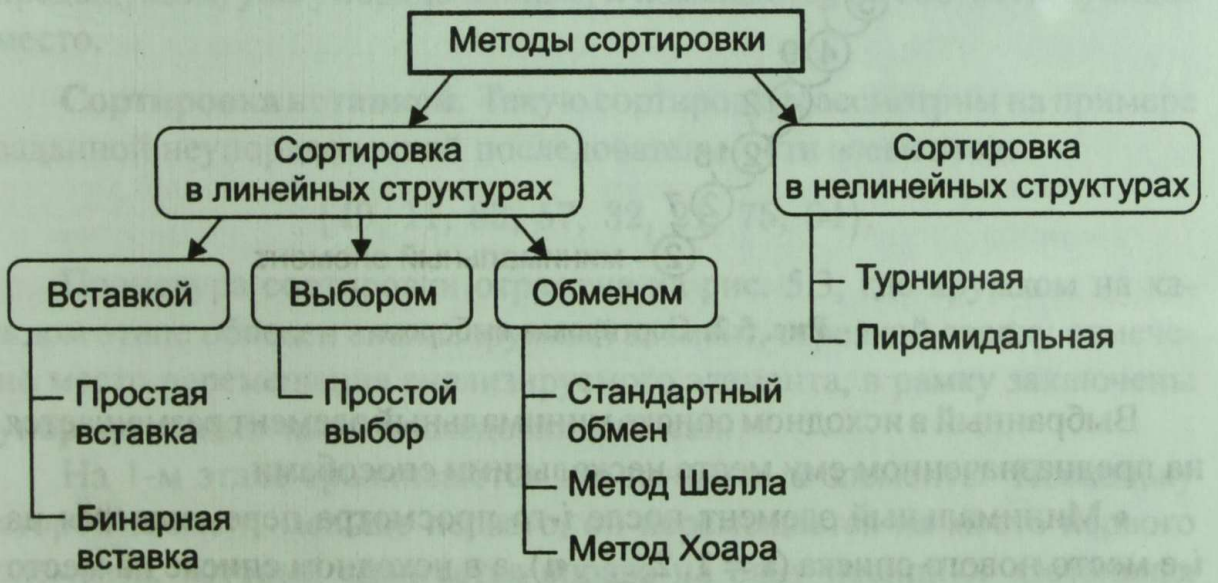


Рис. 5.1. Алгоритмы сортировки

Обычно сортируемые элементы множества называют ***записями*** и обозначают через $k_1, k_2, \ldots, k_n$.

5.1. Сортировка выбором

Сортировка выбором состоит в том, что сначала в неупорядоченном списке выбирается и отделяется от остальных наименьший элемент. После этого исходный список оказывается измененным. Измененный список принимается за исходный и процесс продолжается до тех пор, пока все элементы не будут выбраны. Очевидно, что выбранные элементы образуют упорядоченный список.

Например, требуется найти минимальный элемент списка

$$\{5,\ 11,\ 6,\ 4,\ 9,\ 2,\ 15,\ 7\}.$$

Процесс выбора показан на рис. 5.2, где в каждой строчке выписаны сравниваемые пары. Выбираемые элементы с меньшим весом обведены кружком. Нетрудно видеть, что число сравнений соответствует на рисунке числу строк, а число перемещений — количеству изменений выбранного элемента.

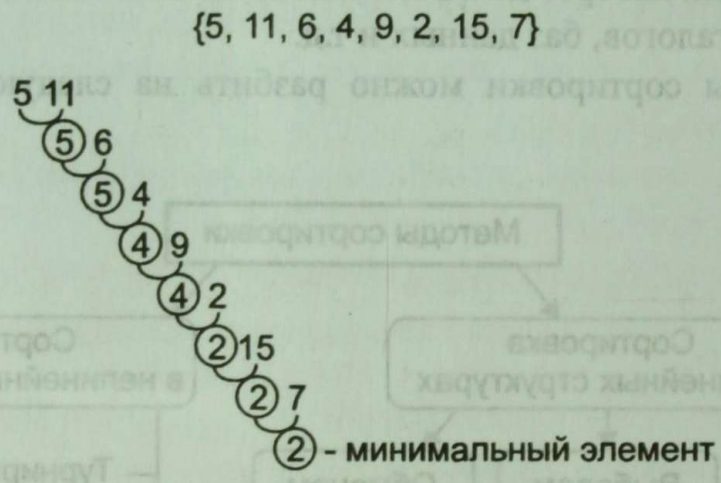


Рис. 5.2. Сортировка выбором

Выбранный в исходном списке минимальный элемент размещается на предназначенном ему месте несколькими способами.

• Минимальный элемент после i-го просмотра перемещается на i-е место нового списка ($i = 1, 2, \ldots, n$), а в исходном списке на место выбранного элемента записывается какое-то очень большое число, превосходящее по величине любой элемент списка. При этом длина заданного списка остается постоянной. Измененный таким образом список можно принимать за исходный.

• Минимальный элемент записывается на i-е место исходного списка ($i = 1, 2, \ldots, n$), а элемент с i-го места — на место выбранного. При этом очевидно, что уже упорядоченные элементы (а они будут расположены, начиная с первого места) исключаются из дальнейшей сортировки, поэтому длина каждого последующего списка (списка, участвующего в каждом последующем просмотре) должна быть на один элемент меньше предыдущего.

• Выбранный минимальный элемент, как и в предыдущем случае, перемещается на i-е место заданного списка, а чтобы это i-е место освободилось для записи очередного минимального элемента, левая от выбранного элемента часть списка перемещается вправо на одну позицию так, чтобы заполнилось место, занимаемое до этого выбранным элементом.

Сложность метода сортировки выбором порядка $O(n^2)$.

5.2. Сортировка вставкой и сортировка слиянием

В этом методе из неупорядоченной последовательности элементов выбирается поочередно каждый элемент, сравнивается с предыдущим, уже упорядоченным, и помещается на соответствующее место.

Сортировка вставкой. Такую сортировку рассмотрим на примере заданной неупорядоченной последовательности элементов:

$$\{40, 11, 83, 57, 32, 21, 75, 64\}.$$

Процедура сортировки отражена на рис. 5.3, где кружком на каждом этапе обведен анализируемый элемент, стрелкой сверху отмечено место перемещения анализируемого элемента, в рамку заключены упорядоченные части последовательности.

На 1-м этапе сравниваются два начальных элемента. Поскольку второй элемент меньше первого, он перемещается на место первого элемента, который сдвигается вправо на одну позицию. Остальная часть последовательности остается без изменения.

На 2-м этапе из неупорядоченной последовательности выбирается элемент и сравнивается с двумя упорядоченными ранее элементами. Так как он больше предыдущих, то остается на месте. Затем

анализируются четвертый, пятый и последующие элементы до тех пор, пока весь список не будет упорядоченным, что имеет место на последнем 7-м этапе.

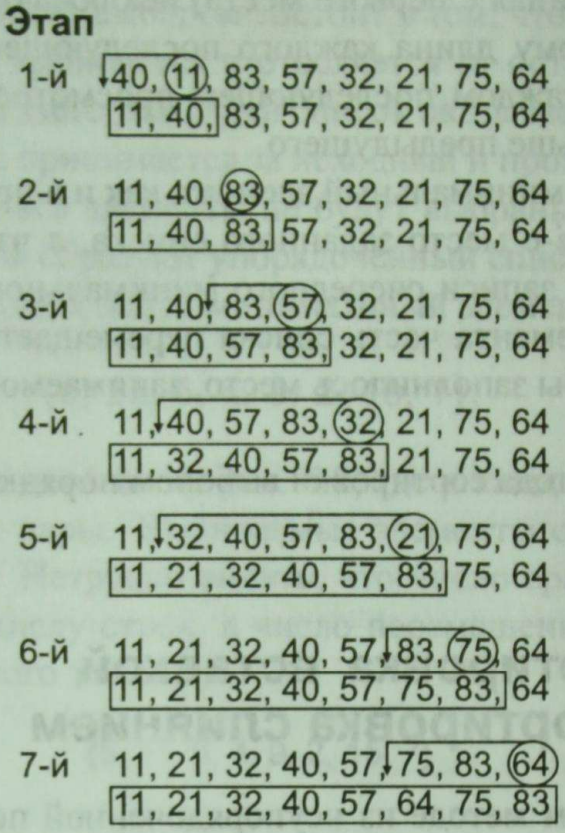


Рис. 5.3. Сортировка вставкой

Сортировка слиянием. Разновидностью сортировки вставкой является метод фон Неймана, или сортировка слиянием. Идея метода состоит в следующем: сначала анализируются первые элементы обоих массивов. Меньший элемент переписывается в новый массив. Оставшийся элемент последовательно сравнивается с элементами из другого массива. В новый массив после каждого сравнения попадает меньший элемент. Процесс продолжается до исчерпания элементов одного из массивов. Затем остаток другого массива дописывается в новый массив. Полученный новый массив упорядочен таким же образом, как исходные.

Пусть имеются два отсортированных в порядке возрастания массива $p[1], p[2], \ldots, p[n]$ и $q[1], q[2], \ldots, q[n]$ и имеется пустой массив $r[1], r[2], \ldots, r[2n]$, который мы хотим заполнить значениями массивов p и q в порядке возрастания. Для слияния выполняются следующие действия: сравниваются $p[1]$ и $q[1]$ и меньшее из значений записывается в $r[1]$. Предположим, что это значение $p[1]$. Тогда $p[2]$ сравнивается

с $q[1]$, и меньшее из значений заносится в $r[2]$. Предположим, что это значение $q[1]$. Тогда на следующем шаге сравниваются значения $p[2]$ и $q[2]$ и т.д., пока мы не достигнем границ одного из массивов. Тогда остаток другого массива просто дописывается в «хвост» массива r. Пример слияния двух массивов показан на рис. 5.4.

Сложность метода сортировки вставкой порядка $O(n^2)$.

5.3. Сортировка обменом и шейкерная сортировка

Сортировка обменом. Это метод, в котором элементы списка последовательно сравниваются между собой и меняются местами в том случае, если предшествующий элемент больше последующего. Требуется, например, провести сортировку списка методом стандартного обмена, или методом «пузырька»:

$$\{40,\ 11,\ 83,\ 57,\ 32,\ 21,\ 75,\ 64\}.$$

Обозначим квадратными скобками со стрелками $\overset{\curvearrowleft}{\llcorner\!\lrcorner}$ обмениваемые элементы, а $\llcorner\!\lrcorner$ — сравниваемые элементы. Первый этап сортировки показан на рис. 5.5, а второй этап — на рис. 5.6.

Нетрудно видеть, что после каждого просмотра списка все элементы, начиная с последнего, занимают свои окончательные позиции, поэтому их не следует проверять при следующих просмотрах. Каждый последующий просмотр исключает очередную позицию с найденным максимальным элементом, тем самым укорачивая список.

После первого просмотра в последней позиции оказался больший элемент, равный 83 (исключаем его из дальнейшего рассмотрения). Второй просмотр выявляет максимальный элемент, равный 75 (см. рис. 5.6).

Процесс сортировки продолжается до тех пор, пока не будут сформированы все элементы конечного списка либо не выполнится условие Айверсона.

Условие Айверсона: если в ходе сортировки при сравнении элементов не было сделано ни одной перестановки, то множество считается упорядоченным (условие Айверсона выполняется только при шаге $d = 1$).

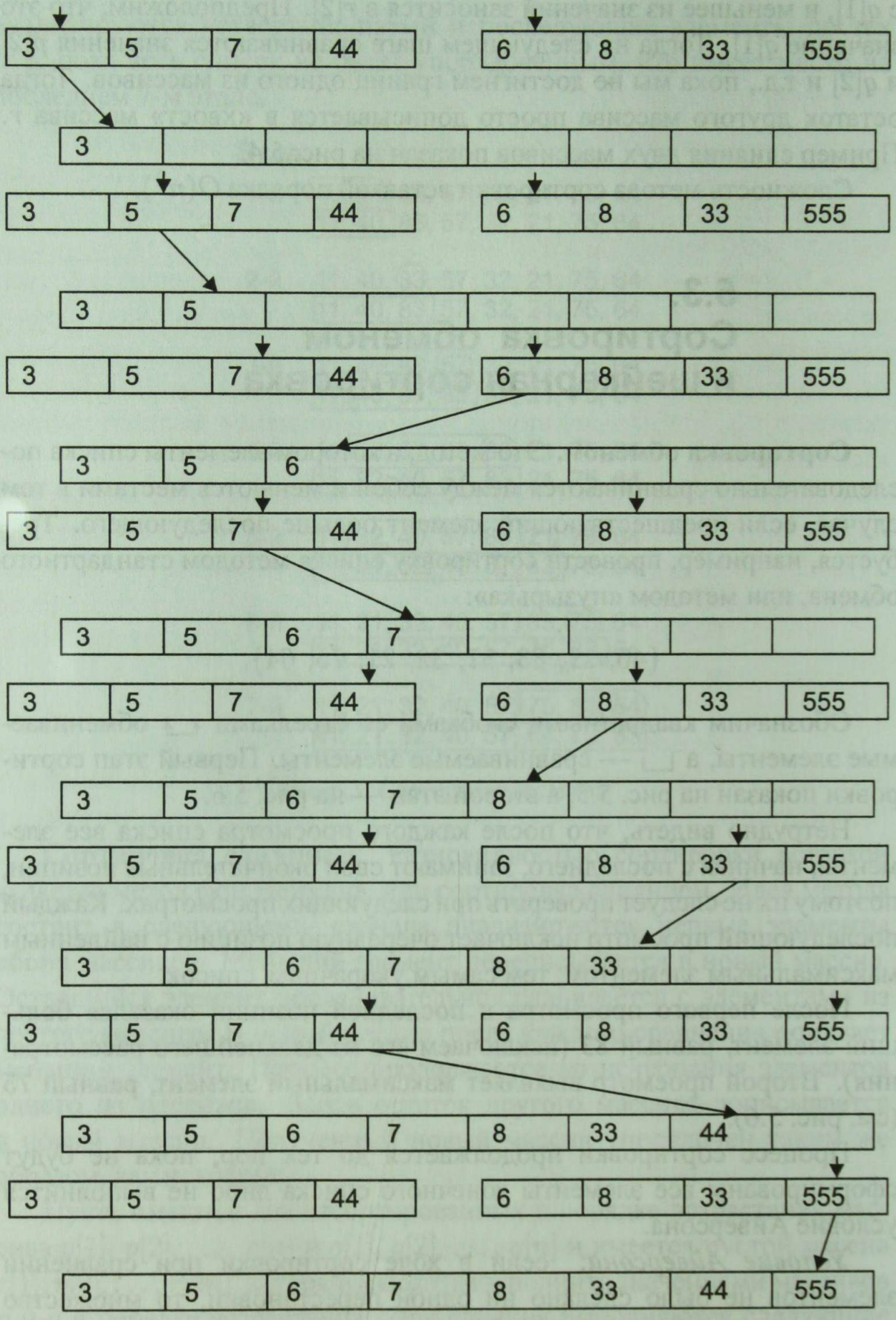


Рис. 5.4. Сортировка слиянием

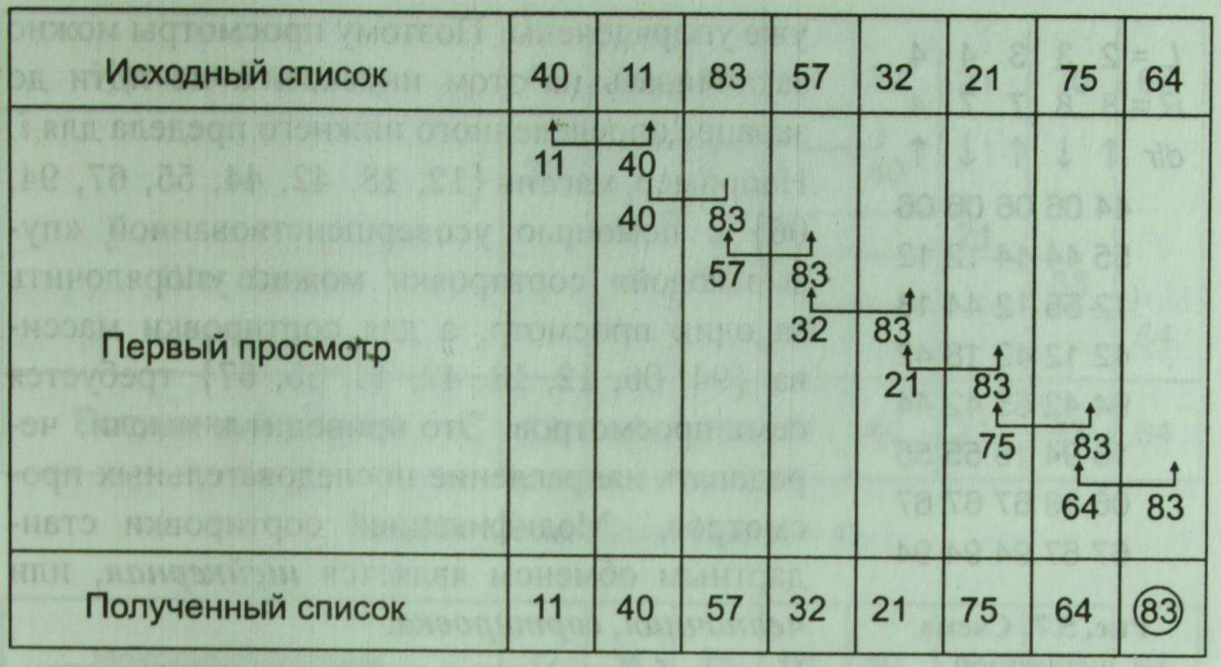


Рис. 5.5. Сортировка обменом (первый просмотр)

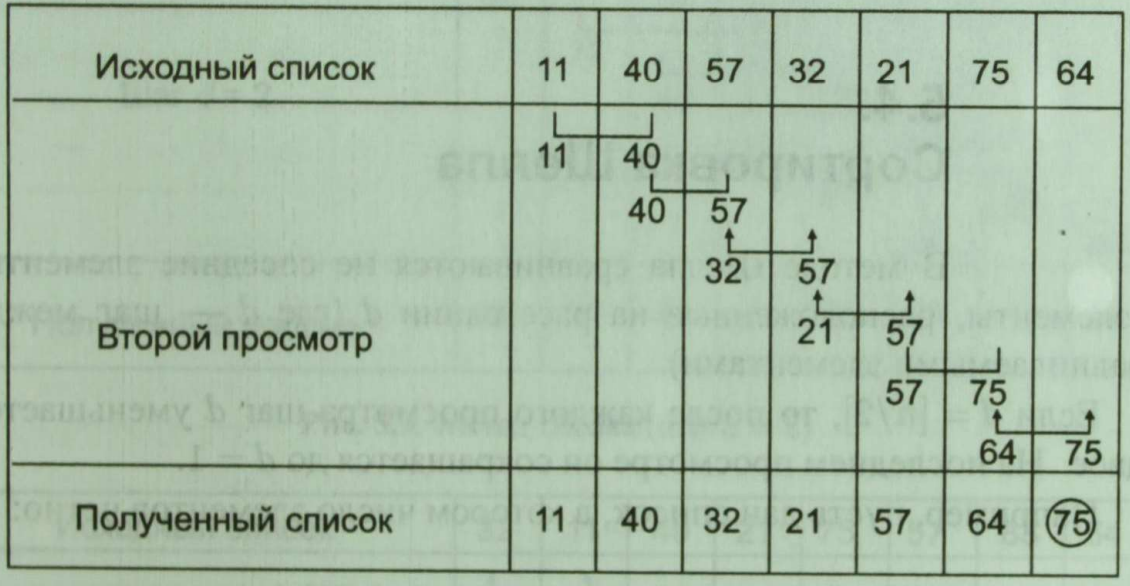


Рис. 5.6. Сортировка обменом (второй просмотр)

Сложность метода стандартного обмена $O(n^2)$.

Шейкерная сортировка. Очевидный прием улучшения алгоритма стандартного обмена — запоминать, были или не были перестановки в процессе некоторого прохода. Если в последнем проходе перестановок не было, то алгоритм можно заканчивать. Это улучшение, однако, можно опять же улучшить, если запоминать не только сам факт, что обмен имел место, но и положение (индекс) последнего обмена. Ясно, что все пары соседних элементов выше этого индекса k

L =	2	3	3	4	4
R =	8	8	7	7	4
dir	↑	↓	↑	↓	↑
	44	06	06	06	06
	55	44	44	12	12
	12	55	12	44	18
	42	12	42	18	42
	94	42	55	42	44
	18	94	18	55	55
	06	18	67	67	67
	67	67	94	94	94

Рис. 5.7. Схема шейкерной сортировки

уже упорядочены. Поэтому просмотры можно заканчивать на этом индексе, а не идти до заранее определенного нижнего предела для i. Например, массив $\{12, 18, 42, 44, 55, 67, 94, 06\}$ с помощью усовершенствованной «пузырьковой» сортировки можно упорядочить за один просмотр, а для сортировки массива $\{94, 06, 12, 18, 42, 44, 55, 67\}$ требуется семь просмотров. Это приводит к мысли: чередовать направление последовательных просмотров. Модификацией сортировки стандартным обменом является ***шейкерная***, или ***челночная***, ***сортировка***.

На рис. 5.7 приведена схема шейкерной сортировки восьми ключей.

5.4. Сортировка Шелла

В методе Шелла сравниваются не соседние элементы, а элементы, расположенные на расстоянии d (где d — шаг между сравниваемыми элементами).

Если $d = [n/2]$, то после каждого просмотра шаг d уменьшается вдвое. На последнем просмотре он сокращается до $d = 1$.

Например, пусть дан список, в котором число элементов четно:

$$\{40, 11, 83, 57, 32, 21, 75, 64\}.$$

Список длины n разбивается на $n/2$ частей, т.е. $d = [n/2] = 4$, где $[\ ]$ — целая часть числа.

При первом просмотре сравниваются элементы, отстоящие друг от друга на $d = 4$ (рис. 5.8), т.е. k_1 и k_5, k_2 и k_6 и т.д. Если $k_i > k_{i+d}$, то происходит обмен между позициями i и $(i + d)$. Перед вторым просмотром выбирается шаг $d = [d/2] = 2$ (рис. 5.9). Затем выбираем шаг $d = [d/2] = 1$ (рис. 5.10), т.е. имеем аналогию с методом стандартного обмена.

Сложность метода Шелла $O\left(0{,}3n(\log_2 n)^2\right)$.

Исходный массив	40	11	83	57	32	21	75	64
Шаг $d = 4$	32	11	75	57	40	21	83	64
Полученный массив	32	11	75	57	40	21	83	64

Рис. 5.8. Метод Шелла (шаг $d = 4$)

Исходный массив	32	11	75	57	40	21	83	64
Шаг $d = 2$	32	11	75	40	57	21	75	57	83	75	57	64
Полученный массив	32	11	40	21	75	57	83	64

Рис. 5.9. Метод Шелла (шаг $d = 2$)

Исходный список	32	11	40	21	75	57	83	64
Шаг $d = 1$	11	32	32	40	21	40	40	75	57	75	75	83	64	83
Полученный список	11	32	21	40	57	75	64	(83)

Рис. 5.10. Метод Шелла (шаг $d = 1$)

5.5. Быстрая сортировка (сортировка Хоара)

В методе быстрой сортировки фиксируется какой-либо ключ (базовый), относительно которого все элементы с большим весом перемещаются вправо, а с меньшим — влево. При этом весь список элементов делится относительно базового ключа на две части. Для каждой части процесс повторяется. Поясним метод на примере.

На рис. 5.11 представлены этапы быстрой сортировки Хоара. В первой строке указана исходная последовательность.

Номер шага	$i \longrightarrow$					$\longleftarrow j$			*Примечание*
	40	11	83	57	32	21	75	64	Исходный список
1	[40]							(64)	$k_0 < k_j$
2	[40]						(75)		$k_0 < k_j$
3	[40] (21)					(21) [40]			Обмен; $k_0 > k_j$
4		(11)				[40]			$k_i < k_0$
5			(83) [40]			[40] (83)			Обмен; $k_i > k_0$
6			[40] (32)		(32) [40]				Обмен; $k_0 > k_j$
7				(57) [40]	[40] (57)				Обмен; $k_i > k_0$
	21	11	32	[40]	57	83	75	64	Полученный список

Рис. 5.11. Метод Хоара

Примем первый элемент последовательности за базовый ключ, выделим его квадратом и обозначим $k_0 = 40$. Установим два указателя i и j, из которых i начинает отсчет слева ($i = 1$), а j — справа ($j = n$).

Сравниваем базовый ключ k_0 и текущий ключ k_j. Если $k_0 \leq k_j$, то устанавливаем $j = j - 1$ и проводим следующее сравнение k_0 и k_j. Продолжаем уменьшать j до тех пор, пока не достигнем условия $k_0 > k_J$. После этого меняем местами ключи k_0 и k_j (см. шаг 3 на рис. 5.11).

Теперь начинаем изменять индекс $i = i + 1$ и сравнивать элементы k_i и k_0. Продолжаем увеличение i до тех пор, пока не получим условие $k_i > k_0$, после чего следует обмен k_i и k_0 (см. шаг 5). Снова возвращаемся к индексу j, уменьшаем его. Чередуя уменьшение j и увеличение i, продолжаем этот процесс с обоих концов к середине до тех пор, пока не получим $i = j$ (см. шаг 7).

В отличие от предыдущих рассмотренных сортировок уже на первом этапе имеют место два факта: во-первых, базовый ключ $k_0 = 40$ занял свое постоянное место в сортируемой последовательности; во-вторых, все элементы слева от k_0 будут меньше него, а справа — больше него. Таким образом, по окончании первого этапа имеем:

21, 11, 32	40	57, 83, 75, 64
Левая часть		Правая часть

Указанная процедура сортировки применяется независимо к левой и правой частям.

Сложность метода Хоара: $O(n \log_2 n)$.

5.6. Турнирная сортировка

Элементы исходного множества представляются листьями дерева. Их попарное сравнение позволяет определить минимальный (максимальный) элемент. Метод турнирной сортировки основан на повторяющихся поисках наименьшего ключа среди n элементов, среди оставшихся $n - 1$ элементов и т.д. Например, сделав $n/2$ сравнений, можно определить в каждой паре ключей меньший. С помощью $n/4$ сравнений — меньший из пары уже выбранных меньших и т.д.

Проделав $n - 1$ сравнений, можно построить дерево выбора (рис. 5.12) и идентифицировать его корень как наименьший ключ.

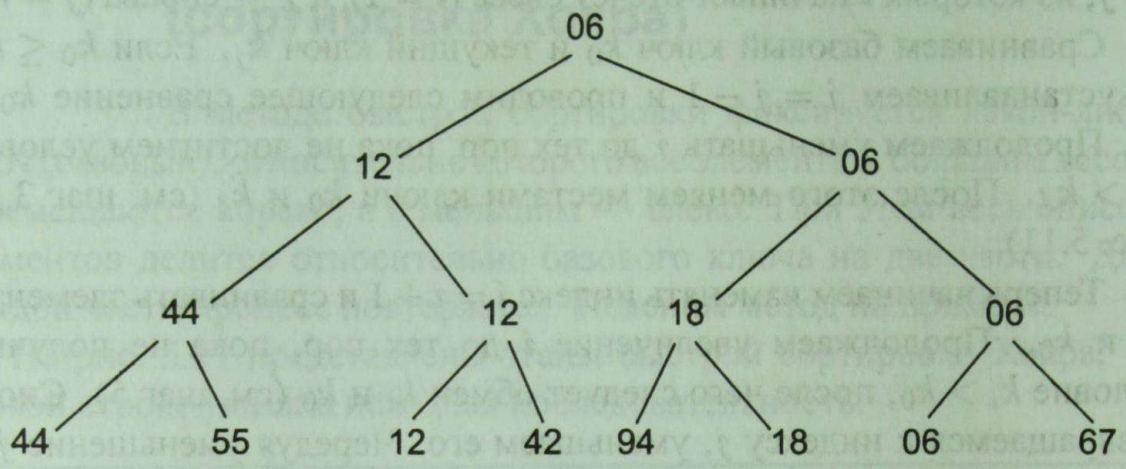


Рис. 5.12. Повторяющиеся выборы среди ключей

Следующий этап сортировки — спуск вдоль пути, отмеченного наименьшим элементом, и исключение его из дерева путем замены либо на пустой элемент (дырку) в самом низу, либо на элемент из соседней ветви в промежуточных вершинах (рис. 5.13).

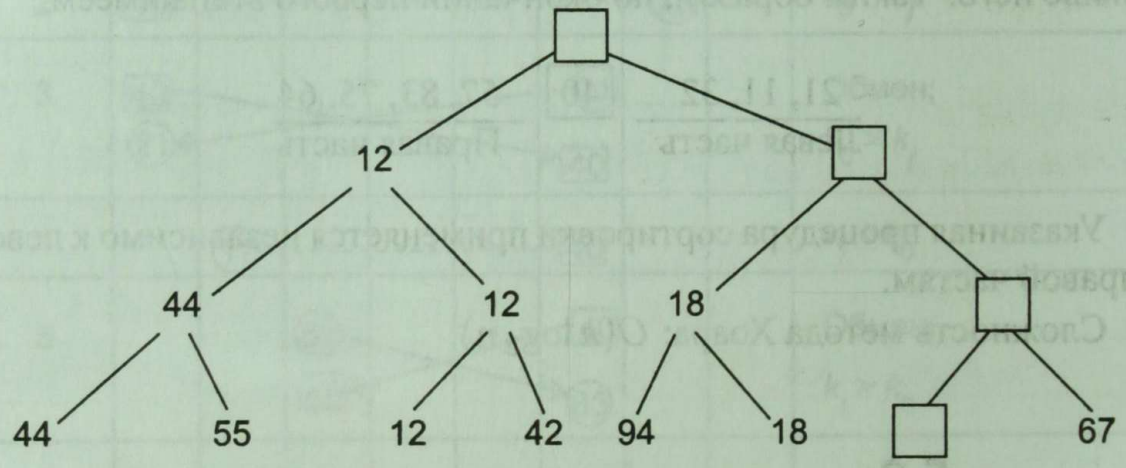


Рис. 5.13. Исключение наименьшего ключа

Элемент, передвинувшийся в корень дерева, вновь будет наименьшим (теперь уже вторым) ключом (рис. 5.14), и его можно исключить. После n таких шагов дерево станет пустым (т.е. в нем останутся одни дырки), и процесс сортировки заканчивается.

Свое название турнирная сортировка получила, потому что она используется при проведении соревнований, турниров и олимпиад.

Пример 1. Дано исходное множество $\{a1, a2, a3, a4, a5, a6, a7, a8\}$. Осуществить турнирную сортировку.

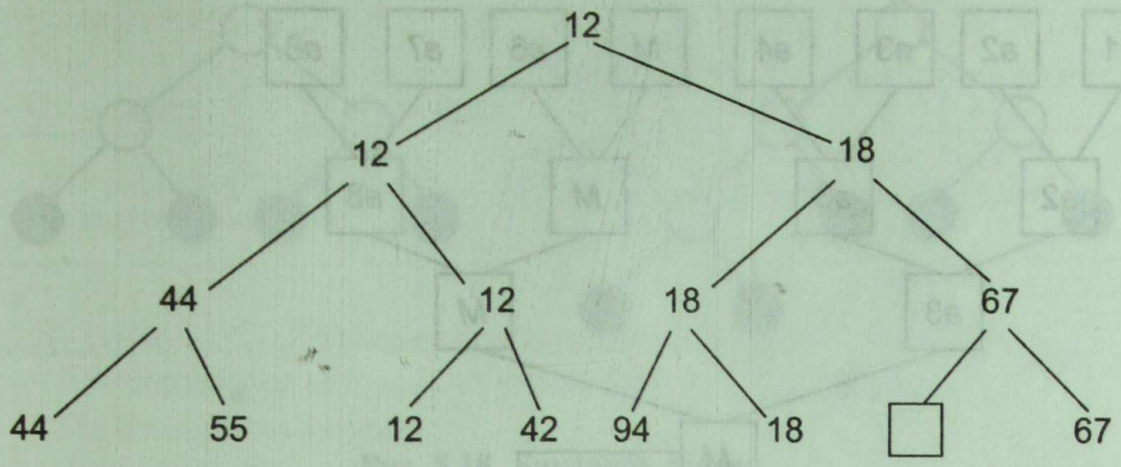


Рис. 5.14. Заполнение дырки

Производится попарное сравнение вершин дерева (рис. 5.15). Найденный минимальный элемент заменяется специальным символом M и помещается в результирующее множество (рис. 5.16).

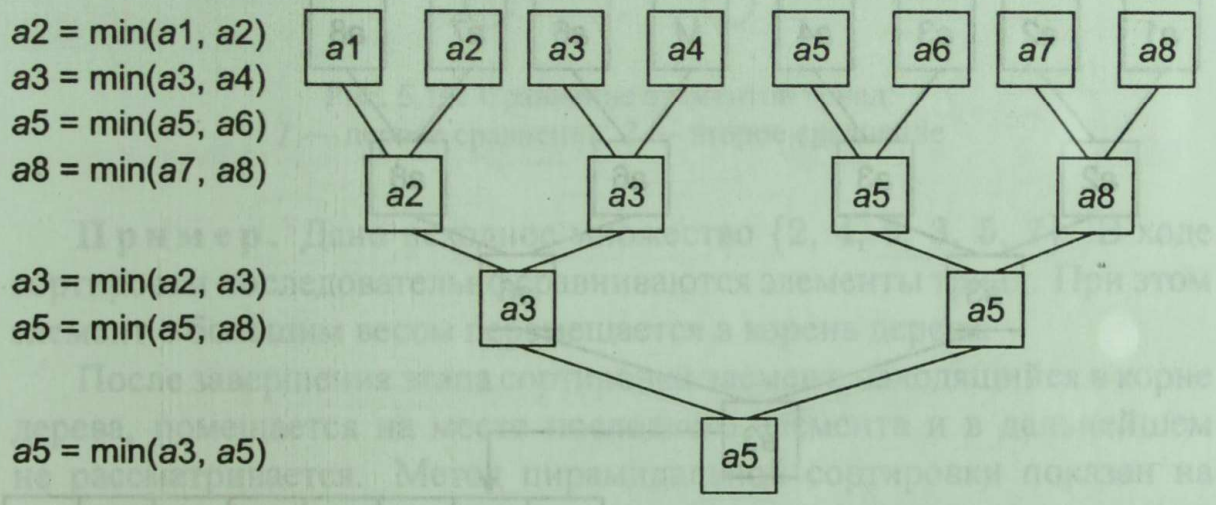


Рис. 5.15. Первый этап турнирной сортировки

На последующих этапах найденные минимальные элементы помещаются в результирующее множество (рис. 5.17).

5.7. Пирамидальная сортировка

Данный тип сортировки заключается в построение пирамидального дерева. Пирамидальное дерево — это бинарное дерево, обладающее тремя свойствами:

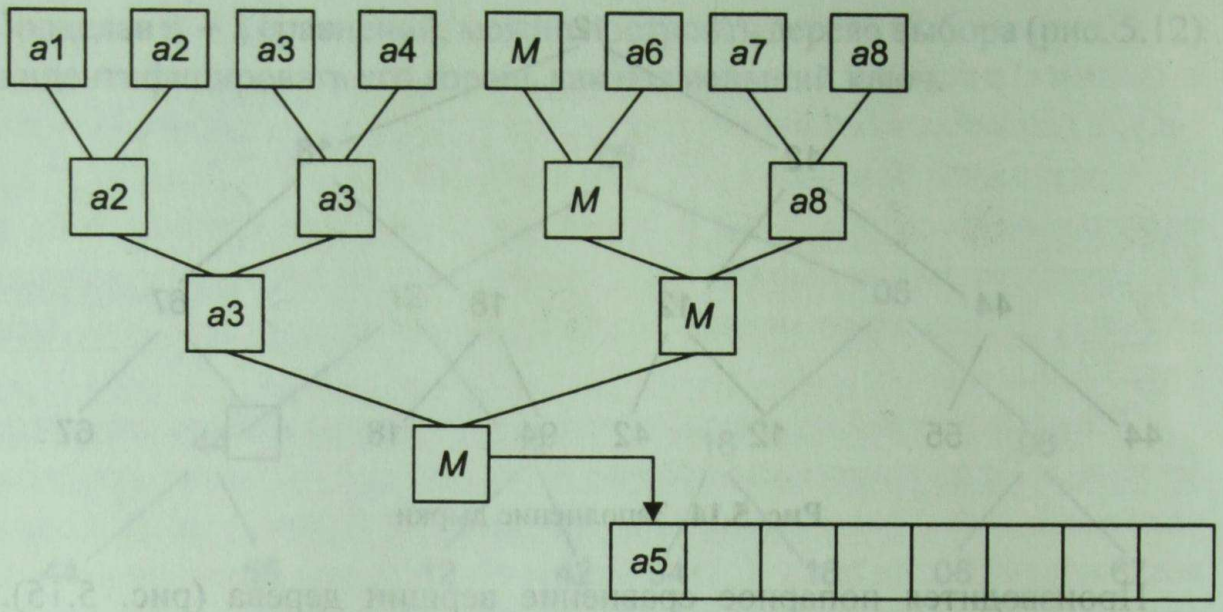


Рис. 5.16. Удаление минимального элемента

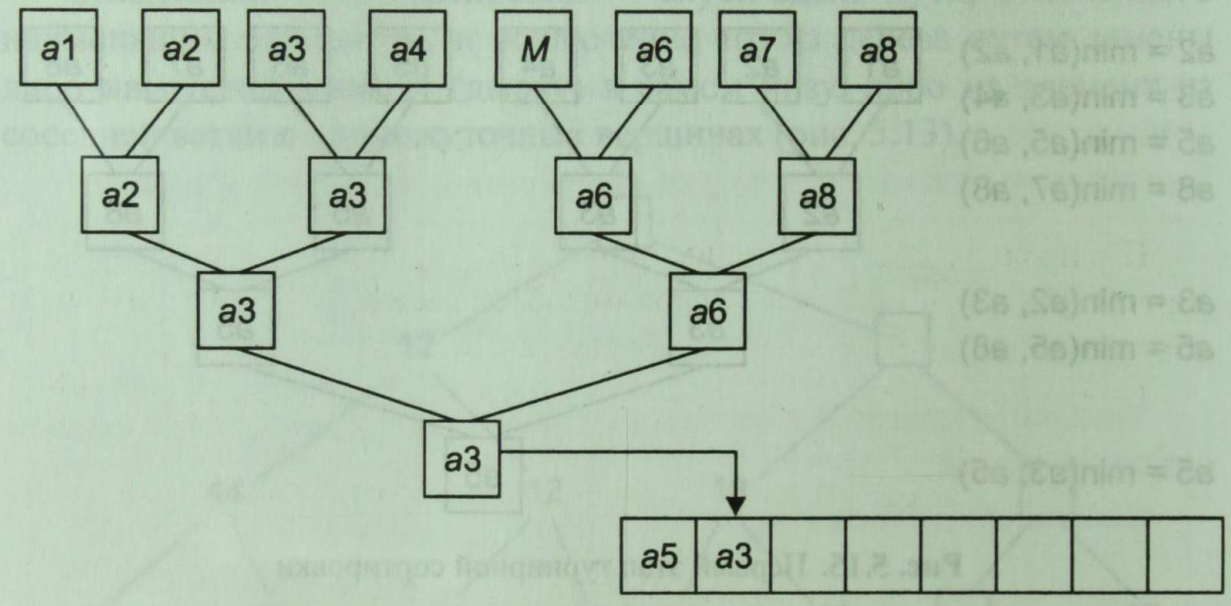


Рис. 5.17. Этапы турнирной сортировки

• в вершине каждой триады располагается элемент с большим весом;

• листья бинарного дерева располагаются либо в одном уровне, либо в двух соседних (рис. 5.18);

• листья нижнего уровня располагаются левее листьев более высокого уровня.

В ходе преобразования элементы триад сравниваются дважды (рис. 5.19), при этом элемент с большим весом перейдет вверх, а с меньшим — вниз.

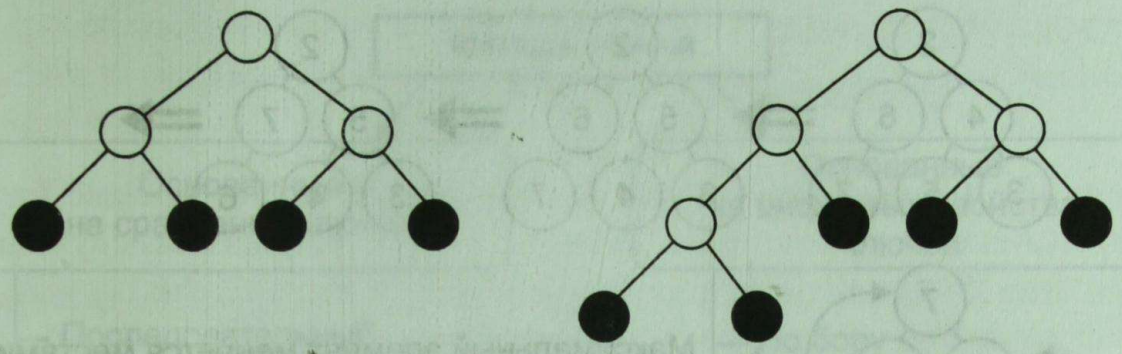


Рис. 5.18. Бинарное дерево:
а — листья на одном уровне;
б — листья на соседних уровнях

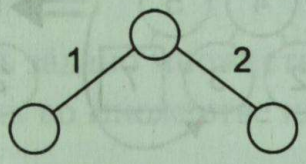


Рис. 5.19. Сравнение элементов триад:
1 — первое сравнение; *2* — второе сравнение

Пример. Дано исходное множество {2, 4, 6, 3, 5, 7}. В ходе сортировки последовательно сравниваются элементы триад. При этом элемент с большим весом перемещается в корень дерева.

После завершения этапа сортировки элемент, находящийся в корне дерева, помещается на место последнего элемента и в дальнейшем не рассматривается. Метод пирамидальной сортировки показан на рис. 5.20.

В результате будет получено упорядоченное множество {2, 3, 4, 5, 6, 7}.

Контрольные вопросы

1. Что понимается под сортировкой?
2. Каковы особенности сортировки: вставкой, выбором, обменом?
3. Каковы особенности сортировки Шелла и Хоара?
4. Каковы особенности сортировки турнирной и пирамидальной?
5. Какова основная идея шейкерной сортировки?
6. К какой группе методов относится сортировка фон Неймана?
7. В чем состоит методика анализа сложности алгоритмов сортировки?

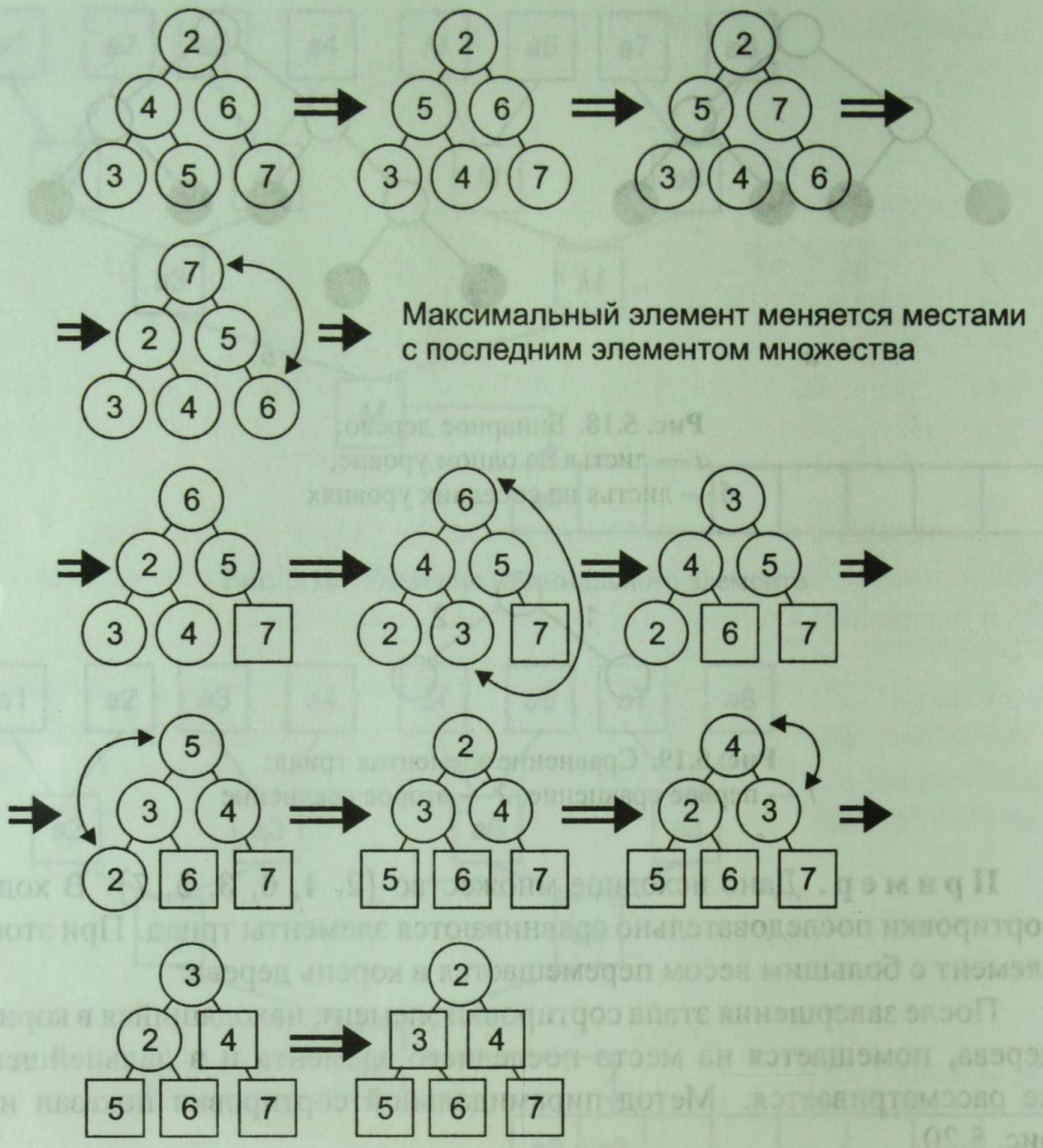


Рис. 5.20. Пирамидальная сортировка

Глава 6
Методы поиска

Предметы (объекты), составляющие множество, называются его ***элементами***. Элемент множества будет называться ***ключом*** и обозначаться латинской буквой k_i с индексом, указывающим номер элемента.

Алгоритмы поиска можно разбить на следующие группы (рис. 6.1).

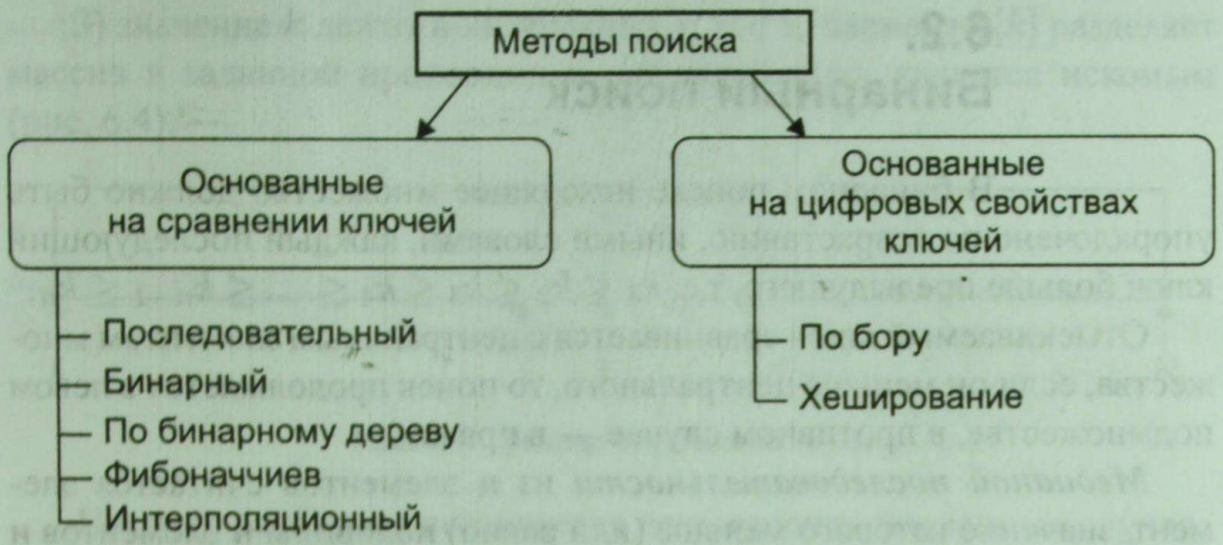


Рис. 6.1. Методы поиска

Задача поиска. Пусть задано множество ключей $\{k_1, k_2, k_3, \ldots, k_n\}$. Необходимо отыскать во множестве ключ k_i. Поиск может быть завершен в двух случаях:

- ключ во множестве отсутствует;
- ключ найден во множестве.

6.1. Последовательный поиск

В последовательном поиске исходное множество не упорядочено, т.е. имеется произвольный набор ключей $\{k_1, k_2, k_3, \ldots, k_n\}$. Метод заключается в том, что отыскиваемый ключ k_i последовательно сравнивается со всеми элементами множества. При этом поиск заканчивается досрочно, если ключ найден.

Алгоритм поиска сводится к последовательности шагов:

Шаг S_1. [Начальная установка.] Установить $i := 1$.

Шаг S_2. [Сравнение.] Если $k = k_i$, алгоритм заканчивается удачно.

Шаг S_3. [Продвижение.] Увеличить i на 1.

Шаг S_4. [Конец файла?] Если $i \leq N$, то вернуться к шагу S_2. В противном случае алгоритм заканчивается неудачно.

6.2. Бинарный поиск

В бинарном поиске исходящее множество должно быть упорядочено по возрастанию, иными словами, каждый последующий ключ больше предыдущего, т.е. $k_1 \le k_2 \le k_3 \le k_4 \le \ldots \le k_{n-1} \le k_n$.

Отыскиваемый ключ сравнивается с центральным элементом множества, если он меньше центрального, то поиск продолжается в левом подмножестве, в противном случае — в правом.

Медианой последовательности из n элементов считается элемент, значение которого меньше (или равно) половины n элементов и больше (или равно) другой половины. Задачу поиска медианы принято связывать с сортировкой, так как медиану всегда можно найти следующим способом: отсортировать n элементов и затем выбрать средний элемент.

В алгоритме К. Хоара для нахождения медианы используется операция разделения, применяемая при быстрой сортировке, с $L = 1$, $R = n$ и с $a[k]$, выбранными в качестве разделяющего значения x. Получаются значения индексов i и j:

1) разделяющее значение x было слишком мало; в результате граница между двумя частями ниже искомого значения k. Процесс разделения следует повторить для элементов $a[i], \ldots, a[R]$ (рис. 6.2);

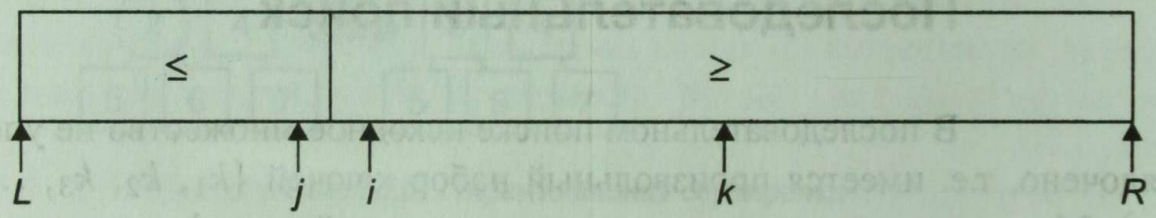


Рис. 6.2. Смещение границы влево

2) выбранная граница x была слишком велика. Операцию разбиения следует повторить на подмассиве $a[L], \ldots, a[j]$ (рис. 6.3);

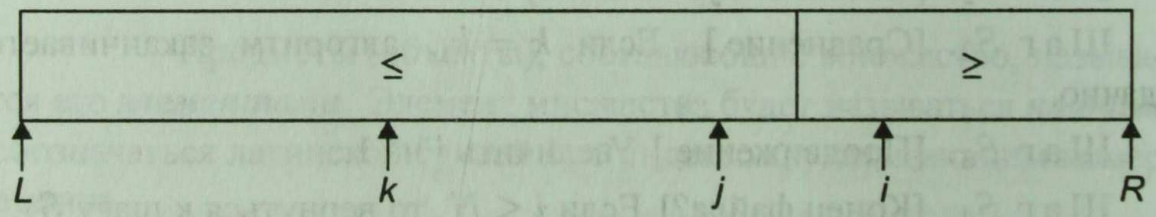


Рис. 6.3. Смещение границы вправо

3) значение k лежит в интервале $j < k < i$: элемент $a[k]$ разделяет массив в заданной пропорции и, следовательно, является искомым (рис. 6.4).

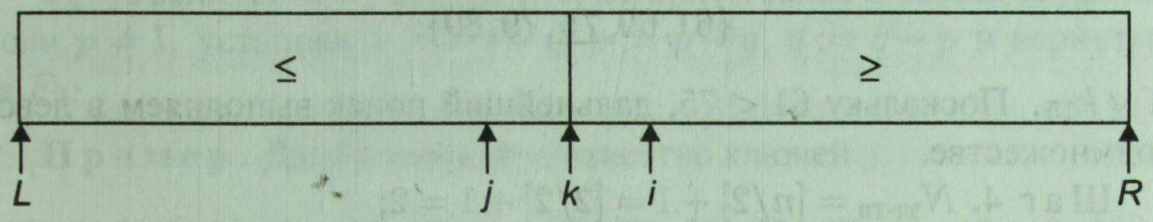


Рис. 6.4. Граница определена точно

Процесс разбиения повторяется до появления последнего случая.

Центральный элемент находится по формуле $N_{\text{эл-та}} = [n/2] + 1$, где квадратные скобки обозначают, что от деления берется только целая часть, дробная часть отбрасывается. В методе бинарного поиска анализируются только центральные элементы подмножеств.

Пример 1. Во множестве элементов отыскать ключ, равный 653. В квадратных скобках выделены множества анализируемых элементов. Центральные элементы подмножеств подчеркнуты.

Поиск ключа $K = 653$ осуществляется за четыре шага:

[061 087 154 170 275 426 503 <u>509</u> 512 612 653 677 703 765 897 908]
061 087 154 170 275 426 503 509 **[512 612 653 <u>677</u> 703 765 897 908]**
061 087 154 170 275 426 503 509 **[512 <u>612</u> 653]** 677 703 765 897 908
061 087 154 170 275 426 503 509 512 612 **[<u>653</u>]** 677 703 765 897 908

Пример 2. Дано упорядоченное множество элементов

{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, 60, 61, 69,
75, 79, 80, 81, 95, 101, 123, 198}.

Найти во множестве ключ $K = 61$.

Шаг 1. $N_{\text{эл-та}} = [n/2] + 1 = [22/2] + 1 = 12$;

{7, 8, 12, 16, 18, 20, 30, 38, 49, 50, 54, **<u>60</u>**, 61, 69,
75, 79, 80, 81, 95, 101, 123, 198};

$K \vee k_{12}$ (значок «$\vee$» обозначает сравнение элементов: чисел, значений). Поскольку $61 > 60$, дальнейший поиск выполняем в правом подмножестве.

Шаг 2. $N_{\text{эл-та}} = [n/2] + 1 = [10/2] + 1 = 6$;

{61, 69, 75, 79, 80, **<u>81</u>**, 95, 101, 123, 198};

$K \vee k_{18}$. Поскольку $61 < 81$, дальнейший поиск выполняем в левом подмножестве.

Шаг 3. $N_{\text{эл-та}} = [n/2] + 1 = [5/2] + 1 = 3$;

$$\{61, 69, \underline{\mathbf{75}}, 79, 80\};$$

$K \vee k_{15}$. Поскольку $61 < 75$, дальнейший поиск выполняем в левом подмножестве.

Шаг 4. $N_{\text{эл-та}} = [n/2] + 1 = [2/2] + 1 = 2$;

$$\{61, \underline{\mathbf{69}}\};$$

$K \vee k_{14}$. Поскольку $61 < 69$, дальнейший поиск выполняем в левом подмножестве.

Шаг 5. $\{\underline{\mathbf{61}}\}$. $K \vee k_{13}$. Так как $61 = 61$, делаем вывод: искомый ключ найден под номером 13.

6.3. Фибоначчиев поиск

В этом поиске анализируются элементы, находящиеся в позициях, равных числам Фибоначчи. Числа Фибоначчи получаются по следующему правилу: каждое последующее число равно сумме двух предыдущих чисел, например:

$$\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots\}.$$

Поиск продолжается до тех пор, пока не будет найден интервал между двумя ключами, где может располагаться отыскиваемый ключ.

Фибоначчиев поиск предназначается для поиска аргумента K среди расположенных в порядке возрастания ключей $K_1 < K_2 < \ldots$ $\ldots < K_n$.

Для удобства описания предполагается, что $n + 1$ есть число Фибоначчи F_{k+1}. Подходящей начальной установкой данный метод можно сделать пригодным для любого n.

F_1. [Начальная установка.] Установить $i := F_k$, $p := F_{k-1}$, $q :=$ $:= F_{k-2}$. (В алгоритме p и q обозначают последовательные числа Фибоначчи.)

F_2. [Сравнение.] Если $K < K_i$, то перейти на F_3; если $K > K_i$, то перейти на F_4; если $K = K_i$, алгоритм заканчивается удачно.

F_3. [Уменьшение i.] Если $q = 0$, алгоритм заканчивается неудачно. Если $q \neq 0$, то установить $i := i - q$, заменить (p, q) на $(q, p - q)$ и вернуться на F_2.

F_4. [Увеличение i.] Если $p = 1$, алгоритм заканчивается неудачно. Если $p \neq 1$, установить $i := i + q$, $p := p - q$, $q := q - p$ и вернуться на F_2.

Пример. Дано исходное множество ключей

$$\{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52\}.$$

Пусть отыскиваемый ключ равен 42 ($K = 42$).

Последовательное сравнение отыскиваемого ключа будет проводиться с элементами исходного множества, расположенными в позициях, равных числам Фибоначчи: 1, 2, 3, 5, 8, 13, 21, ...

Шаг 1. $K \sim k_1$, $42 > 3$, отыскиваемый ключ сравнивается с ключом, стоящим в позиции, равной числу Фибоначчи.

Шаг 2. $K \sim k_2$, $42 > 5$, сравнение продолжается с ключом, стоящим в позиции, равной следующему числу Фибоначчи.

Шаг 3. $K \sim k_3$, $42 > 8$, сравнение продолжается.

Шаг 4. $K \sim k_5$, $42 > 11$, сравнение продолжается.

Шаг 5. $K \sim k_8$, $42 > 19$, сравнение продолжается.

Шаг 6. $K \sim k_{13}$, $42 > 35$, сравнение продолжается.

Шаг 7. $K \sim k_{18}$, $42 < 52$, найден интервал, в котором находится отыскиваемый ключ, т.е. отыскиваемый ключ может находиться в исходном множестве между позициями 13 и 18, т.е. $\{35, 37, 42, 45, 48, 52\}$.

В найденном интервале поиск вновь ведется в позициях, равных числам Фибоначчи.

6.4. Интерполяционный поиск

Исходное множество должно быть упорядочено по возрастанию весов. Первоначальное сравнение осуществляется на расстоянии шага d, который определяется по формуле:

$$d = \left[\frac{(j - i)(K - K_i)}{K_j - K_i} \right],$$

где i — номер первого рассматриваемого элемента;
j — номер последнего рассматриваемого элемента;
K — отыскиваемый ключ;
K_i, K_j — значения ключей в позициях i и j;
[ ] — целая часть числа.

Идея метода заключается в следующем: шаг d меняется после каждого этапа по формуле, приведенной выше (рис. 6.5).

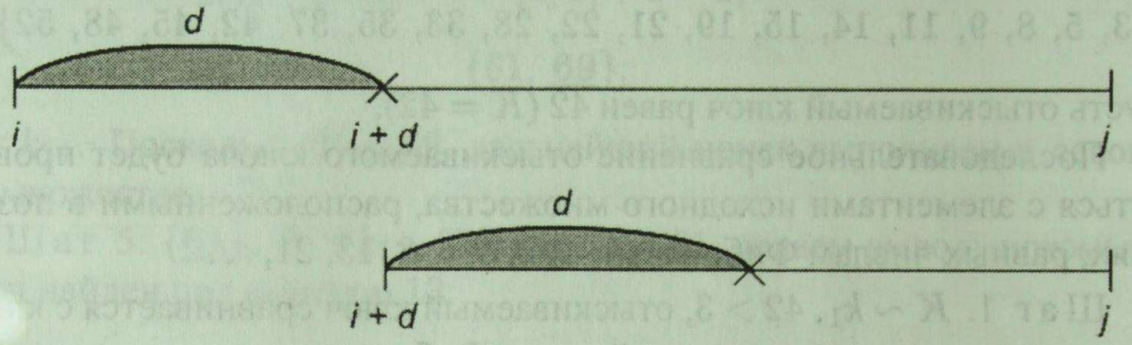


Рис. 6.5. Схема интерполяционного поиска

Алгоритм заканчивает работу при $d = 0$, при этом анализируются соседние элементы, после чего принимается окончательно решение о результатах поиска.

Этот метод прекрасно работает, если исходное множество представляет собой арифметическую прогрессию или множество, приближенное к ней.

Пример 1. Дано множество ключей:

$\{2, 9, 10, 12, 20, 24, 28, 30, 37, 40, 45, 50, 51, 60, 65, 70, 74, 76\}$.

Пусть искомый ключ $K = 70$.

Определим шаг d для исходного множества ключей ($i = 1; j = 18$):

$$d = \left[\frac{(18-1)(70-2)}{76-2}\right] = 15.$$

Сравниваем ключ, стоящий под шестнадцатым порядковым номером в данном множестве с искомым ключом: $k_{16} \vee K$; $70 = 70$; ключ найден.

Пример 2. Дано множество ключей:

$\{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95, 98, 102\}$.

Требуется отыскать ключ $K = 90$.

Ш а г 1. Определим шаг d для исходного множества ключей ($i = 1$; $j = 17$):

$$d = \left[\frac{(17-1)(90-4)}{102-4}\right] = 14.$$

Сравниваем ключ, стоящий под пятнадцатым порядковым номером, с отыскиваемым ключом: $k_{15} \vee K$, $95 > 90$, следовательно, сужается область поиска:

$$\{4,\ 5,\ 10,\ 23,\ 24,\ 30,\ 47,\ 50,\ 59,\ 60,\ 64,\ 65,\ 77,\ 90,\ 95\}.$$

Ш а г 2. Определим шаг d для множества ключей ($i = 1$; $j = 15$):

$$d = \left[\frac{(15-1)(90-4)}{95-4}\right] = 13.$$

Сравниваем ключ, стоящий под четырнадцатым порядковым номером, с отыскиваемым ключом: $k_{14} \vee K$; $k_{14} = K$; $90 = 90$; ключ найден.

6.5. Поиск по бинарному дереву

Дерево двоичного поиска для множества чисел S — это размеченное *бинарное дерево*, каждой *вершине* которого сопоставлено число из множества S, причем все пометки удовлетворяют следующему простому правилу: «если больше — направо, если меньше — налево».

Например, для набора чисел $\{7,\ 3,\ 5,\ 2,\ 8,\ 1,\ 6,\ 10,\ 9,\ 4,\ 11\}$ получится бинарное дерево (рис. 6.6). Для того чтобы правильно учесть повторения чисел, можно ввести дополнительное поле, которое будет хранить количество вхождений для каждого числа.

Бинарное дерево, соответствующее бинарному поиску среди n записей, можно построить следующим образом: при $n = 0$ дерево сводится к узлу 0. В противном случае корневым узлом является $[n/2]$, левое поддерево соответствует бинарному дереву с $[n/2] - 1$ узлами, а правое — дереву с $[n/2]$ узлами и числами в узлах, увеличенными на $[n/2]$ (рис. 6.7).

Число узлов, порожденных отдельным узлом (число поддеревьев данного корня), называется его ***степенью***. Узел с нулевой степенью

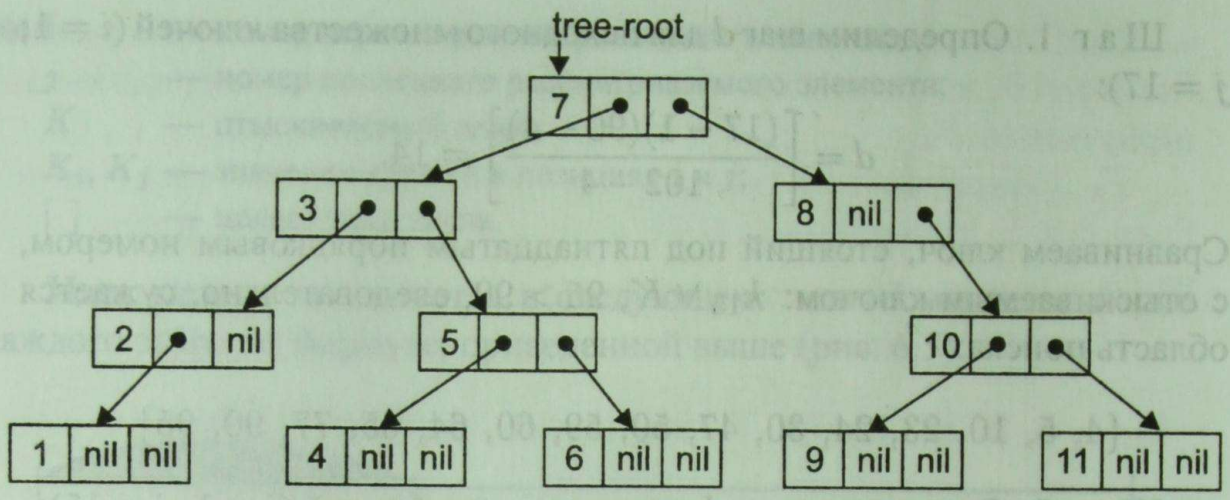


Рис. 6.6. Дерево двоичного поиска

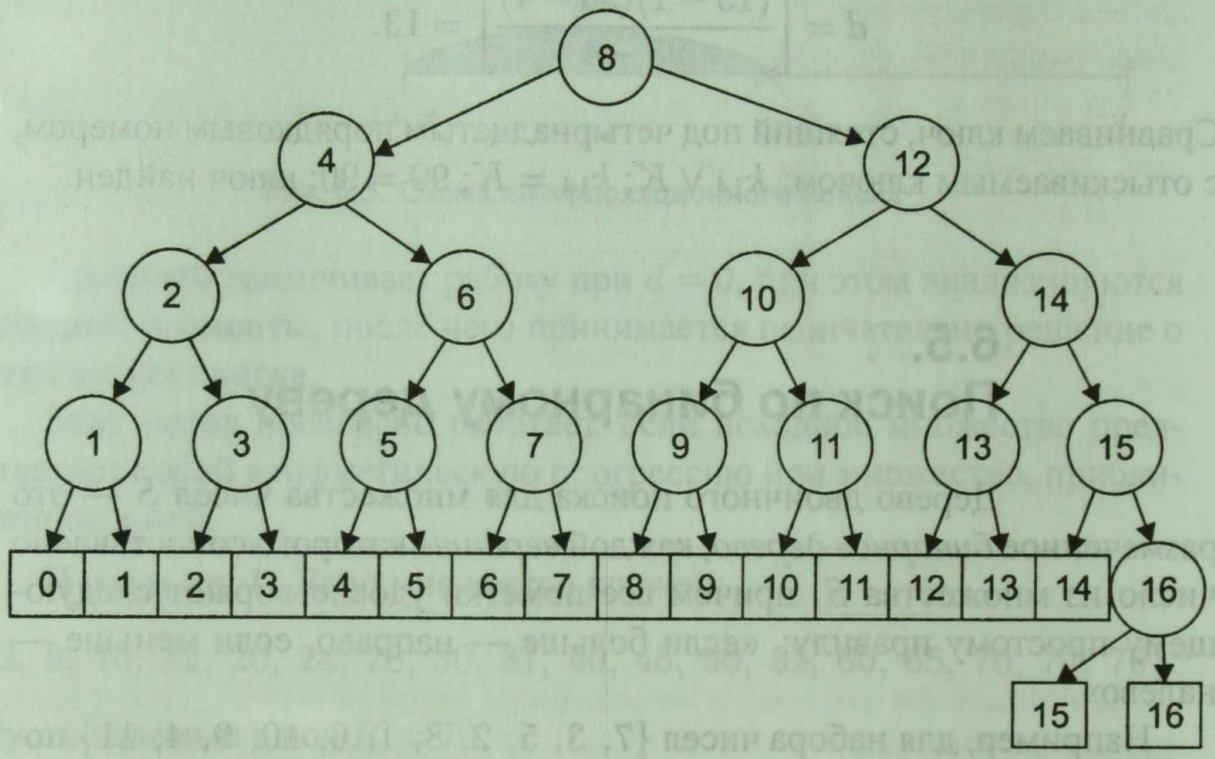


Рис. 6.7. Бинарное дерево, соответствующее бинарному поиску ($n = 16$)

называют ***листом***, или ***концевым узлом***. Максимальное значение степени всех узлов данного дерева называется ***степенью дерева***.

Алгоритм поиска по бинарному дереву: вначале аргумент поиска сравнивается с ключом, находящимся в корне. Если аргумент совпадает с ключом, поиск закончен, если не совпадает, то в случае, когда аргумент оказывается меньше ключа, поиск продолжается в левом поддереве, а в случае, когда больше ключа, — в правом поддереве. Увеличив уровень на 1, повторяют сравнение, считая текущий узел корнем.

Использование структуры бинарного дерева позволяет быстро вставлять и удалять записи и проводить эффективный поиск по таблице. Такая гибкость достигается добавлением в каждую запись двух полей для хранения ссылок.

Пусть дано бинарное дерево поиска (рис. 6.8). Требуется по бинарному дереву отыскать ключ SAG.

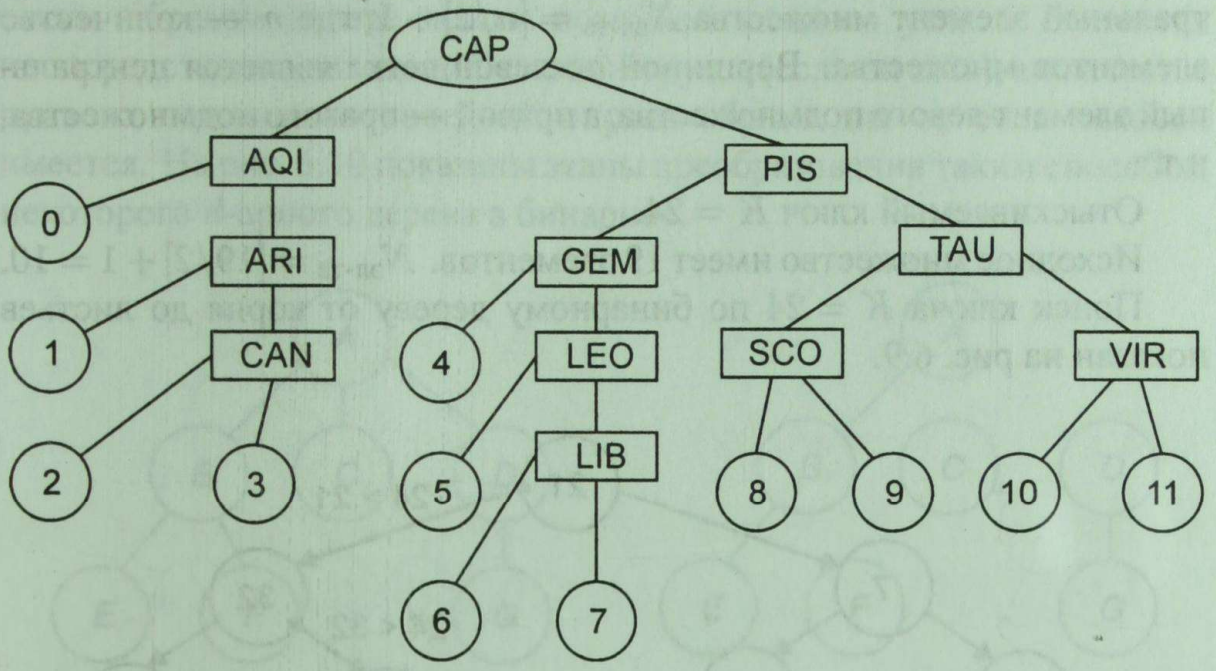


Рис. 6.8. Бинарное дерево

При просмотре от корня дерева видно, что по первой букве латинского алфавита название SAG больше чем CAP. Следовательно, дальнейший поиск будем осуществлять в правой ветви. Это слово больше, чем PIS, значит, снова идем вправо; оно меньше, чем TAU, — идем влево; оно меньше, чем SCO, попадаем в узел 8. Таким образом, название SAG должно находиться в узле 8. При этом узлы дерева имеют структуру, представленную в табл. 6.1.

Таблица 6.1

Структура узлов дерева

Ключ	Информационная часть	Указатель на левое поддерево	Указатель на правое поддерево
KEY	(может отсутствовать)	LLINK	RLINK

Пример. Пусть дано исходное множество ключей

$\{2, 4, 5, 6, 7, 9, 12, 14, 18, \underline{21}, 24, 25, 27, 30, 32, 33, 34, 37, 39\}$.

Исходное множество ключей должно быть упорядочено по возрастанию.

От линейного списка переходим к построению бинарного дерева поиска (рис. 6.9). В данном случае корнем дерева является центральный элемент множества $N_{\text{эл-та}} = [n/2] + 1$, где n — количество элементов множества. Вершиной по левой ветке является центральный элемент левого подмножества, а правой — правого подмножества, и т.д.

Отыскиваемый ключ $K = 24$.

Исходное множество имеет 19 элементов. $N_{\text{эл-та}} = [19/2] + 1 = 10$.

Поиск ключа $K = 24$ по бинарному дереву от корня до листьев показан на рис. 6.9.

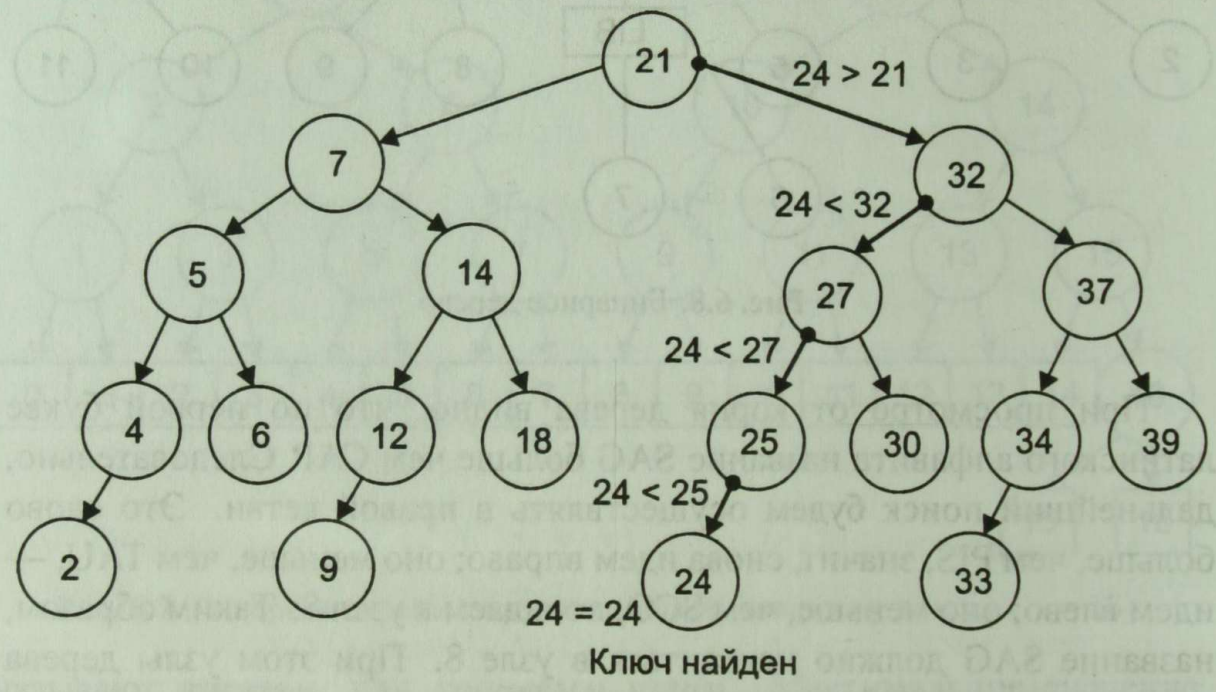


Рис. 6.9. Бинарное дерево поиска

Преобразование произвольного дерева в бинарное. Упорядоченные деревья степени 2 называются ***бинарными деревьями***. Бинарное дерево состоит из конечного множества элементов (узлов), каждый из которых либо пуст, либо состоит из корня (узла), связанного с двумя различными бинарными деревьями, называемыми *левым* и *правым поддеревом корня*. Деревья, у которых $d > 2$, называются *d-арными*.

Преобразование произвольного дерева с упорядоченными узлами в бинарное дерево сводится к следующим действиям. Сначала в каждом узле исходного дерева вычеркиваем все ветви, кроме самых левых ветвей. После этого в получившемся графе соединяем горизонтальными ветвями те узлы одного уровня, которые являются «братьями» в исходном дереве. (Поясним, что если несколько узлов имеют общего предка, то такие узлы называются «братьями»). В получившемся таким образом дереве левым потомком каждого узла x считается непосредственно находящийся под ним узел (если он есть), а в качестве правого потомка — соседний справа «брат» для x, если таковой имеется. На рис. 6.10 показаны этапы преобразования таким способом некоторого d-арного дерева в бинарное.

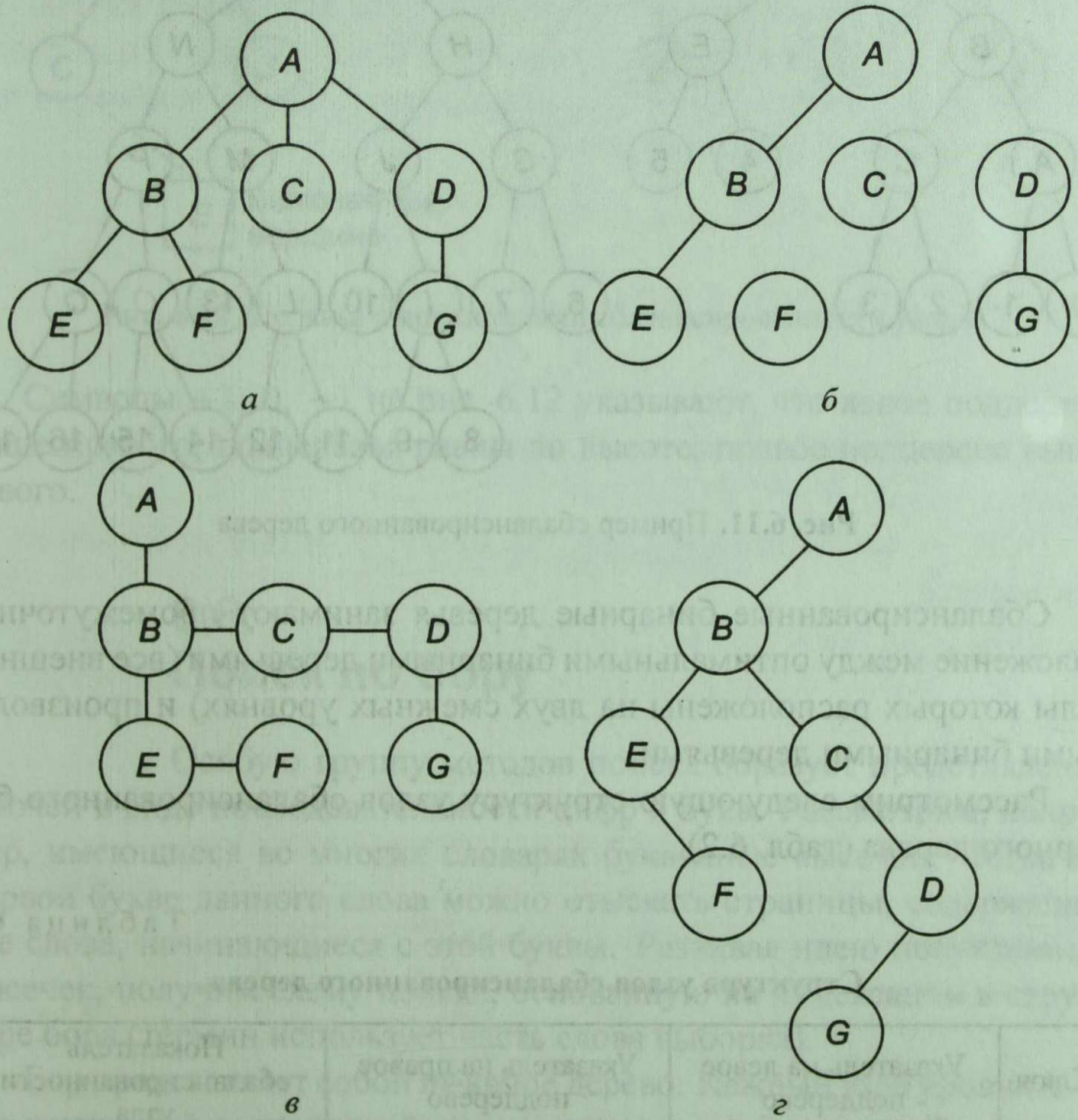


Рис. 6.10. Этапы преобразования d-арного дерева в бинарное ($а$–$г$)

Переход от произвольного дерева к его бинарному эквиваленту не только облегчает анализ логической структуры, но также упрощает машинное представление, т.е. физическую структуру дерева.

Сбалансированное бинарное дерево. Бинарное дерево называется сбалансированным (*B-balanced*), если высота левого поддерева каждого узла отличается от высоты правого не более чем на 1 (рис. 6.11).

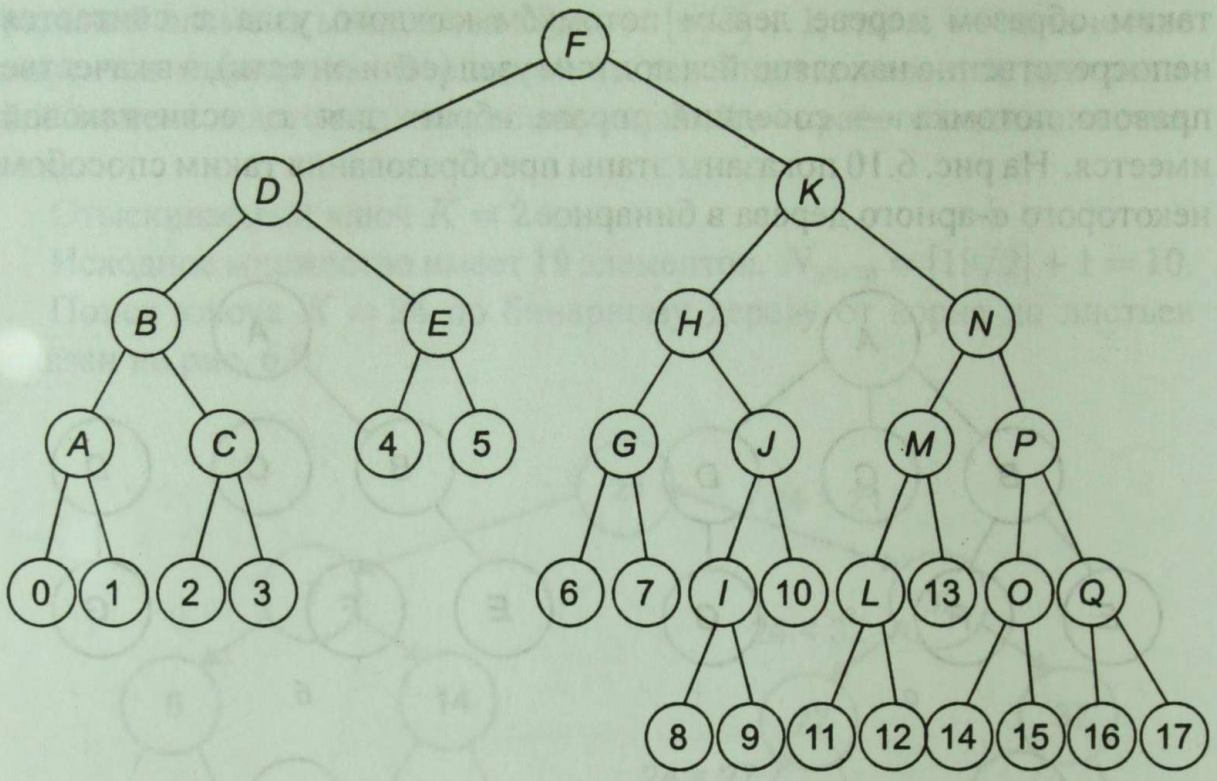


Рис. 6.11. Пример сбалансированного дерева

Сбалансированные бинарные деревья занимают промежуточное положение между оптимальными бинарными деревьями (все внешние узлы которых расположены на двух смежных уровнях) и произвольными бинарными деревьями.

Рассмотрим следующую структуру узлов сбалансированного бинарного дерева (табл. 6.2).

Таблица 6.2

Структура узлов сбалансированного дерева

Ключ	Указатель на левое поддерево	Указатель на правое поддерево	Показатель сбалансированности узла
KEY	LLINK	RLINK	B

B — показатель сбалансированности узла, т.е. разность высот правого и левого поддерева ($B = +1; 0; -1$).

При восстановлении баланса дерева по высоте учитывается показатель B (рис. 6.12).

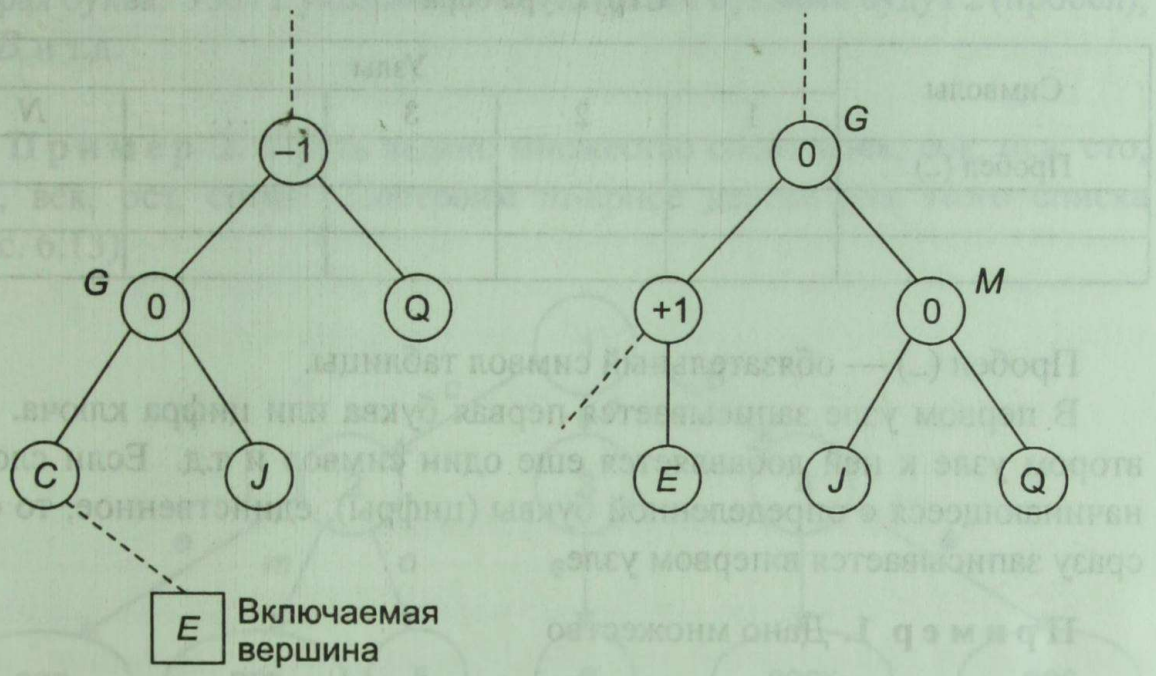


Рис. 6.12. Деревья с показателями сбалансированности узлов

Символы $+1$, 0, -1 на рис. 6.12 указывают, что левое поддерево выше правого, поддеревья равны по высоте, правое поддерево выше левого.

6.6. Поиск по бору

Особую группу методов поиска образует представление ключей в виде последовательности цифр и букв. Рассмотрим, например, имеющиеся во многих словарях буквенные высечки. Тогда по первой букве данного слова можно отыскать страницы, содержащие все слова, начинающиеся с этой буквы. Развивая идею побуквенных высечек, получим схему поиска, основанную на индексации в структуре бора (термин использует часть слова выборка).

Бор представляет собой m-арное дерево. Каждый узел уровня h — это множество всех ключей, начинающихся с определенной последовательности из h литер. Узел определяет m-путевое разветвление в

зависимости от $(h+1)$-й литеры. Структуру бора можно представить в виде табл. 6.3.

Таблица 6.3

Структура бора

Символы	Узлы				
	1	2	3	...	N
Пробел (␣)					

Пробел (␣) — обязательный символ таблицы.

В первом узле записывается первая буква или цифра ключа. Во втором узле к ней добавляется еще один символ и т.д. Если слово, начинающееся с определенной буквы (цифры), единственное, то оно сразу записывается в первом узле.

Пример 1. Дано множество

{A, AA, AB, ABC, ABCD, ABCA, ABCC, BOR,
C, CC, CCC, CCCD, CCCB, CCCA}.

От исходного множества перейдем к построению бора. Исходный алфавит — $\{A, B, C, D\}$.

BOR — единственное слово на букву B, и оно побуквенно не разбивается.

Узлы бора представляют собой векторы, каждая компонента которых представляет собой либо ключ, либо ссылку (возможно пустую) — табл. 6.4.

Таблица 6.4

Пример поиска по бору

Символы	Узлы						
	1	2	3	4	5	6	7
␣		A␣	AB␣	ABC␣	C␣	CC␣	CCC␣
A	2	AA		ABCA			CCCA
B	BOR	3					CCCB
C	5		4	ABCC	6	7	
D				ABCD			CCCD

Узел 1 — корень, и первую букву следует искать здесь. Если первой оказалась, например, буква B, то из таблицы видно, что ему соответствует слово BOR. Если же первая буква A, то первый узел передает управление к узлу 2, где аналогичным образом отыскивается вторая буква. Узел 2 указывает, что вторыми буквами будут ␣ (пробел), A, B и т.д.

Пример 2. Пусть задано множество слов: воск, сок, оса, сто, сев, век, ост, сотка. Построим m-арное дерево для этого списка (рис. 6.13).

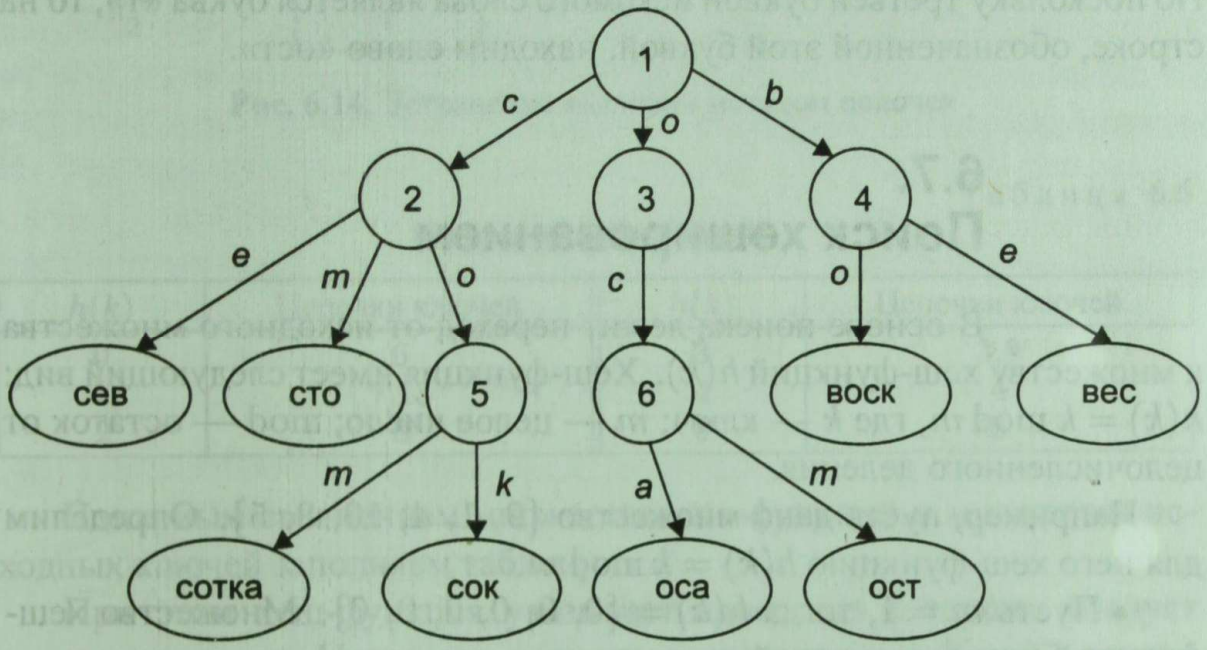


Рис. 6.13. m-арное дерево

На основе анализа этого дерева может быть построена табл. 6.5.

Таблица 6.5

Пример поиска по бору

Символы	Узлы					
	1	2	3	4	5	6
С	(2)		(6)			
О	(3)	(5)		воск		
Т		сто			сотка	ост
К					сок	
А						оса
В	(4)					
Е		сев		вес		

Следует отметить, что при построении бора порядок букв в первом столбце таблицы может быть произвольным. Естественно, что для каждого конкретного порядка букв заполнение таблицы будет разным. Допустим, для нашего примера в списке слов нужно найти слово «ост». Слово начинается на букву «о». Смотрим пересечение первого столбца со строкой, обозначенной буквой «о». Там записана ссылка (3); следовательно, обращаемся к третьему столбцу бора. Вторая буква слова «ост» — буква «с». На пересечении третьего столбца и строки с буквой «с» записана ссылка (6), которая отправляет нас на просмотр шестого столбца бора. В этом столбце видим два слова: «ост» и «оса». Но поскольку третьей буквой искомого слова является буква «т», то на строке, обозначенной этой буквой, находим слово «ост».

6.7. Поиск хешированием

В основе поиска лежит переход от исходного множества к множеству хеш-функций $h(k)$. Хеш-функция имеет следующий вид: $h(k) = k \bmod m$, где k — ключ; m — целое число; mod — остаток от целочисленного деления.

Например, пусть дано множество $\{9, 1, 4, 10, 8, 5\}$. Определим для него хеш-функцию $h(k) = k \bmod m$.

- Пусть $m = 1$, тогда $h(k) = \{0, 0, 0, 0, 0, 0\}$. Множество хеш-функций состоит из нулей.
- Пусть $m = 20$, тогда $h(k) = \{9, 1, 4, 10, 8, 5\}$. Множество хеш-функций повторяет исходное множество.
- Пусть m равно половине максимального ключа $m = [K_{\max}/2]$, тогда $m = [10/2] = 5$; $h(k) = \{4, 1, 4, 0, 3, 0\}$.

Хеш-функция указывает адрес, по которому следует отыскивать ключ. Для разных ключей хеш-функция может принимать одинаковые значения, такая ситуация называется *коллизией*. Таким образом, поиск хешированием заключается в устранении коллизий методом цепочек (рис. 6.14).

Пример 1. Дано множество ключей $\{7, 13, 6, 3, 9, 4, 8, 5\}$. Найти ключ $K = 27$.

Хеш-функция равна $h(k) = k \bmod m$; $m = [13/2] = 6$ (так как 13 — максимальный ключ); $h(k) = \{1, 1, 0, 3, 3, 4, 2, 5\}$. Для устранения коллизий построим табл. 6.6.

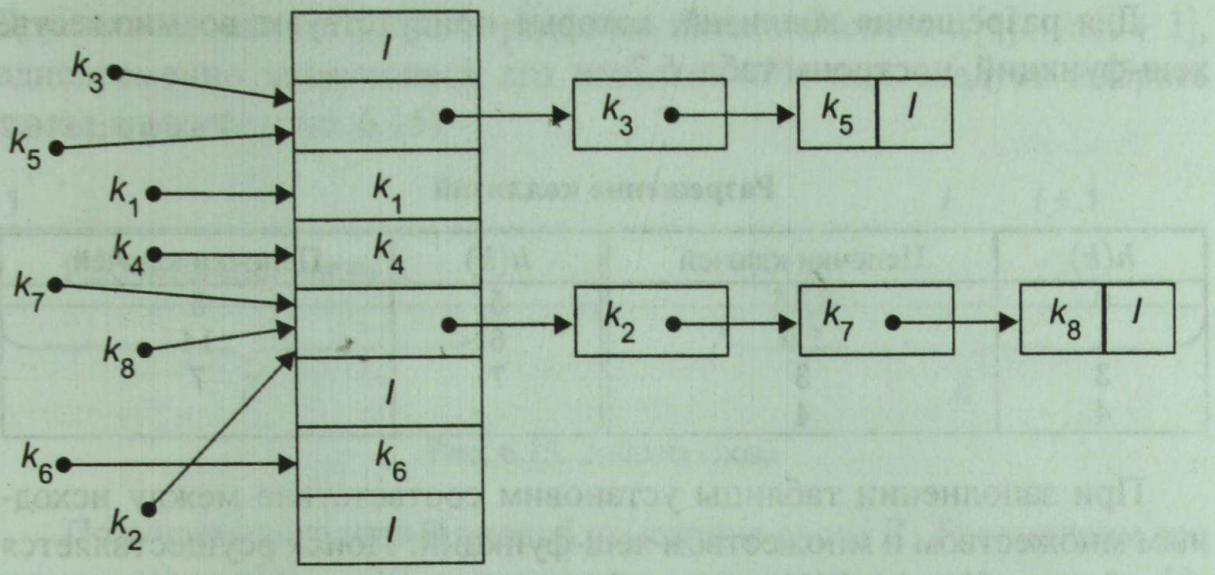


Рис. 6.14. Устранение коллизий методом цепочек

Таблица 6.6

Разрешение коллизий

$h(k)$	Цепочки ключей	$h(k)$	Цепочки ключей
0	6	3	3,9
1	7,13	4	4
2	8	5	5

Попарным сравнением множества хеш-функций и множества исходных ключей заполняем таблицу.

При этом хеш-функция указывает адрес, по которому следует отыскивать ключ. Например, если отыскивается ключ $K = 27$, тогда

$$h(k) = 27 \bmod 6 = 3.$$

Это значит, что ключ $K = 27$ может находиться только в строке, где $h(k) = 3$. Так как его там нет, то данный ключ отсутствует в исходном множестве.

Пример 2. Дано множество ключей

$$\{7,\ 1,\ 8,\ 5,\ 14,\ 9,\ 16,\ 3,\ 4\}.$$

Найти ключ $K = 14$.

Хеш-функция равна $h(k) = k \bmod m$, где $m = [16/2] = 8$ (так как 16 — максимальный ключ). Следовательно,

$$h(k) = \{7,\ 1,\ 0,\ 5,\ 6,\ 1,\ 0,\ 3,\ 4\}.$$

Для разрешения коллизий, которые присутствуют во множестве хеш-функций, построим табл. 6.7.

Таблица 6.7

Разрешение коллизий

$h(k)$	Цепочки ключей	$h(k)$	Цепочки ключей
0	8,16	5	5
1	1,9	6	14
3	3	7	7
4	4		

При заполнении таблицы установим соответствие между исходным множеством и множеством хеш-функций. Поиск осуществляется по таблице: $K = 14; h(k) = 14 \bmod 8 = 6$. Это значит, что ключ $K = 14$ может быть только в строке со значением $h(k) = 6$.

6.8. Алгоритмы поиска словесной информации

В настоящее время наличие сверхпроизводительных микропроцессоров и дешевизна электронных компонентов позволяют делать значительные успехи в алгоритмическом моделировании. Рассмотрим несколько алгоритмов обработки слов.

Алгоритм Кнута — Морриса — Пратта (КМП). Данный алгоритм получает на вход слово $X = x[1]x[2]\ldots x[n]$ и просматривает его слева направо, буква за буквой, заполняя при этом массив натуральных чисел $l[1]\ldots l[n]$, где $l[i]$ — длина слова $l(x[1]\ldots x[i])$. Таким образом, $l[i]$ есть длина наибольшего начала слова $x[1]\ldots x[i]$, одновременно являющегося его концом.

Пример. Используя алгоритм КМП, определить, является ли слово A подсловом слова B.

Решение. Применим алгоритм КМП к слову $A\#B$, где $\#$ — специальная буква, не встречающаяся ни в A, ни в B. Слово A является подсловом слова B тогда и только тогда, когда среди чисел в массиве l будет число, равное длине слова A.

Предположим, что первые i значений $l[1]\ldots l[i]$ уже найдены. Читается очередная буква слова, т.е. $x[i+1]$, и вычисляется $l[i+1]$.

Другими словами, нужно определить начала Z слова $x[1]\ldots x[i+1]$, одновременно являющиеся его концами. Из них следует выбрать самое длинное (рис. 6.15).

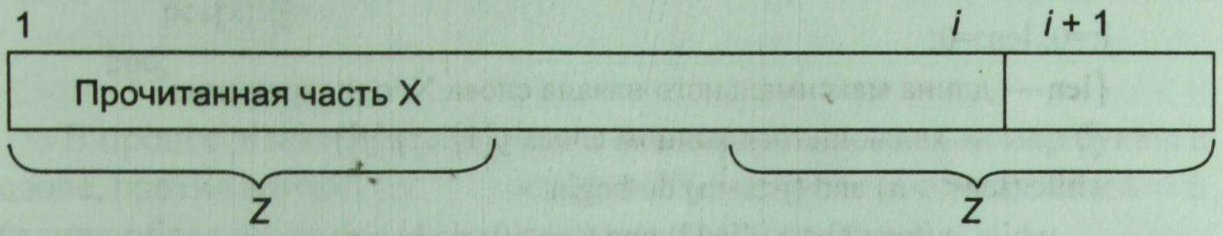


Рис. 6.15. Анализ слова

Получаем следующий способ отыскания слова Z. Рассмотрим все начала слова $x[1]\ldots x[i]$, являющиеся одновременно его концами. Из них выберем подходящие — те, за которыми следует буква $x[i+1]$. Из подходящих выберем самое длинное. Приписав в его конец $x[i+1]$, получим искомое слово Z. Теперь пора воспользоваться сделанными приготовлениями и вспомнить, что все слова, являющиеся одновременно началами и концами данного слова, можно получить повторными применениями к нему функции l.

Получим следующий фрагмент программы.

```
i:=1; l[1]:=0;
{таблица l[1]..l[i] заполнена правильно}
while i <> n do begin
    len:= l[i];
    {len — длина начала слова x[1]...x[i], которое является
        его концом; все более длинные начала
        оказались неподходящими}
    while (x[len+1]<>x[i+1]) and (len>0) do begin
        {начало не подходит, применяем к нему функцию l}
        len:=l[len];
    end;
    {нашли подходящее слово или убедились в его отсутствии}
    if x[len+1]=x[i+1] do begin
        {x[1]..x[len] — самое длинное подходящее начало}
        l[i+1]:=len+1;
    end else begin
        {подходящих нет}
        l[i+1]:= 0;
    end;
    i:=i+1;
end;
```

Представим алгоритм, который позволяет проверить, является ли слово $X = x[1]\ldots x[n]$ подсловом слова $Y = y[1]\ldots y[m]$.

Решение. Вычисляем таблицу $l[1]\ldots l[n]$, как раньше.

```
j:=0; len:=0;
{len — длина максимального начала слова X, одновременно
        являющегося концом слова y[1]...y[j]}
while (len<>n) and (j<>m) do begin
    while (x[len+1]<>y[j+1]) and (len>0) do begin
        {начало не подходит, применяем к нему функцию l}
        len: = l[len];
    end;
    {нашли подходящее слово или убедились в его отсутствии}
    if x[len+1]=y[j+1] do begin
        { x[1]...x[len] — самое длинное подходящее начало}
        len:=len+1;
    end else begin
        {подходящих нет}
        len:=0;
    end;
    j:=j+1;
end;
{если len = n, слово X встретилось; иначе мы дошли до конца
        слова Y, так и не встретив X}
```

Алгоритм Бойера — Мура (БМ). Этот алгоритм делает то, что на первый взгляд кажется невозможным: в типичной ситуации он читает лишь небольшую часть всех букв слова, в котором ищется заданный образец. Пусть, например, отыскивается образец *abcd*. Посмотрим на четвертую букву слова: если, к примеру, это буква *e*, то нет никакой необходимости читать первые три буквы. (В самом деле, в образце буквы *e* нет, поэтому он может начаться не раньше пятой буквы.)

Приведем самый простой вариант этого алгоритма, который не гарантирует быстрой работы во всех случаях. Пусть $x[1]\ldots x[n]$ — образец, который надо искать. Для каждого символа s найдем самое правое его вхождение в слово X, т.е. наибольшее k, при котором $x[k] = s$. Эти сведения будем хранить в массиве $\mathrm{pos}[s]$; если символ s вовсе не встречается, то будет удобно предположить $\mathrm{pos}[s] = 0$.

Решение.

```
Принять все pos[s] равными 0
for i:=1 to n do begin
    pos[x[i]]:=i;
end;
```

В процессе поиска будем хранить в переменной last номер буквы в слове, против которой стоит последняя буква образца. Вначале last = n (длина образца), затем last постепенно увеличивается.

```
last:=n;
{все предыдущие положения образца уже проверены}
while last <= m do begin {слово не кончилось}
    if x[m] <> y[last] then begin {последние буквы разные}
        last := last + (n - pos[y[last]]);
        {n - pos[y[last]] — минимальный сдвиг образца, при котором
            напротив y[last] встанет такая же буква в образце.
            Если такой буквы нет вообще, то сдвигаем
            на всю длину образца}
    end else begin
        {если нынешнее положение подходит, т.е. если
            x[i]...x[n] = y[last - n + 1]...y[last],
            то сообщить о совпадении}
        last:=last+1;
    end;
end;
```

Алгоритм Рабина. Этот алгоритм основан на простой идее. Представим себе, что в слове длиной m ищется образец длиной n. Вырежем окошко размером n и будем двигать его по входному слову. При этом проверяем, не совпадает ли слово в окошке с заданным образцом. Сравнивать по буквам долго. Вместо этого фиксируем некоторую функцию, определенную на словах длиной n. Если значения этой функции на слове в окошке и на образце различны, то совпадения нет. Только если значения одинаковы, нужно проверять совпадение по буквам.

Выигрыш при таком подходе состоит в следующем. Чтобы вычислить значение функции на слове в окошке, нужно прочесть все буквы этого слова. Так уж лучше их сразу сравнить с образцом. Тем не менее выигрыш возможен, так как при сдвиге окошка слово не

меняется полностью, а лишь добавляется буква в конце и убирается в начале. Заменим все буквы в слове и образце их номерами, представляющими собой целые числа. Тогда удобной функцией является сумма цифр. (При сдвиге окошка нужно добавить новое число и вычесть пропавшее.) Выберем некоторое число p (желательно простое) и некоторый вычет x по модулю p. Каждое слово длиной n представим как последовательность целых чисел (заменив буквы кодами). Эти числа будем рассматривать как коэффициенты многочлена степени $n-1$ и вычислим значение этого многочлена по модулю p в точке x. Это и будет одна из функций семейства (для каждой пары p и x получается, таким образом, своя функция). Сдвиг окошка на 1 соответствует вычитанию старшего члена (x^{n-1} следует вычислить заранее), умножению на x и добавлению свободного члена. Следующее соображение говорит в пользу того, что совпадения не слишком вероятны.

Пусть число p фиксировано и к тому же простое, а X и Y — два различных слова длиной n. Тогда им соответствуют различные многочлены (предполагаем, что коды всех букв различны — это возможно, если p больше числа букв алфавита). Совпадение значений функции означает, что в точке x эти два различных многочлена совпадают, т.е. их разность обращается в 0. Разность есть многочлен степени $n-1$ и имеет не более $n-1$ корней. Таким образом, если n много меньше p, то у случайного x мало шансов попасть в неудачную точку.

Контрольные вопросы

1. Что понимается под поиском?
2. Каковы особенности последовательного и бинарного поиска?
3. Каковы особенности интерполяционного и фибоначчиевого поиска?
4. Каковы особенности поиска по бинарному дереву?
5. Каковы особенности поиска по бору и хешированием?
6. В чем состоит методика анализа сложности алгоритмов поиска?
7. В чем заключается особенность алгоритмов поиска словесной информации?
8. Что такое коллизия?
9. Какой метод используется для разрешения коллизий?
10. Что определяет показатель сбалансированности узла дерева?
11. Назовите алгоритмы поиска словесной информации.

Глава 7
Итеративные и рекурсивные алгоритмы

Эффективным средством программирования для некоторого класса задач является рекурсия. С ее помощью можно решать сложные задачи численного анализа, комбинаторики, алгоритмов трансляции, операций над списковыми структурами и т.д. Программы в этом случае имеют небольшие объемы по сравнению с итерацией и требуют меньше времени на отладку.

Под ***рекурсией*** понимают способ задания функции через саму себя, например способ задания факториала в виде

$$n! = (n-1)! \cdot n.$$

В программировании под рекурсивной процедурой (функцией) понимают способ обращения процедуры (функции) к самой себе.

Под ***итерацией*** понимают результат многократно повторяемой какой-либо операции, например представление факториала в виде

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n.$$

Среди широкого класса задач удобно представлять с использованием рекурсивных процедур (функций) те задачи, которые сводятся на подзадачи того же типа, но меньшей размерности.

Общая методика анализа рекурсии содержит три этапа:

1. Параметризация задачи, заключающаяся в выделении различных элементов, от которых зависит решение, в частности размерности решаемой задачи. После каждого рекурсивного вызова размерность должна убывать.

2. Поиск тривиального случая и его решение. Как правило, это ключевой этап в рекурсии, размерность задачи при этом часто равна 0 или 1.

3. Декомпозиция общего случая, имеющая целью привести задачу к одной или нескольким задачам того же типа, но меньшей размерности.

Рассмотрим понятие итеративного и рекурсивного алгоритмов на примере вычисления факториала.

7.1. Итеративный алгоритм

Наиболее простой и естественной формой представления итеративного алгоритма при реализации на компьютере является его описание с использованием цикла.

Рассмотрим алгоритм итеративного алгоритма вычисления факториала $n!$

В соответствии с определением функции $n!$ имеем

$$n! = \begin{cases} 1, & \text{если} \quad n = 0; \\ 1 \cdot 2 \cdot \ldots \cdot n, & \text{если} \quad n \neq 0. \end{cases}$$

Вначале в зависимости от текущего значения происходит выбор способа вычисления $n!$. Если $n = 0$, то переменная *fctrl* принимает значение 1. Если $n \neq 0$, в цикле вычисляется произведение $1 \cdot 2 \cdot \ldots \cdot n$. Переменная i — это параметр цикла, который последовательно принимает значения 2, 3, 4 и т.д. до n включительно. Для каждого значения параметра цикла выполняется тело цикла:

$$fctrl := fctrl * i.$$

Последовательность итераций цикла для $n = 6$ показана в табл. 7.1.

Под итерацией цикла будем понимать выполнение тела цикла для конкретного значения параметра цикла.

Программа состоит из процедуры-функции FACTORIAL и основной программы. В основной программе происходит ввод значения N, вызов процедуры-функции и печать результата.

```
{Итеративное вычисление факториала}
PROGRAM FI;
VAR
FAC: LONGINT;
N: INTEGER;
{Функция вычисления факториала}
FUNCTION FACTORIAL (N: INTEGER): LONGINT;
VAR F: LONGINT;
I: INTEGER;
```

Таблица 7.1

Пошаговое вычисление факториала

Итерации		Состояние		
		i	$fctrl$	n
1-я итерация $i \le n \rightarrow$	$\begin{cases} i := 2 \\ fctrl := fctrl * i \end{cases}$	2	1	6
		2	2	6
2-я итерация $i \le n \rightarrow$	$\begin{cases} i := i + 1 \\ fctrl := fctrl * i \end{cases}$	3	2	6
		3	6	6
3-я итерация $i \le n \rightarrow$	$\begin{cases} i := i + 1 \\ fctrl := fctrl * i \end{cases}$	4	6	6
		4	24	6
4-я итерация $i \le n \rightarrow$	$\begin{cases} i := i + 1 \\ fctrl := fctrl * i \end{cases}$	5	24	6
		5	120	6
5-я итерация $i \le n \rightarrow$	$\begin{cases} i := i + 1 \\ fctrl := fctrl * i \end{cases}$	6	120	6
		6	720	6

```
  BEGIN
    IF (N=O) OR (N=1) THEN FACTORIAL:=1 ELSE
    BEGIN
      F:= 1;
      FOR I: = 2 TO N DO
        F: = F * I;
      FACTORIAL:=F;
    END;
  END;
  {Основная программа}
  BEGIN
    WRITELN ('ВВЕДИТЕ ЗНАЧЕНИЕ N')
    READLN (N);
    FAC:= FACTORIAL(N); {Вызов функции FACTORIAL}
    WRITELN(' ФАКТОРИАЛ =', FAC);
    READLN;
  END.
```

7.2. Рекурсивный алгоритм

Рекуррентные соотношения довольно часто встречаются в математических выражениях. Рекурсия в определении состоит в том, что определяемое понятие определяется через само это понятие. Рекурсия в вычислениях выступает в форме рекуррентных соотношений, которые показывают, как вычислить очередное значение, используя предыдущие.

Например, рекуррентное соотношение $x_i = x_{i-2} + x_{i-1}$, где $x_1 = 1$, $x_2 = 2$, задает правило вычисления так называемых чисел Фибоначчи.

Другим примером рекуррентных соотношений могут служить правила вычисления членов арифметической прогрессии

$$a_{n+1} = a_n + d,$$

где d — разность прогрессии, либо геометрической прогрессии

$$a_{n+1} = qa_n,$$

где q — коэффициент прогрессии.

Рекурсивное представление факториала имеет вид:

$$n! = (n-1)! \cdot n.$$

Проведем анализ рекурсивного вычисления факториала.

1. Параметризация. В данном случае имеется всего один параметр n — целое число.

2. Поиск тривиального случая. При $n = 0$ или $n = 1$ значение факториала равно 1, что соответствует выходу из рекурсии.

3. Декомпозиция общего случая: $n!$ вычисляется через меньшую размерность этой же задачи $(n-1)!$.

Покажем алгоритм вычисления факториала для $n = 4$ (рис. 7.1).

Сначала образуется так называемый рекурсивный фрейм 1 при $n = 4$. ***Фрейм*** — структура, содержащая некоторую информацию. Для этого фрейма отводится память и в нем фиксируются все значения переменных тела функции при $n = 4$. Отметим, что в рекурсивном фрейме фиксируются значения всех переменных функции, кроме глобальных.

Затем происходит вызов $\mathrm{Factorial}(n)$ при $n = 3$. Образуется фрейм 2, где фиксируются значения переменных тела функции при $n = 3$.

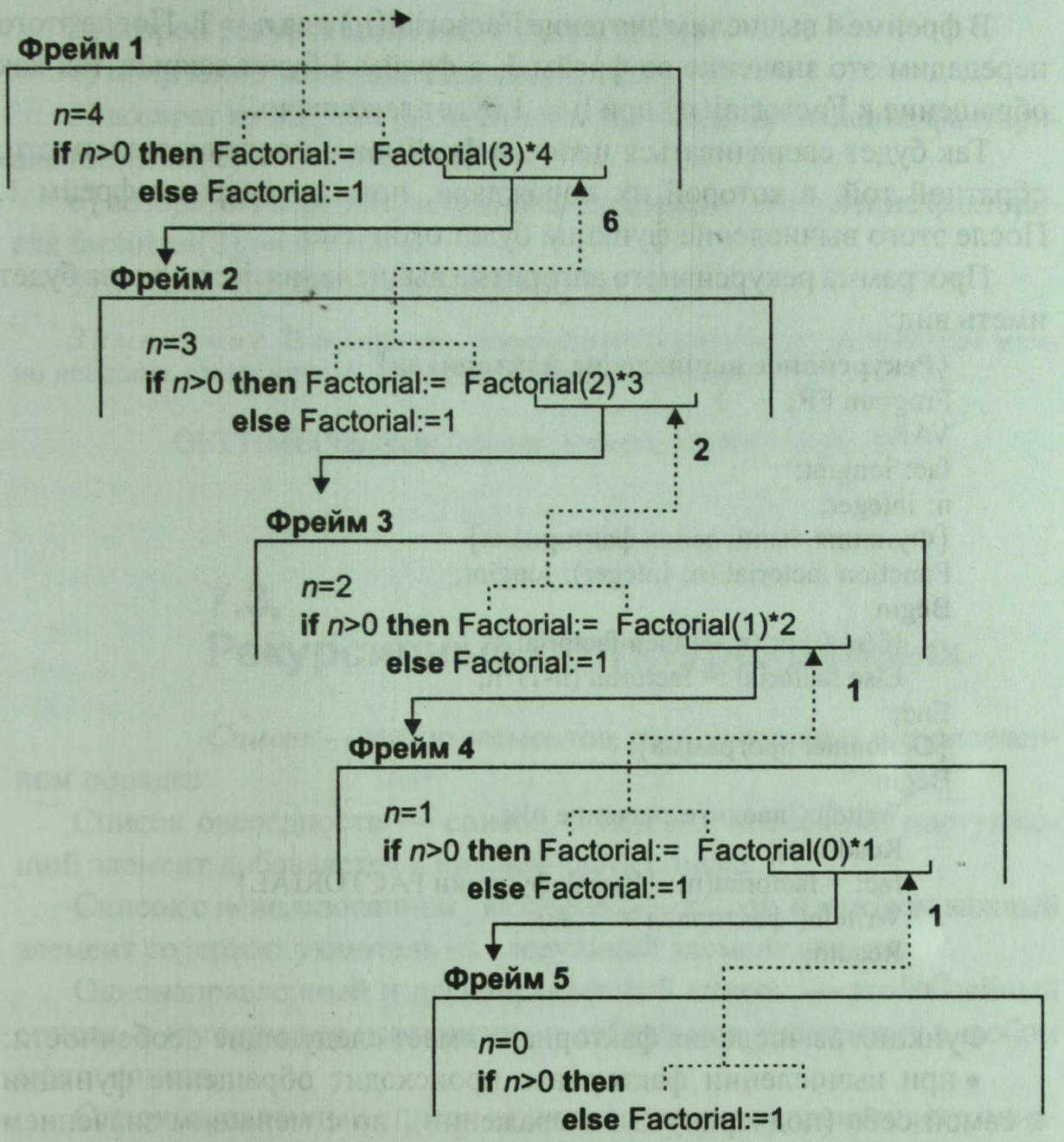


Рис. 7.1. Рекурсивное вычисление факториала

При этом фрейм 1 также хранится в памяти. Из фрейма 2 происходит обращение к Factorial(n) при $n = 2$. В результате этого обращения образуется фрейм 3, где фиксируются значения переменных тела функции при $n = 2$ и т.д. до тех пор, пока при очередном обращении к функции Factorial условие $n > 0$ не примет значение false.

Это произойдет в фрейме 5. В этом фрейме получим значение Factorial $= 1$ и передадим это значение в фрейм 4.

После этого фрейм 5 будет уничтожен, так как обращение Factorial(n) при $n = 0$ будет выполнено.

В фрейме 4 вычислим значение Factorial(n) для $n = 1$. После этого передадим это значение во фрейм 3, а фрейм 4 будет закрыт, так как обращение к Factorial(n) при $n = 1$ будет закончено.

Так будет сворачиваться цепочка фреймов в последовательности, обратной той, в которой их порождали, пока не свернем фрейм 1. После этого вычисление функции будет окончено.

Программа рекурсивного алгоритма вычисления факториала будет иметь вид:

```
{Рекурсивное вычисление факториала}
Program FR;
VAR
fac: longint;
n: integer;
{Функция вычисления факториала}
Function factorial (n: integer): longint;
Begin
    If (n=0) or (n=1) then factorial := 1
    Else factorial := factorial (n-1)*n;
End;
{Основная программа}
Begin
    Writeln('введите значение n');
    Readln(n);
    fac: = factorial(n); {Вызов функции FACTORIAL}
    Writeln('факториал = ', fac);
    Readln;
End.
```

Функция вычисления факториала имеет следующие особенности:

• при вычислении факториала происходит обращение функции к самой себе (подчеркнуто в выражении), но с меньшим значением аргумента $n - 1$ по сравнению с первым вызовом n:

$$\text{factorial} := \underline{\text{factorial}(n-1)} * n;$$

• при вычислении факториала не используется цикл, что является существенной особенностью рекурсивного алгоритма.

Рассмотрим последовательность действий при рекурсивном вычислении факториала для $n = 3$:

1) внешний вызов из основной программы factorial(3);

2) первый рекурсивный вызов factorial(2) в операторе

$$\text{factorial} := \underline{\text{factorial}(n-1)} * n,$$

где не происходят никакие вычисления — только вызов (подчеркнуто);

3) второй рекурсивный вызов factorial(1);

4) получение значения factorial(1) := 1;

5) возврат из второго рекурсивного вызова и вычисление факториала factorial(2) := 1 * 2 = 2;

6) возврат из первого рекурсивного вызова и вычисление факториала factorial(3) := 2 * 3 = 6;

7) возврат в основную программу fac := 6.

Замечание. В программу рекурсивного вычисления факториала можно добавить стандартную функцию текущего времени

GETTIME(Var Hour, Minute, Second, Sec100: WORD).

7.3. Рекурсивные структуры данных

Список — набор элементов, расположенных в определенном порядке.

Список очередности — список, в котором последний поступающий элемент добавляется к нижней части списка.

Список с использованием указателей — список, в котором каждый элемент содержит указатель на следующий элемент списка.

Однонаправленный и двунаправленный список — это линейный список, в котором все исключения и добавления происходят в любом месте списка.

Однонаправленный список отличается от двунаправленного списка только связью, т.е. в однонаправленном списке можно перемещаться только в одном направлении (из начала в конец), а в двунаправленном — в любом (рис. 7.2).

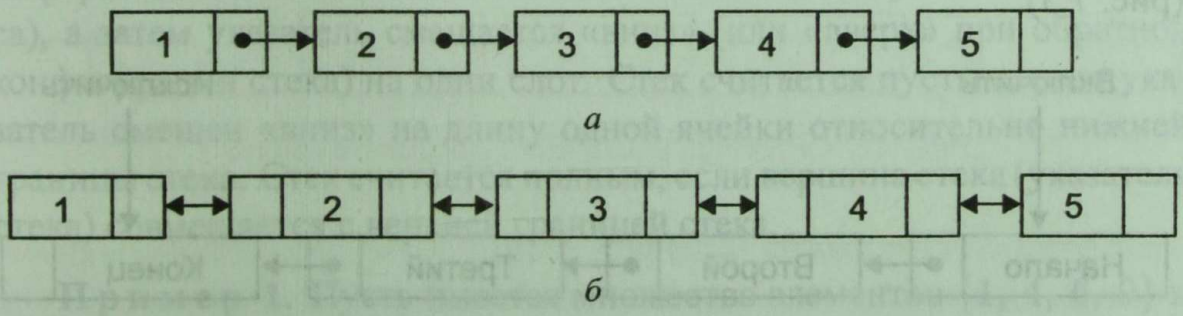


Рис. 7.2. Однонаправленный и двунаправленный списки

На рис. 7.3 и 7.4 показано, как добавляется и удаляется элемент из двунаправленного списка. При добавлении нового элемента (обозначен N) связь от 3 идет к N, а от N к 4, а связь между 3 и 4 удаляется.

В однонаправленном списке структура добавления и удаления такая же, только связь между элементами односторонняя.

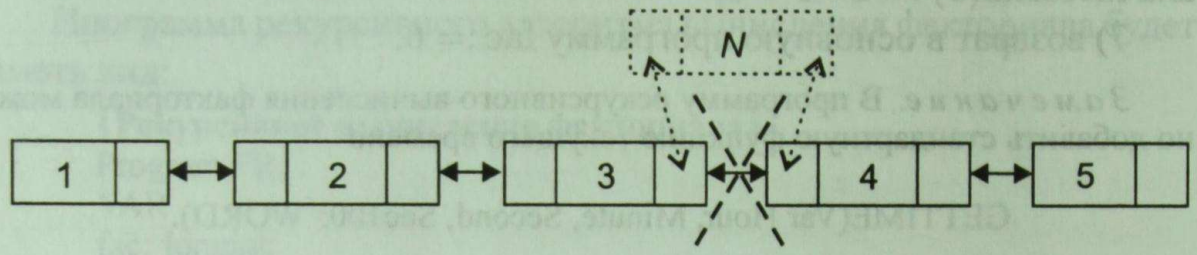


Рис. 7.3. Добавление элемента в список

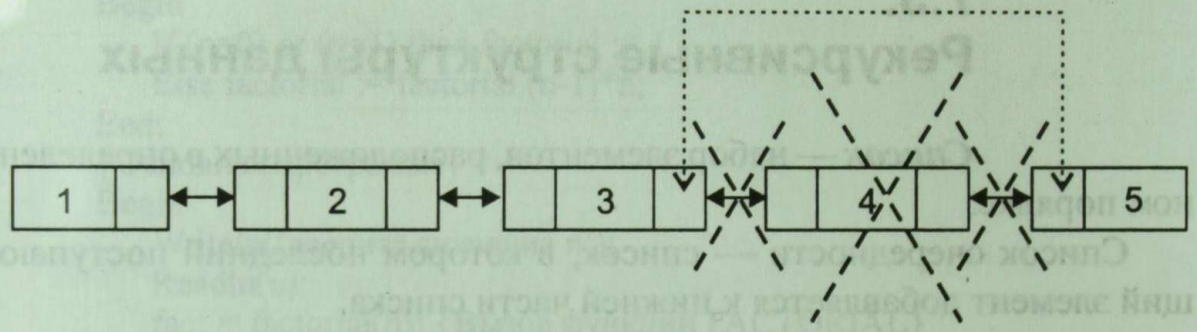


Рис. 7.4. Удаление элемента из списка

Очередь — тип данных, при котором новые данные располагаются следом за существующими в порядке поступления; поступившие первыми данные при этом обрабатываются первыми.

Очередь иногда называют циклической памятью или списком типа FIFO («first-in-first-out» — «первым пришел — первым обслуживается»). Другими словами, у очереди есть голова и хвост.

В очереди новый элемент добавляется только с одного конца (рис. 7.5).

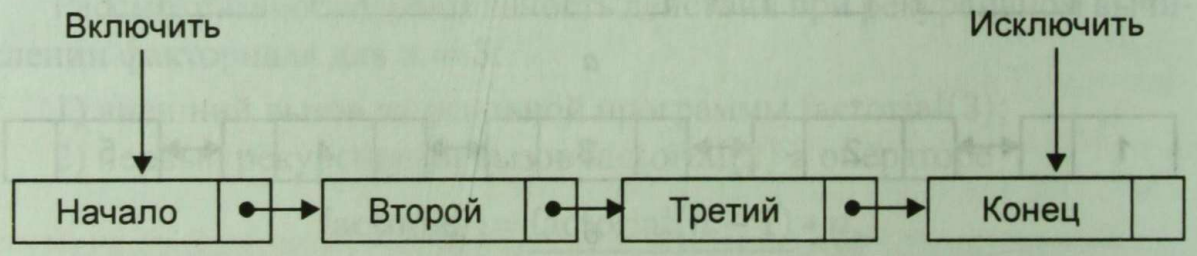


Рис. 7.5. Структура очереди

Удаление элемента происходит на другом конце. В данном случае это может быть только четвертый элемент. Очередь — по сути однонаправленный список, только добавление и исключение элементов происходит на концах списка.

Стек — линейный список, в котором все включения и исключения делаются в одном конце списка. Стек называют (push-down) списком, реверсивной памятью, гнездовой памятью, магазином, списком типа LIFO («last-in-first-out» — «последним пришел — первым обслуживается»). Стек — часть памяти ОЗУ компьютера, которая предназначается для временного хранения байтов, используемых микропроцессором. Действия со стеком производятся при помощи регистра указателя стека. Любое повреждение этой части памяти приводит к фатальному сбою.

Логическая структура стека представлена на рис. 7.6.

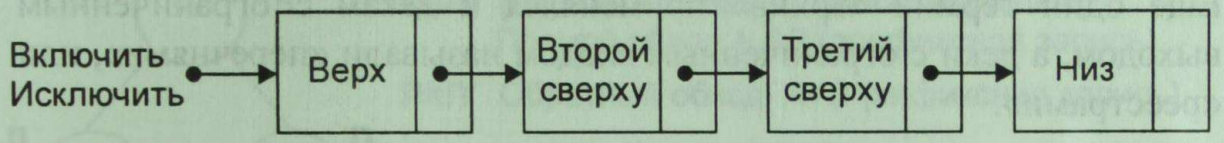


Рис. 7.6. Структура стека

Важнейшие операции доступа к стеку — включение элементов и исключение элементов — осуществляются с вершины стека, причем в каждый момент для исключения или включения доступен один элемент, находящийся на вершине стека.

Вершина стека адресуется с помощью специального указателя. Для включения нового элемента в стек указатель сначала перемещается «вверх» (возможна и другая конфигурация стека, когда стек «надстраивается снизу») на длину слота, или ячейки, а затем по значению указателя (индекса) в стек помещается информация о новом элементе. При исключении элемента из стека сначала прочитывается информация об исключаемом элементе по значению указателя (индекса), а затем указатель смещается «вниз» (или «вверх» при обратной конфигурации стека) на один слот. Стек считается пустым, если указатель смещен «вниз» на длину одной ячейки относительно нижней границы стека. Стек считается полным, если вершина стека (указатель стека) совмещается с верхней границей стека.

Пример 1. Пусть имеется множество элементов $\{1, 4, 6, 9\}$ и к стеку применена последовательность операций: (I, I, O, I, O, O,

I, *O*), где символ *I* означает запись элемента в стек, а символ *O* — считывание элемента из стека. Этапы выполнения операций показаны на рис. 7.7.

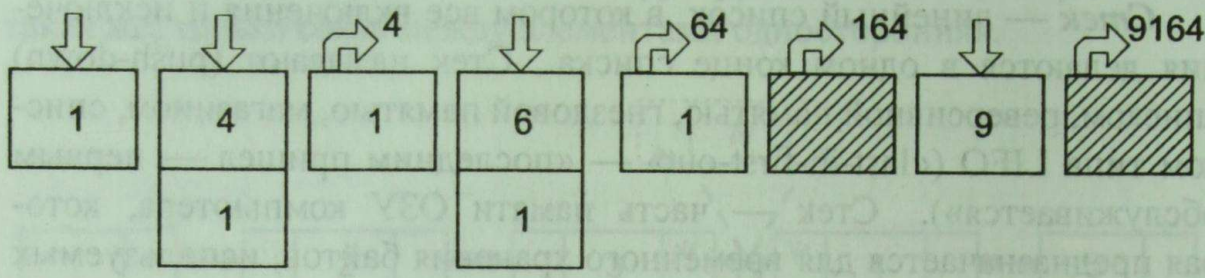


Рис. 7.7. Этапы выполнения операций

Дек (стек с двумя концами) — линейный список, в котором все включения и исключения делаются на обоих концах списка (рис. 7.8). Еще один термин «архив» применялся к декам с ограниченным выходом, а деки с ограниченным входом называли «перечнями», или «реестрами».

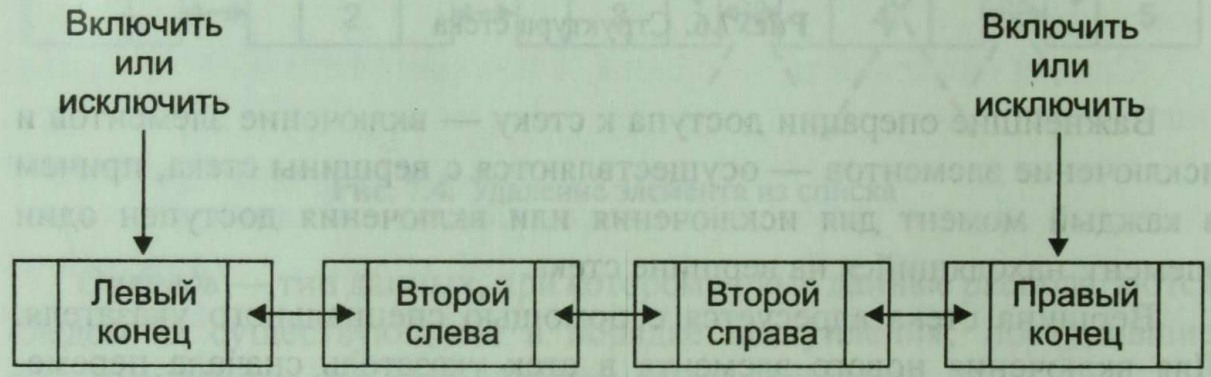


Рис. 7.8. Структура дека

Циклически связанный список обладает той особенностью, что связь его последнего узла идет назад к первому узлу списка (рис. 7.9).

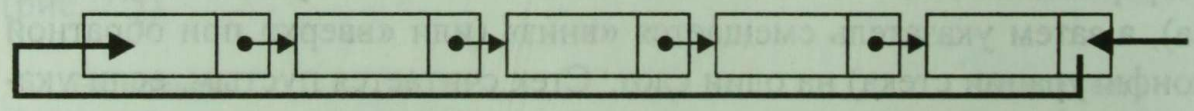


Рис. 7.9. Структура циклически связанного списка

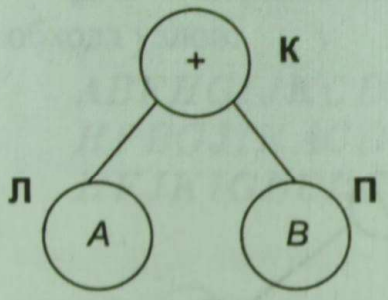
В этом случае можно получить доступ к любому элементу, находящемуся в списке, отправляясь от любой заданной точки. При этом не приходится различать в списке «последний» или «первый» узел.

7.4. Виды обхода бинарных деревьев

Набор способа обхода дерева позволяет ввести отношение порядка для узлов дерева.

Наиболее распространены три способа обхода узлов дерева (рис. 7.10), которые получили следующие названия:

- обход в направлении слева направо (обратный порядок, инфиксная запись);
- сверху вниз (прямой порядок, префиксная запись);
- снизу вверх (концевой порядок, постфиксная запись).



КЛП (Корень – Левая – Правая вершины)
Прямой обход + *AB* (префиксная запись)

ЛКП Обратный обход *A+B* (инфиксная запись)

ЛПК Концевой обход *AB+* (постфиксная запись)

Рис. 7.10. Три различных обхода по бинарному дереву и соответствующие формы

В результате обхода дерева, приведенного на рис. 7.11, порождаются следующие последовательности прохождения узлов:

abcdefghi (прямой порядок);
cdbfhigea (концевой порядок).

Рекурсивно можно представить не только алгоритм решения задачи, но и обрабатываемую информацию.

Пример. Пусть задано арифметическое выражение

$$(((A-B)*C)+(D/(E^F))).$$

Возможно описание этого арифметического выражения с помощью бинарного дерева — рис. 7.12.

Для прохождения этого дерева в прямом порядке начинаем с корня (узел +). Затем пройдем в прямом порядке левое поддерево (с корнем, помеченным $*$) и далее правое поддерево в прямом порядке

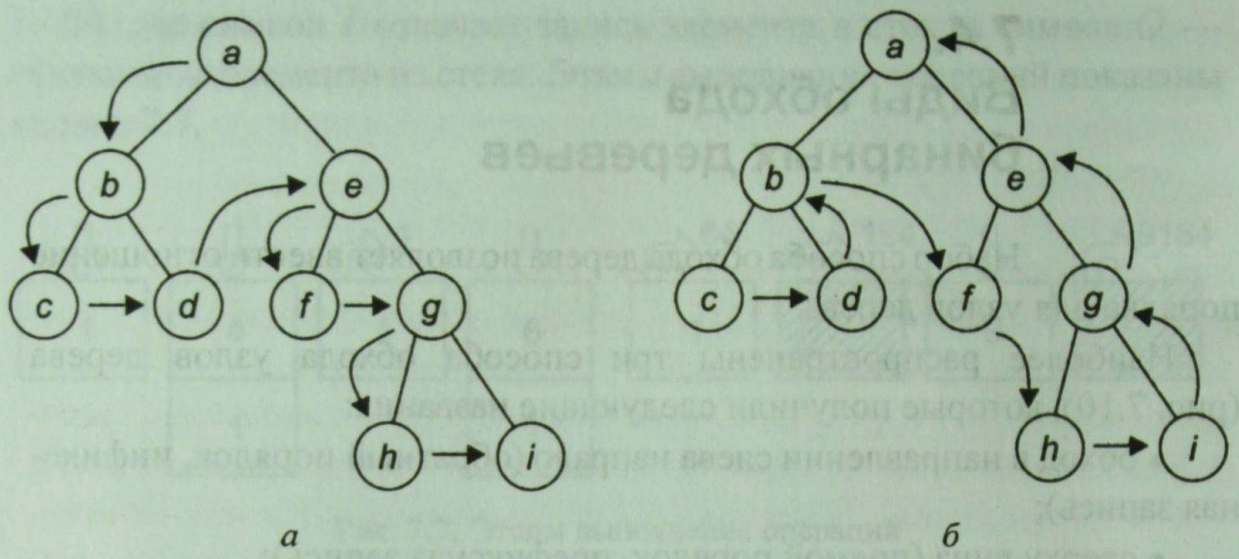


Рис. 7.11. Способы обхода бинарного дерева

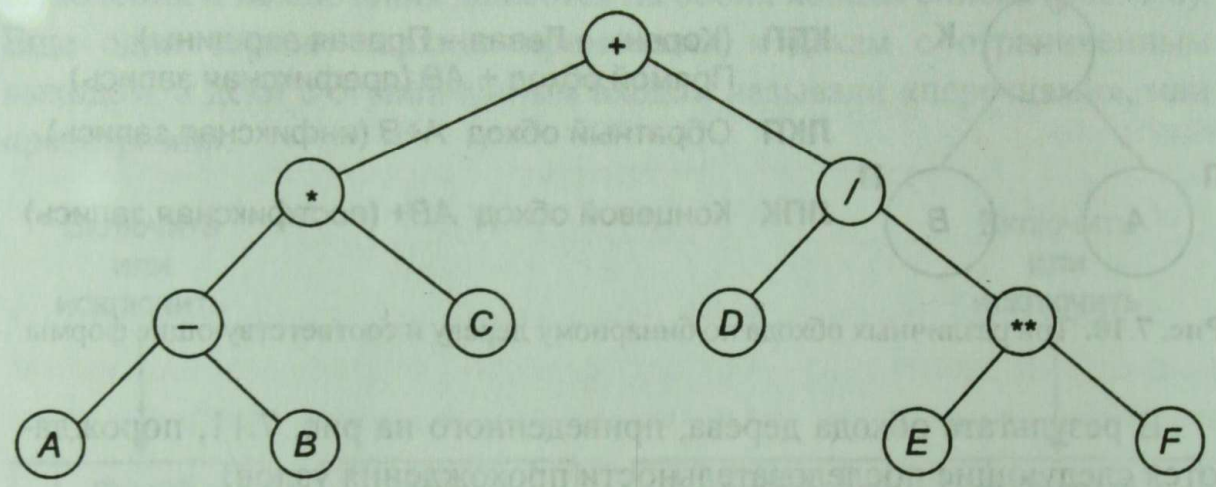


Рис. 7.12. Бинарное дерево для описания арифметического выражения

(с корнем, помеченным /). Прохождение левого поддерева в прямом порядке начинается с корня (помеченного $*$), затем следует прямое прохождение его левого поддерева (порождающее последовательность $-AB$) и далее правого поддерева (последовательность C). Поэтому вся последовательность, порождаемая прохождением левого поддерева основного корня, будет $*-ABC$. Аналогично прохождение в прямом порядке правого поддерева с корнем, помеченным /, порождает последовательность $/D**EF$. Таким образом, прохождение всего дерева в прямом порядке порождает последовательность

$$\underset{К}{+} \quad \underset{Л}{*-ABC} \quad \underset{П}{/D**EF}.$$

Прохождение этого дерева в обратном и концевом порядках порождает соответственно последовательности:

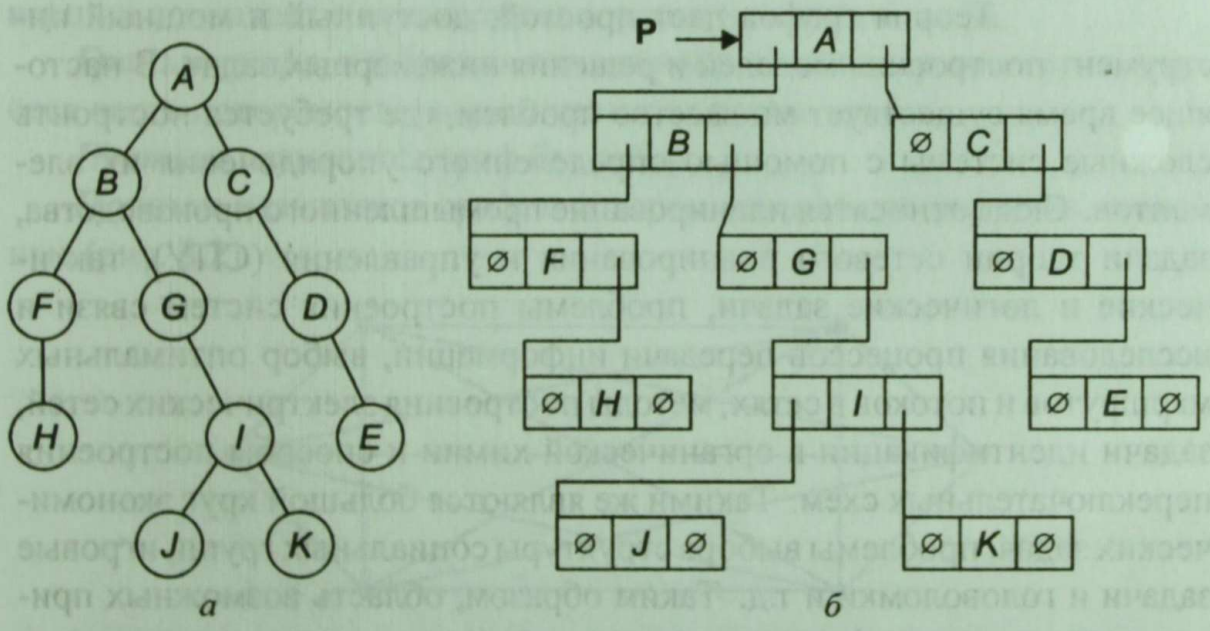
$$\frac{A-B*C}{Л} \quad \frac{+}{K} \quad \frac{D/E**F}{П} \quad \text{(инфиксная);}$$

$$\frac{AB-C*}{Л} \quad \frac{DEF**/}{П} \quad \frac{+}{K} \quad \text{(постфиксная).}$$

Заметим, что во всех трех выражениях порядок вхождения переменных совпадает; меняется только порядок знаков операций. При этом ни одно из этих выражений не имеет скобок, и, таким образом, если не заданы правила приоритета, значение приведенного выше выражения в инфиксной форме нельзя вычислить однозначно.

Для бинарного дерева на рис. 7.13 имеем следующий порядок обхода узлов:

ABFHGIJKCDE — для прямого обхода;
HFBGJIKACDE — для обратного обхода;
HFJKIGBEDCA — для концевого обхода.



а *б*

Рис. 7.13. Пример логической структуры (*а*) и спискового представления (*б*) бинарного дерева

Контрольные вопросы

1. Что понимается под рекурсией и итерацией в математике?
2. Каковы особенности итеративного алгоритма?

3. Каковы особенности рекурсивного алгоритма?
4. В чем состоит методика анализа рекурсивного алгоритма?
5. В каких случаях целесообразно использовать рекурсивный или итеративный алгоритм?
6. Приведите примеры итерации и рекурсии.
7. Все ли языки программирования дают возможность рекурсивного вызова процедур?
8. Укажите виды обхода бинарных деревьев.
9. Приведите пример рекурсивной структуры данных.
10. Что такое указатели и динамические переменные в алгоритмических языках?

Глава 8
Основные определения теории графов

Теория графов дает простой, доступный и мощный инструмент построения моделей и решения инженерных задач. В настоящее время существует множество проблем, где требуется построить сложные системы с помощью определенного упорядочения их элементов. Сюда относятся планирование промышленного производства, задачи теории сетевого планирования и управления (СПУ), тактические и логические задачи, проблемы построения систем связи и исследования процессов передачи информации, выбор оптимальных маршрутов и потоков в сетях, методы построения электрических сетей, задачи идентификации в органической химии и способы построения переключательных схем. Такими же являются большой круг экономических задач, проблемы выбора структуры социальных групп, игровые задачи и головоломки и т.д. Таким образом, область возможных применений теории графов очень широка.

Граф (сеть) $G = (V, E)$ состоит из конечного непустого множества m вершин ($m \geq 1$) и конечного множества n неупорядоченных пар элементов (u, v) ($n \geq 0$), называемых ребрами (рис. 8.1).

Две вершины u и v являются *смежными*, если в графе G существует ребро (u, v); в противном случае u и v *независимы*. Про ребро (u, v) также говорят, что оно *инцидентно* вершинам u и v.

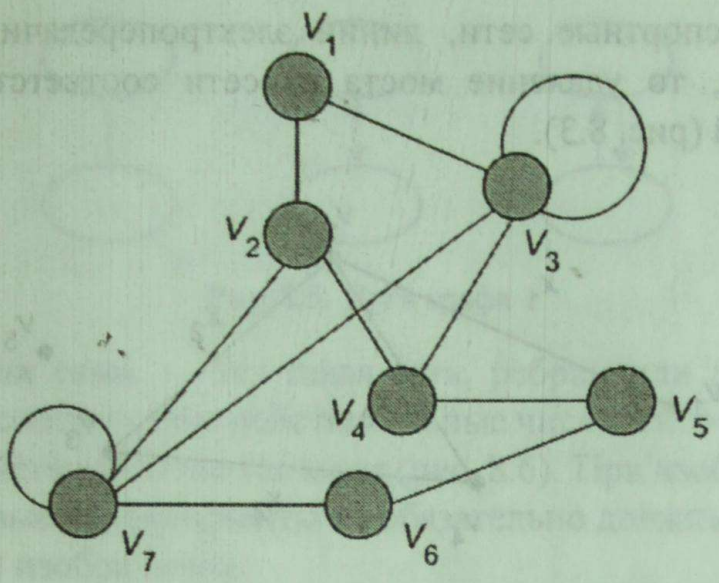


Рис. 8.1. Графовая структура

Приведем основные определения.

Граф называется ***вырожденным***, если у него нет ребер.

Граф называется ***связным***, если для любых двух вершин существует путь, соединяющий эти вершины.

Подграф называется ***остовным подграфом***, если множество его вершин совпадает с множеством вершин исходного графа.

Гранью графа, изображенного на некоторой поверхности, называется часть поверхности, ограниченная ребрами графа.

Пустым называется граф без ребер;

Полным называется граф, в котором каждые две вершины смежные (рис. 8.2).

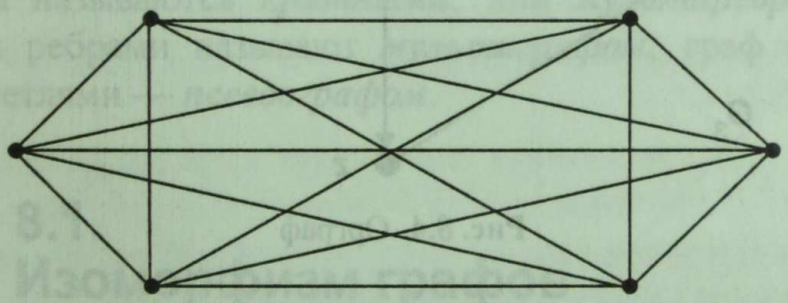


Рис. 8.2. Полный граф

Граф называется полным, если каждые две различные вершины его соединены одним и только одним ребром.

Мост — точка сочленения графа, удаление которой (и всех инцидентных ей ребер) увеличивает число компонентов связности. Например, если сеть поставлена в соответствие коммуникационным

линиям (транспортные сети, линии электропередачи, телефонные линии и т.п.), то удаление моста из сети соответствует разрыву коммуникаций (рис. 8.3).

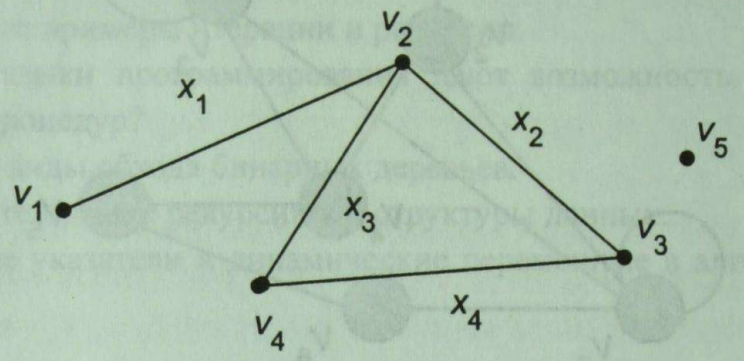


Рис. 8.3. Висячая v_1 и изолированная v_5 вершины

Ребро, соединяющее вершину саму с собой, называется ***петлей***.

Если сеть состоит из конечного множества вершин и множества упорядоченных пар $[u, v]$ различных вершин, то такая сеть называется ***ориентированной*** (***орграфом***) (на рис. 8.4 $V(G_3) = \{u, v, w, x, y, z\}$ и $E(G_3) = \{[u, v], [v, w], [v, x], [x, w], [w, y], [w, z], [z, v]\}$.

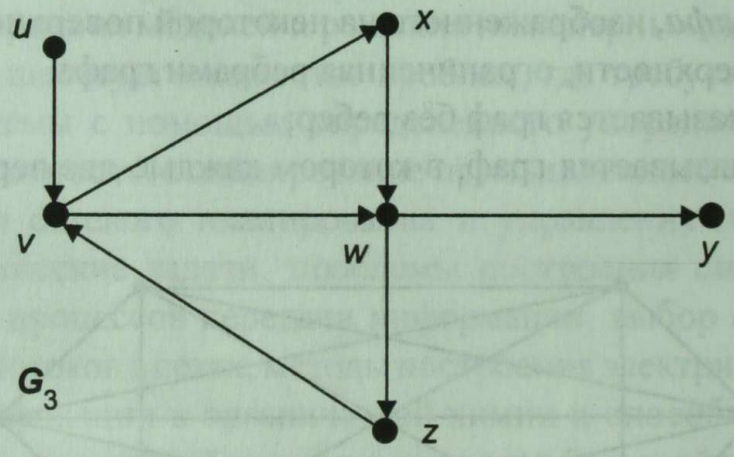


Рис. 8.4. Орграф

При работе с ориентированной сетью $G = (V, E)$ элементы V обычно называют ***узлами***, а элементы E — ***дугами***. Если $[u, v]$ — дуга ориентированной сети, то мы говорим, что u — левый сосед v, а v — правый сосед u.

В большинстве применений можно без потери смысла заменить ненаправленную дугу на двунаправленную, а двунаправленную — на две однонаправленные дуги, например так, как показано на рис. 8.5.

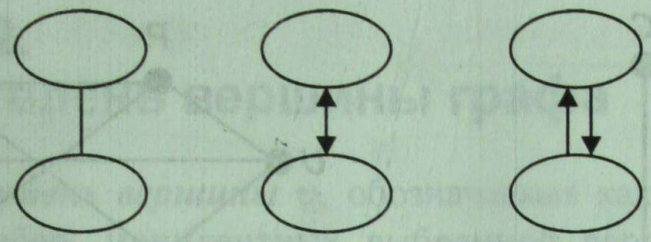


Рис. 8.5. Дуги графа

Взвешенная сеть — это такая сеть, ребрам или дугам которой поставлены в соответствие действительные числа или величины, принимающие действительные значения (рис. 8.6). При изображении сети веса или весовые коэффициенты не обязательно должны соответствовать масштабу изображения.

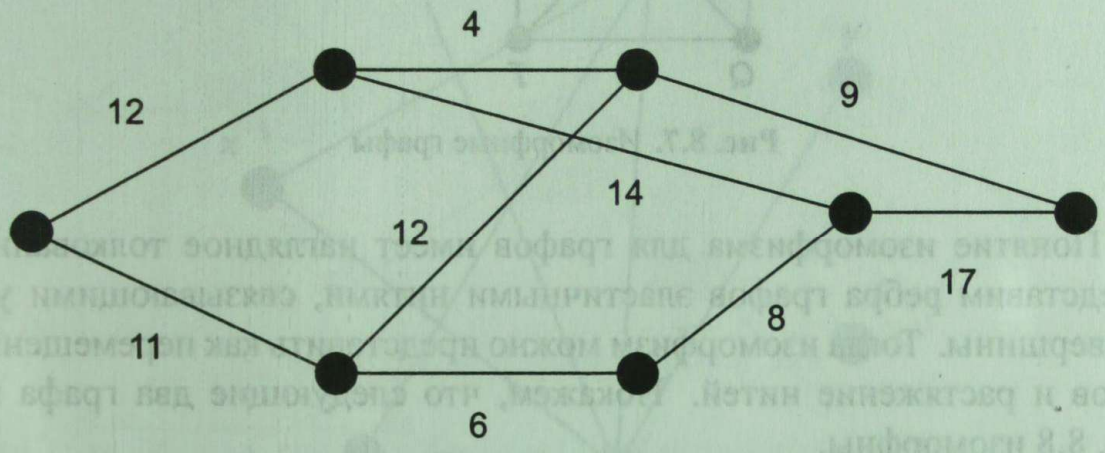


Рис. 8.6. Взвешенная сеть

Если две вершины соединяются более чем одним ребром, то такие ребра называются ***кратными***, или ***мультиребрами***. Граф с кратными ребрами называют ***мультиграфом***, граф с кратными ребрами и петлями — ***псевдографом***.

8.1. Изоморфизм графов

Два графа называют ***изоморфными***, если существует взаимно-однозначное соответствие между их вершинами, а именно: число ребер, соединяющих любые две вершины одного графа равно числу ребер, соединяющих соответствующие вершины другого графа. Три графа изоморфны (рис. 8.7) при соответствии:

$$A \leftrightarrow P; \quad B \leftrightarrow R; \quad C \leftrightarrow T; \quad D \leftrightarrow Q; \quad E \leftrightarrow S; \quad F \leftrightarrow U.$$

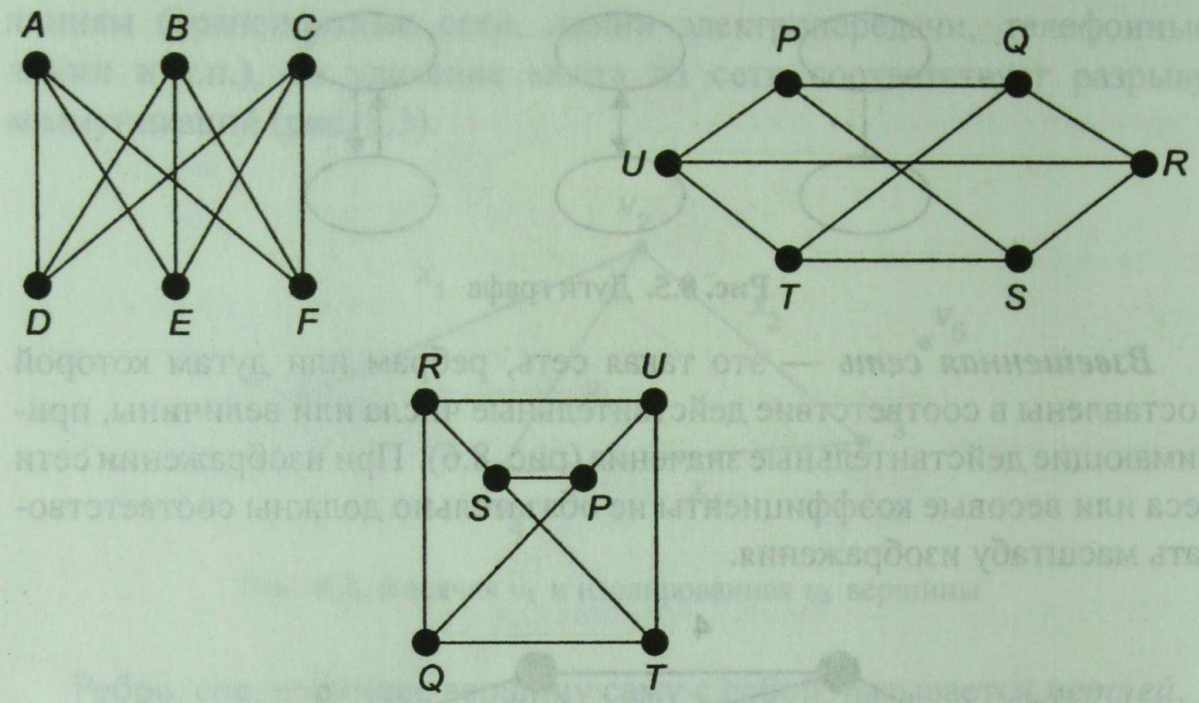


Рис. 8.7. Изоморфные графы

Понятие изоморфизма для графов имеет наглядное толкование. Представим ребра графов эластичными нитями, связывающими узлы-вершины. Тогда изоморфизм можно представить как перемещение узлов и растяжение нитей. Покажем, что следующие два графа на рис. 8.8 изоморфны.

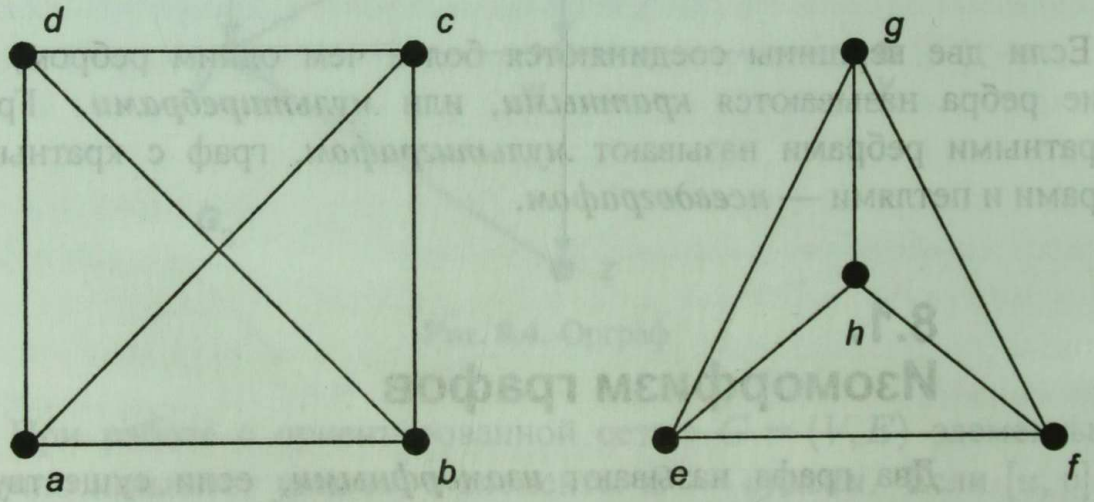


Рис. 8.8. Изоморфные графы

Действительно, отображение $a \leftrightarrow e$, $b \leftrightarrow f$, $c \leftrightarrow g$, $d \leftrightarrow h$, являющееся изоморфизмом, легко представить как модификацию первого графа, передвигающую вершину d в центр рисунка.

8.2. Степень вершины графа

Степень вершины v, обозначаемая как d_v или $\deg(v)$, равна числу ребер, инцидентных выбранной вершине, т.е. $d_v = \deg(v) = |N(v)|$. Таким образом, если вершины в G_1 (рис. 8.9) расположить в порядке w, v, x, z, u, y, t, то их соответствующие степени равны 4, 4, 2, 2, 1, 1, 0, т.е. $d_w = 4$, $d_v = 4$, $d_x = 2$ и т.д. При определении степени вершины графа петля учитывается дважды.

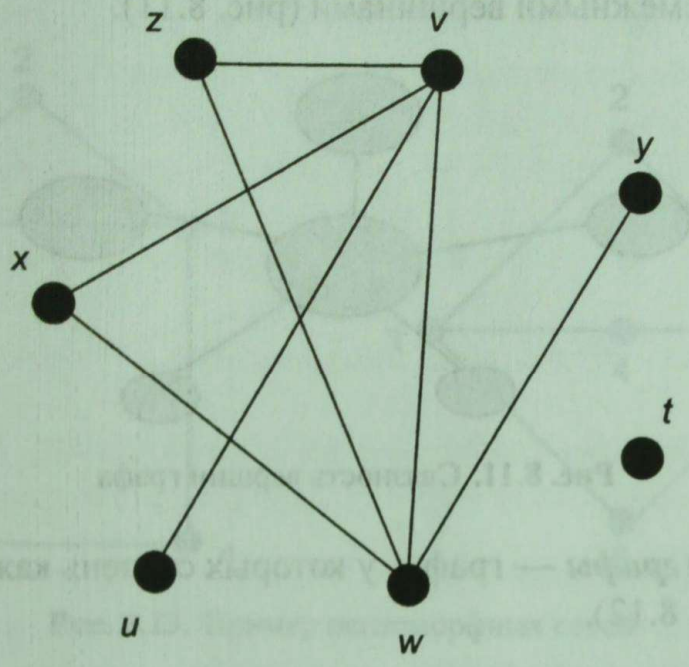


Рис. 8.9. Исходный граф

На рис. 8.9 вершины u и y тупиковые, а вершина t — изолированная.

Вершина v называется ***тупиковой вершиной***, если $\deg(v) = 1$; если $\deg(v) = 0$, то говорят, что v — ***изолированная вершина***.

Граф называется K-связным, если каждая его вершина связана с K другими вершинами; при этом говорят о слабо- и сильносвязных графах (рис. 8.10).

Иногда связность определяют как характеристику не каждой, а одной (произвольной) вершины. Тогда появляются определения типа: граф называется K-связным, если хотя бы одна его вершина связана с K другими вершинами.

В ряде случаев связность определяют как экстремальное значение количественной характеристики. Например, граф является K-связ-

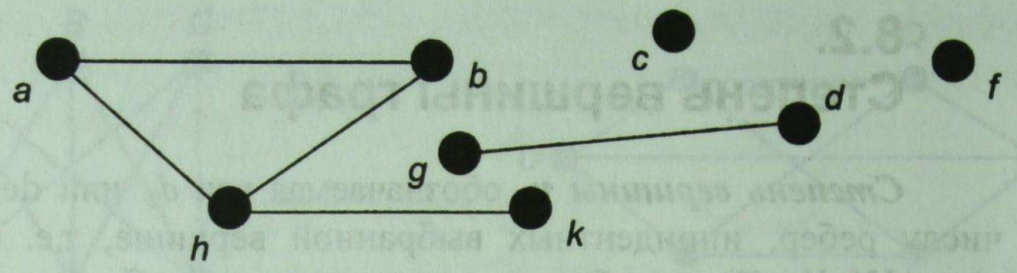


Рис. 8.10. Несвязный граф

ным, если в графе существует хотя бы одна вершина, связанная с K смежными вершинами и не существует ни одной вершины, связанной с более чем K смежными вершинами (рис. 8.11).

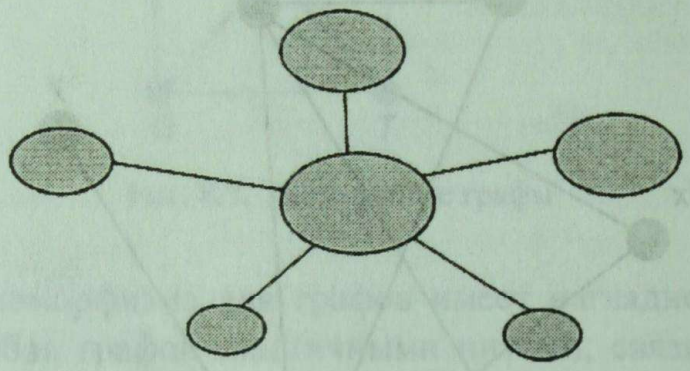


Рис. 8.11. Связность вершин графа

Регулярные графы — графы, у которых степень каждой вершины одинакова (рис. 8.12).

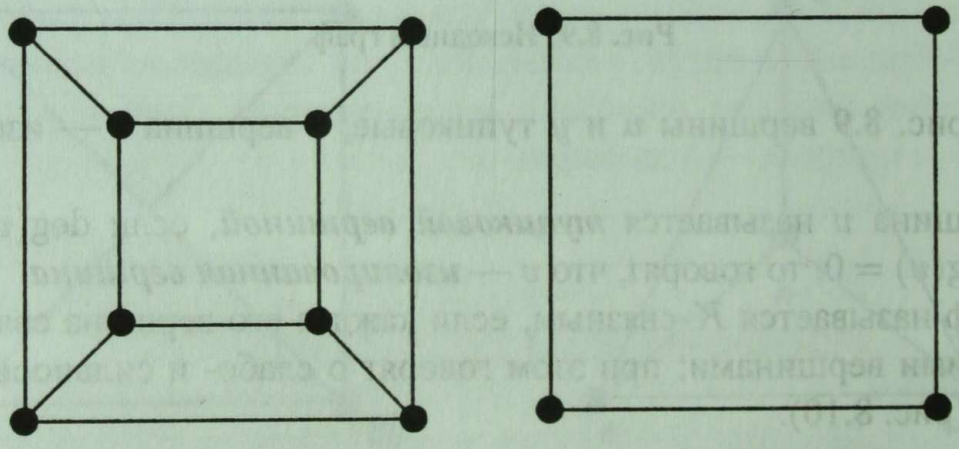


Рис. 8.12. Регулярные графы: *а* — степени 3; *б* — степени 2

Введем в рассмотрение понятие — ***инвариант сети***. Инвариантом сети G называется параметр, имеющий одно и то же значение

для всех сетей, изоморфных G. Среди самых очевидных инвариантов отметим следующие:

- число вершин;
- число ребер;
- число компонент;
- последовательность степеней, т.е. список степеней вершин в убывающем порядке значений.

Например, для двух пятивершинных сетей на рис. 8.13 все четыре из перечисленных выше инвариантов совпадают, но эти сети не изоморфны.

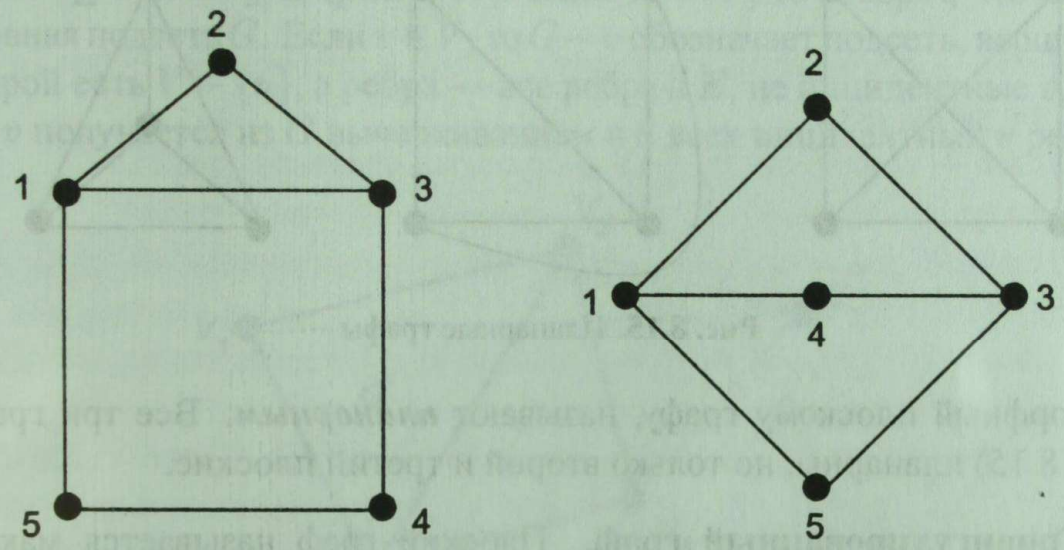


Рис. 8.13. Пример неизоморфных сетей

Двудольный граф. Граф называется двудольным, если множество его вершин можно разбить на два непересекающихся подмножества V_1 и V_2 так, что каждое ребро графа G соединяет какую-либо вершину из V_1 с какой-либо вершиной из V_2. Такие графы обозначают $G(V_1, V_2)$. Двудольный граф можно определить в терминах раскраски его вершин двумя цветами (например, синим и красным). При этом граф называют двудольным, если каждую его вершину можно окрасить красным или синим цветом так, чтобы любое ребро имело один конец красный, другой синий (рис. 8.14).

В двудольном графе совсем не обязательно, чтобы каждая вершина из V_1 соединялась с каждой вершиной из V_2.

Плоский граф. Плоским графом называют граф, изображенный на плоскости так, что никакие его два ребра геометрически не пересекаются нигде, кроме инцидентной им обоим вершины. Граф,

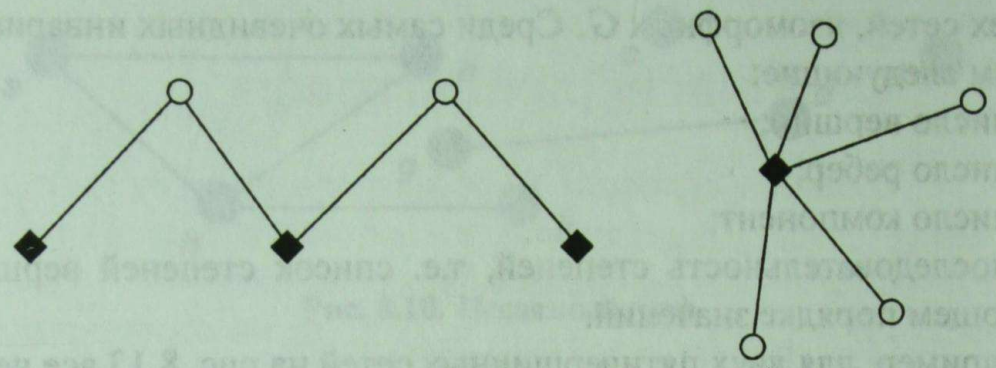


Рис. 8.14. Двудольные графы

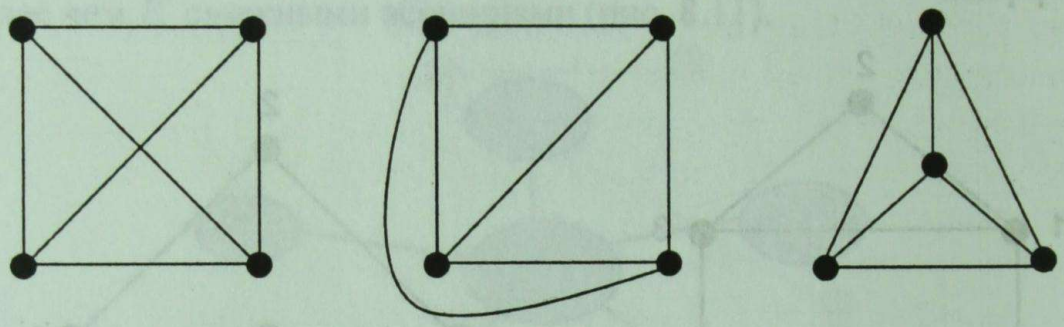
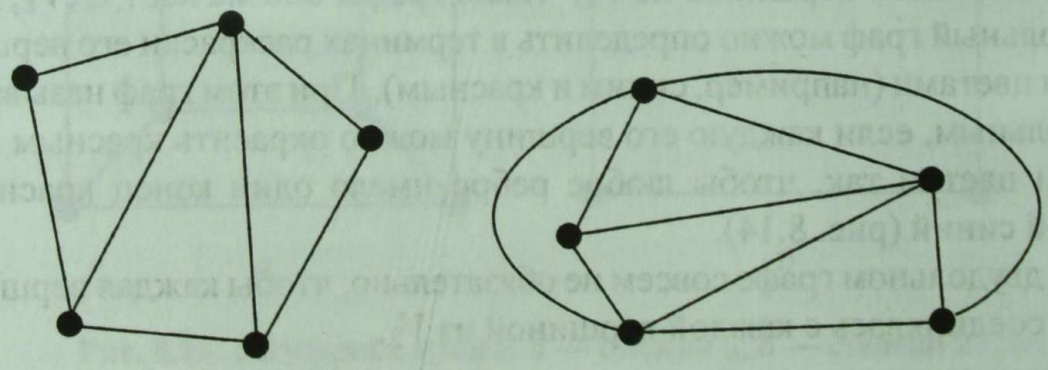


Рис. 8.15. Планарные графы

изоморфный плоскому графу, называют ***планарным***. Все три графа (рис. 8.15) планарны, но только второй и третий плоские.

Триангулированный граф. Плоский граф называется максимально плоским, если невозможно добавить к нему ни одного ребра так, чтобы полученный граф был плоским. Каждая грань в плоском представлении максимально плоского графа имеет три вершины. Максимально плоский граф называется еще триангулированным (рис. 8.16). Операция дополнения новых ребер, в результате которой



а *б*

Рис. 8.16. Плоский (*а*) и триангулированный (*б*) графы

в плоском представлении каждая грань имеет ровно три вершины, называется ***триангуляцией графа***.

Примечание. Существует только один триангулированный граф с четырьмя вершинами и только один с пятью вершинами.

8.3. Понятие подграфа

Сеть $G' = (V', E')$ является подсетью сети $G = (V, E)$, если $V' \subseteq V$ и $E' \subseteq E$ (рис. 8.17). Если $V' = V$, то говорят, что G' — остовная подсеть G. Если $v \in V$, то $G - v$ обозначает подсеть, вершины которой есть $V - \{v\}$, а ребра — все ребра в E, не инцидентные v, т.е. $G - v$ получается из G вычеркиванием v и всех инцидентных v ребер.

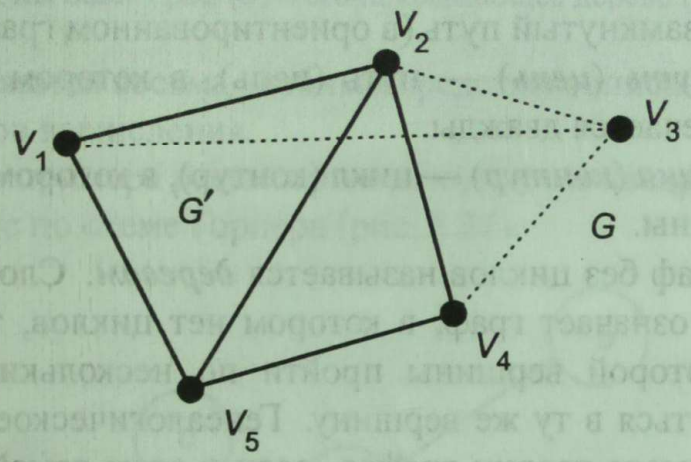


Рис. 8.17. Подсеть G' сети G

8.4. Циклы на графе

Маршрут на графе $G = (V, E)$ из вершины u_1 в вершину u_n — это конечная последовательность вершин $W = u_1, u_2, \ldots, u_n$ таких, что $(u_i, u_{i+2}) \in E(G)$ для каждого i, $1 \leq i \leq n - 1$. Маршрут W замкнут, если $u_1 = u_n$; в противном случае маршрут W открыт. Маршрут называется путем, если ни одна вершина (и, следовательно, ни одно из ребер) не появляется в W более одного раза (рис. 8.18).

На рис. 8.18 $uvxwywzvwvu$ есть маршрут; запись $uvxwz$ — путь, а $vxwzv$ — цикл.

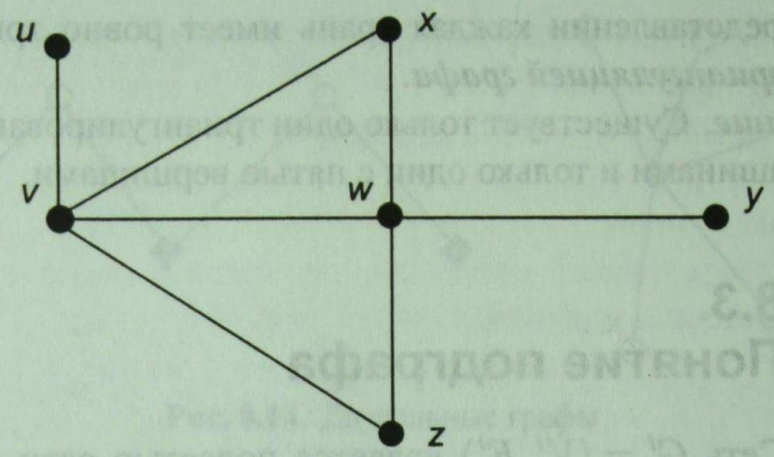


Рис. 8.18. Маршруты и циклы

Цепь — незамкнутый маршрут (путь), в котором все ребра (дуги) попарно различны.

Цикл — замкнутая цепь (в неориентированном графе).

Контур — замкнутый путь (в ориентированном графе).

Простой путь (***цепь***) — путь (цепь), в котором ни одна дуга (ребро) не встречается дважды.

Простой цикл (***контур***) — цикл (контур), в котором все вершины попарно различны.

Связный граф без циклов называется ***деревом***. Слово «дерево» в теории графов означает граф, в котором нет циклов, т.е. в котором нельзя из некоторой вершины пройти по нескольким различным ребрам и вернуться в ту же вершину. Генеалогическое дерево будет деревом и в смысле теории графов, если в этом семействе не было браков между родственниками. Деревья особенно часто возникают на практике при изображении различных иерархий. На рис. 8.19 показано библейское генеалогическое дерево.

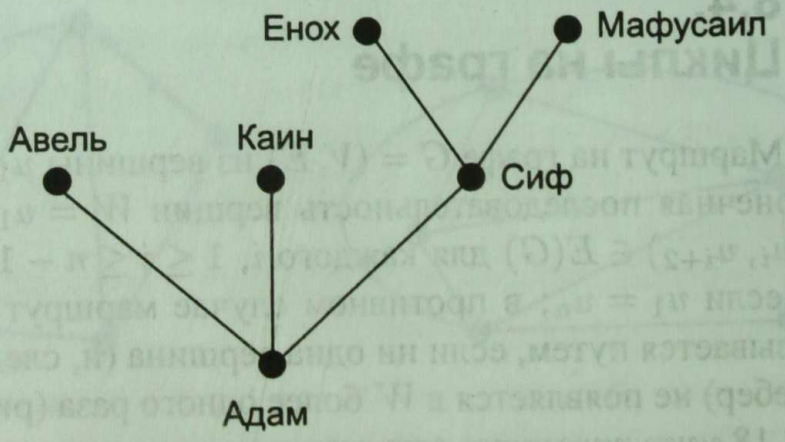


Рис. 8.19. Генеалогическое дерево

Граф является деревом тогда и только тогда, когда каждая пара различных вершин соединяется одной и только одной цепью. Если все вершины графа G принадлежат дереву T, то считается, что дерево покрывает граф G, т.е. имеем покрывающее дерево (рис. 8.20).

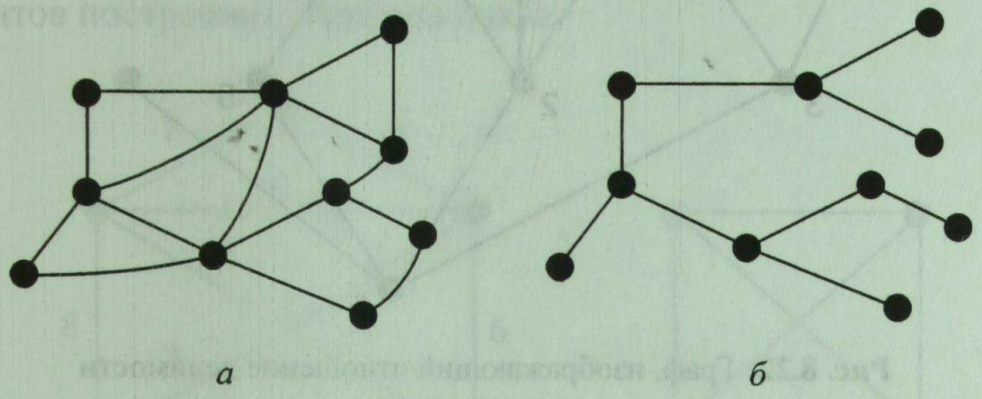


Рис. 8.20. Граф (*а*) и его покрывающее дерево (*б*)

Графы являются весьма удобным средством описания и оптимизации алгоритмов вычисления.

В качестве примера рассмотрим вычисление квадратного полинома $ax^2 + bx + c$ по схеме Горнера (рис. 8.21).

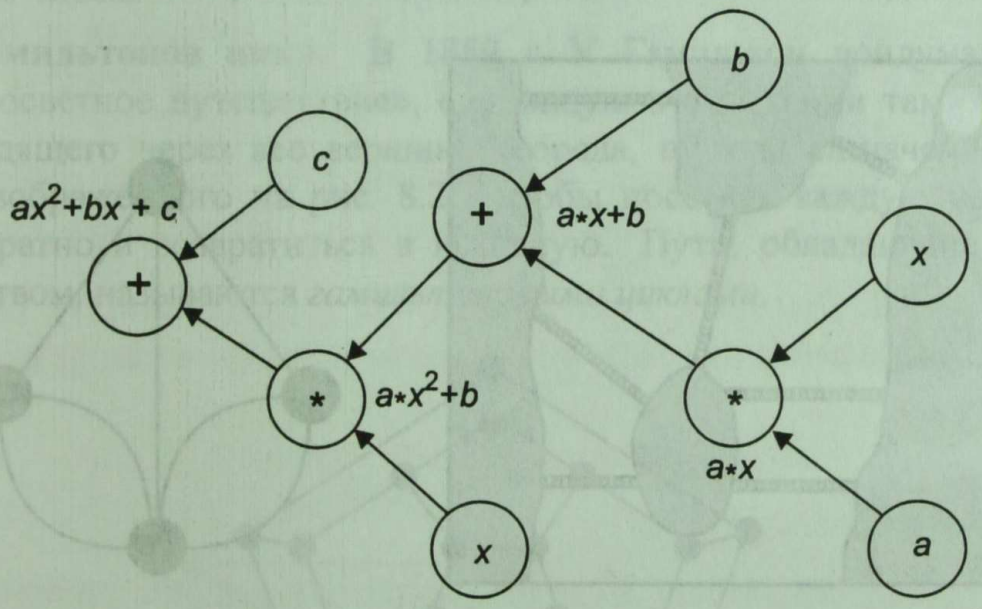


Рис. 8.21. Граф вычисления квадратного полинома по схеме Горнера

Построим граф, изображающий отношение делимости на множестве $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Принцип такой: если от одного числа до другого есть цепь, ведущая вверх, тогда второе число делится на первое (рис. 8.22).

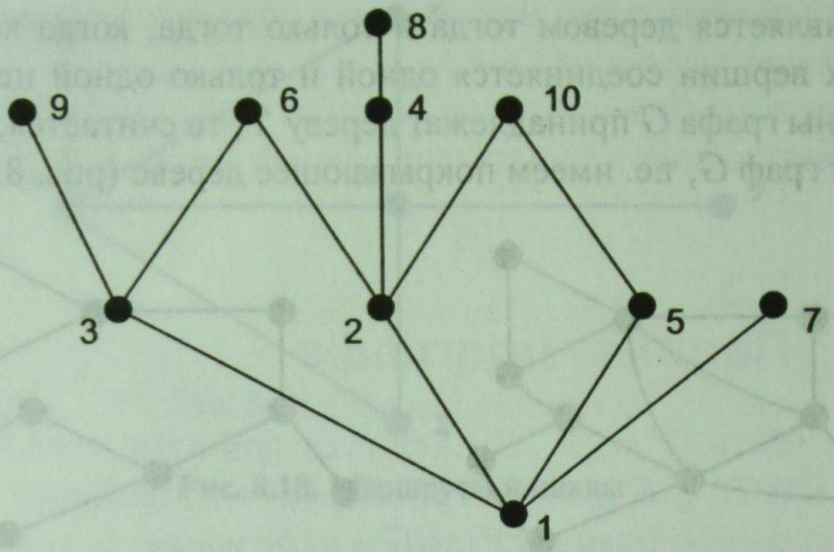
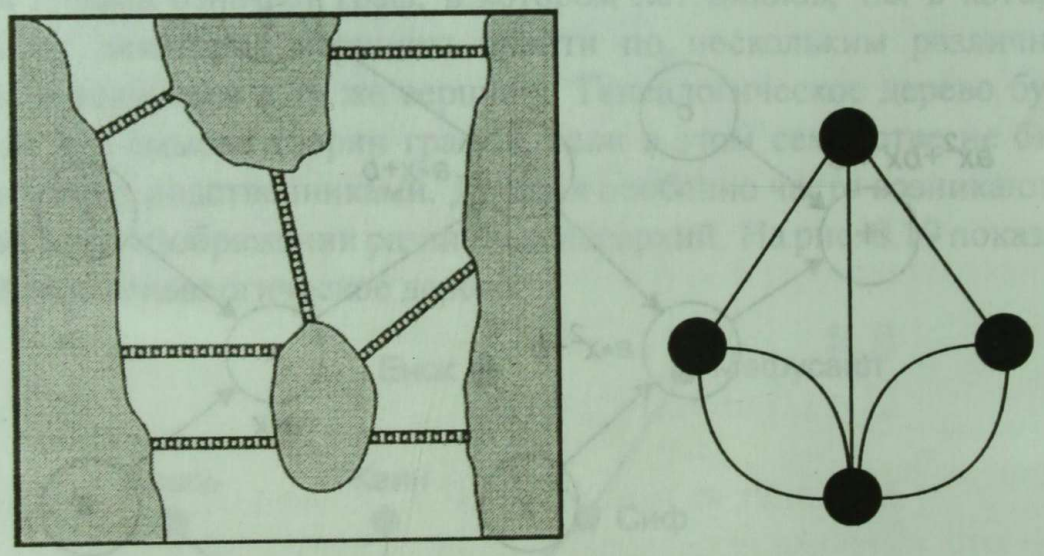


Рис. 8.22. Граф, изображающий отношение делимости на множестве

Эйлеров цикл. Цикл называется эйлеровым, если он проходит через все ребра и при этом только по одному разу. Граф, содержащий эйлеров цикл, называется эйлеровым графом.

Задача 1. Через город Калининград протекает река Прегель, омывающая остров. На реке имеется семь мостов (рис. 8.23). Может ли пешеход обойти все мосты, пройдя по каждому только один раз?



а *б*

Рис. 8.23. Задача о мостах:
а — рисунок мостов; *б* — графовая модель

Решение. Эйлеров цикл в графе существует тогда и только тогда, когда граф связный и все его вершины имеют четные степени. В графе,

имеющем более двух вершин с нечетной степенью, такого обхода не существует.

Задача 2. Начертить рисунок, не отрывая карандаш от бумаги и не проходя линию дважды (рис. 8.24). Цифрами показан один из вариантов построения эйлерова цикла.

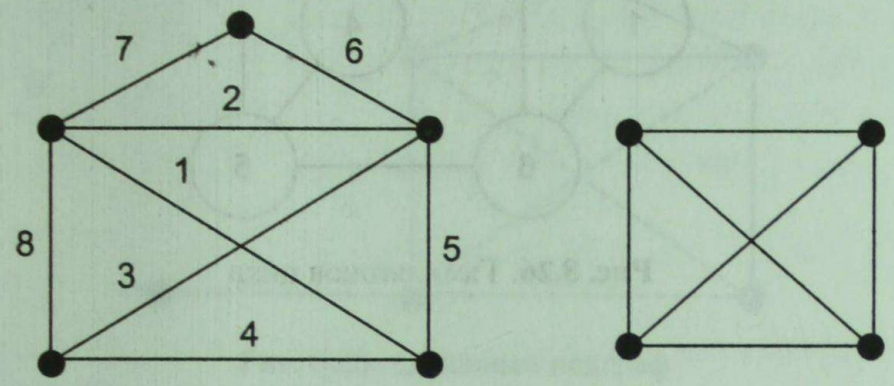


Рис. 8.24. Графы, содержащие циклы:
а — эйлеров цикл; *б* — неэйлеров цикл

Критерий наличия эйлерова цикла в графе: связный граф является эйлеровым тогда и только тогда, когда степени всех его вершин — четные числа.

Гамильтонов цикл. В 1859 г. У. Гамильтон придумал игру «Кругосветное путешествие», состоящую в отыскании такого пути, проходящего через все вершины (города, пункты назначения) графа, изображенного на рис. 8.25, чтобы посетить каждую вершину однократно и возвратиться в исходную. Пути, обладающие таким свойством, называются *гамильтоновыми циклами.*

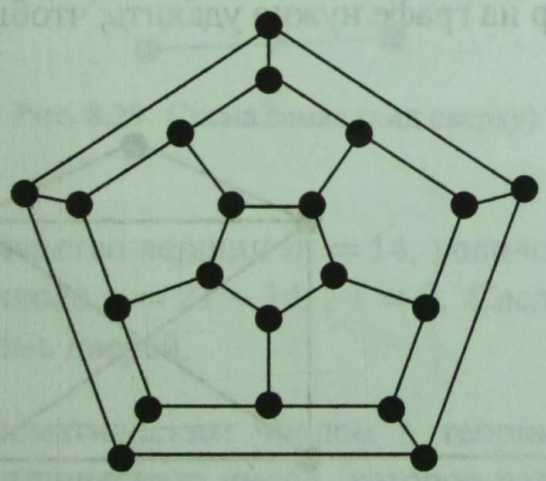


Рис. 8.25. Игра «Кругосветное путешествие»

Например, на графе (рис. 8.26) имеется гамильтонов цикл: 1 – 2 – 3 – 4 – 5 – 6 – 1.

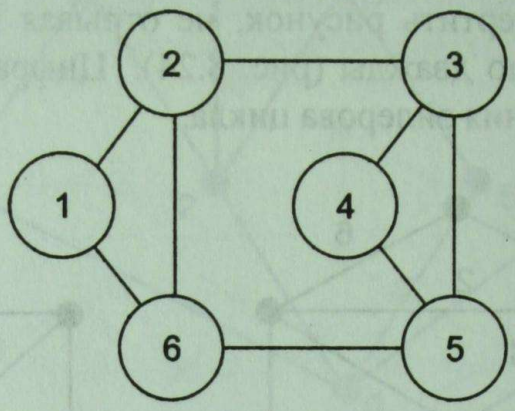


Рис. 8.26. Гамильтонов цикл

8.5. Цикломатическое число графа

Цикломатическим числом графа называется число, равное увеличенной на единицу разности между количеством ребер и количеством вершин графа: $\gamma = n - m + 1$, где n — количество ребер; m — количество вершин.

Цикломатическое число графа показывает, сколько ребер надо удалить из графа, чтобы в нем не осталось ни одного цикла.

Задача 1. Дан граф, у которого $m = 6$, $n = 9$ (рис. 8.27). Определить, сколько ребер на графе нужно удалить, чтобы в нем не осталось ни одного цикла.

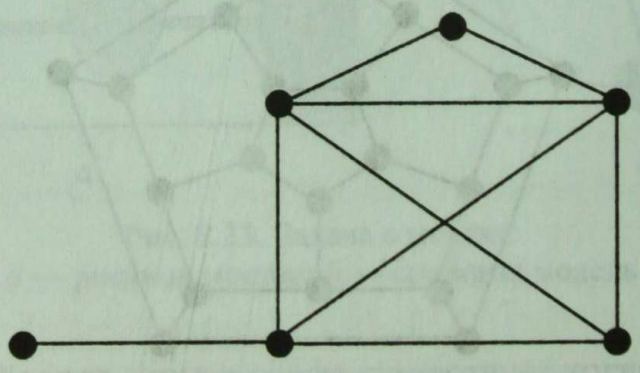


Рис. 8.27. Исходный граф

Для заданного графа цикломатическое число $\gamma = 9 - 6 + 1 = 4$. Это означает, что если на графе удалить 4 ребра, то в нем не останется ни одного цикла. Тогда остовный подграф будет иметь вид, показанный на рис. 8.28.

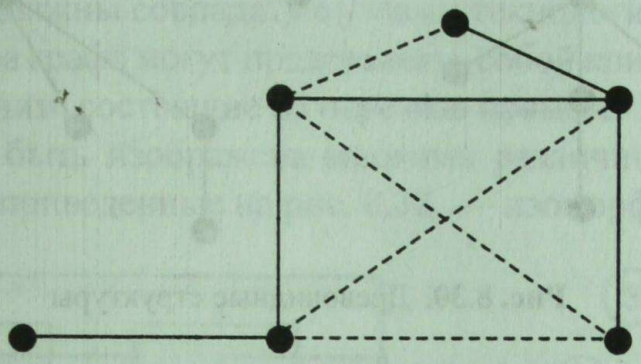


Рис. 8.28. Остовный подграф

Задача 2. Какое минимальное число дверей нужно сделать в замке (рис. 8.29), чтобы попасть во все комнаты (дверь — одно ребро).

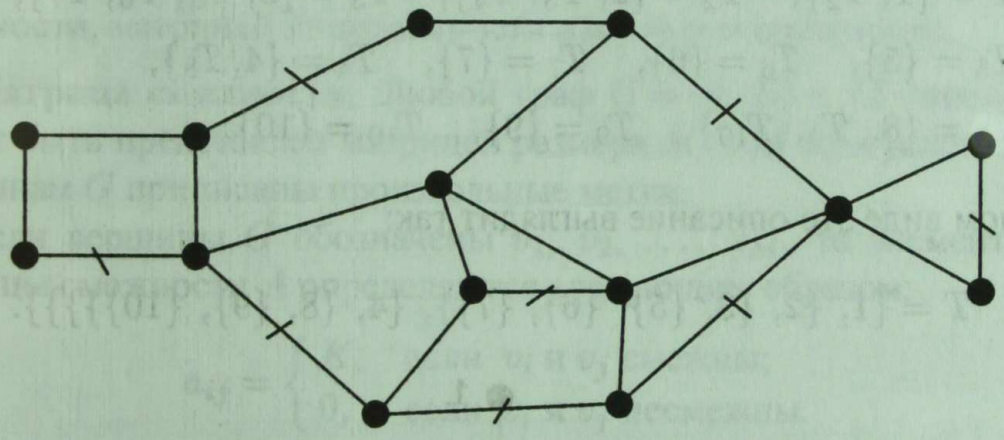


Рис. 8.29. Схема замка (вид сверху)

Решение. Количество вершин $m = 14$, количество ребер $n = 21$. Цикломатическое число $\gamma = 21 - 14 + 1 = 8$. Следовательно, в замке нужно сделать восемь дверей.

Наряду с цикломатическим числом в теории графов вводится понятие ***коцикломатического числа***, которое равно полному числу ребер в остовном дереве графа.

Дерево — это неориентированная связная сеть без циклов. На рис. 8.30 приведены различные деревья.

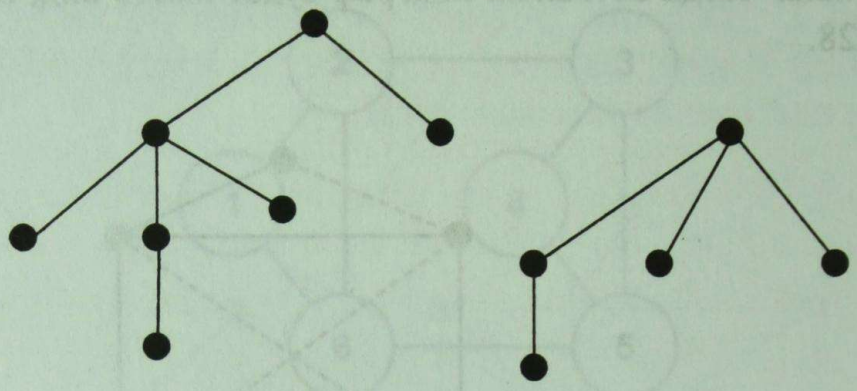


Рис. 8.30. Древовидные структуры

Ориентированная сеть является деревом тогда и только тогда, когда она дерево с дугами, рассматриваемыми как неориентированные ребра. Ациклическая ориентированная сеть не обладает ориентированными циклами. Например, дерево на рис. 8.31 можно описать следующим образом:

$$T = \{1,\ T_2\}, \quad T_2 = \{2,\ T_3,\ T_4\}, \quad T_3 = \{3,\ T_5,\ T_6,\ T_7\}, \tag{1}$$

$$T_5 = \{5\}, \quad T_6 = \{6\}, \quad T_7 = \{7\}, \quad T_4 = \{4, T_8\}, \tag{2}$$

$$T_8 = \{8,\ T_9,\ T_{10}\}, \quad T_9 = \{9\}, \quad T_{10} = \{10\}. \tag{3}$$

В явном виде это описание выглядит так:

$$T = \{1,\ \{2,\ \{3,\ \{5\},\ \{6\},\ \{7\}\},\ \{4,\ \{8,\ \{9\},\ \{10\}\}\}\}\}.$$

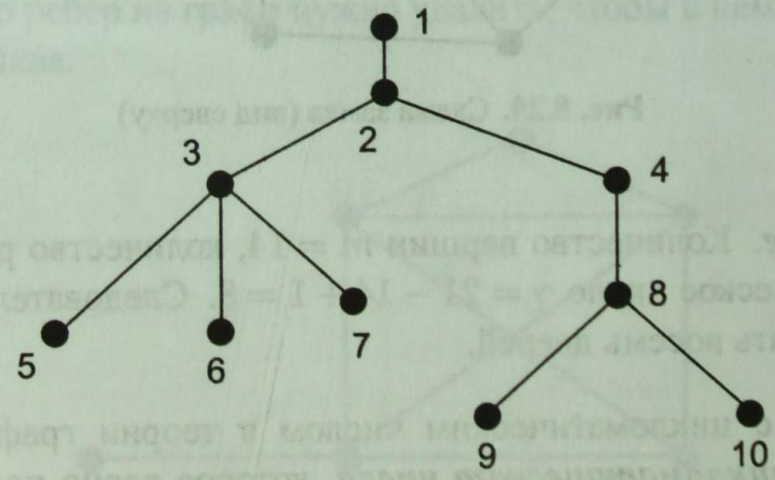


Рис. 8.31. Дерево

8.6. Представление графов в ПЭВМ

Очертанием графа считается любая топологически связанная область, ограниченная ребрами графа. Для многих приложений все узлы графа должны совпадать с узлами технологической сетки. В этом случае ребра графа могут представлять собой кривые линии, дуги или ломаные линии, состоящие из отрезков прямых. Важно помнить, что сеть может быть изображена многими различными способами; например, сети, приведенные на рис. 8.32, — изоморфные.

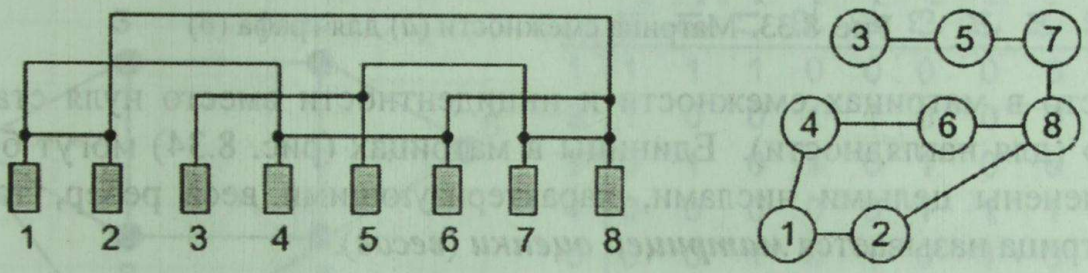


Рис. 8.32. Способы изображения сети

Графы в ПЭВМ можно представить тремя способами: матрицей смежности, матрицей инцидентности и вектором смежности.

Матрица смежности. Любой граф $G = (V, E)$ с M вершинами может быть представлен матрицей размера $M \times M$ при условии, что вершинам G приписаны произвольные метки.

Если вершины G обозначены $v_1, v_2, \ldots, v_M$, то элементы a_{ij} матрицы смежности A определяются следующим образом:

$$a_{ij} = \begin{cases} K, & \text{если } v_i \text{ и } v_j \text{ смежны;} \\ 0, & \text{если } v_i \text{ и } v_j \text{ несмежны.} \end{cases}$$

Здесь K — вес ребра, соединяющего данные вершины. На рис. 8.33 представлены граф и матрица смежности, описывающая его.

Матрица смежности неориентированного графа является симметричной.

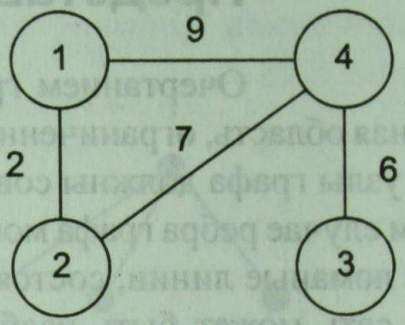
Матрица инцидентности. Если вершины графа обозначены v_1, $v_2, \ldots, v_M$, а ребра — метками $x_1, x_2, \ldots, x_N$, то матрица инцидентности определяется следующим образом:

$$b_{ij} = \begin{cases} 1, & \text{если ребро } x_j \text{ инцидентно вершине } v_i; \\ 0, & \text{если ребро } x_j \text{ не инцидентно вершине } v_i. \end{cases}$$

$A =$

	v_1	v_2	v_3	v_4
v_1	–	2	–	9
v_2	2	–	–	7
v_3	–	–	–	6
v_4	9	7	6	–

а



б

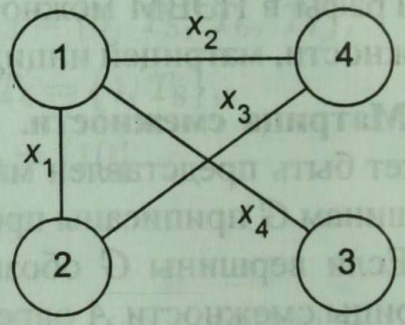
Рис. 8.33. Матрица смежности (*а*) для графа (*б*)

Часто в матрицах смежности и инцидентности вместо нуля ставят «–» (для наглядности). Единицы в матрицах (рис. 8.34) могут быть заменены целыми числами, характеризующими веса ребер, такая матрица называется ***матрицей оценки*** (***весов***).

$B =$

	x_1	x_2	x_3	x_4
v_1	1	1	–	1
v_2	1	–	1	–
v_3	–	–	–	1
v_4	–	1	1	–

а



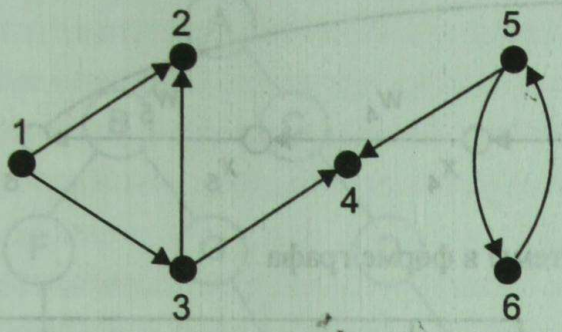
б

Рис. 8.34. Матрица инцидентности (инциденций) (*а*) для графа (*б*)

На рис. 8.35 приведены матрицы инциденций для ориентированного и неориентированного графов.

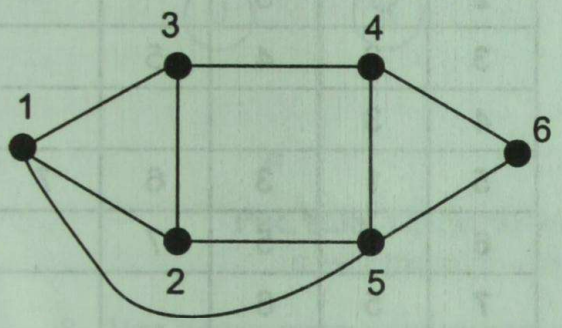
Модель системы часто представляется ориентированным графом $G = (V, E)$ с множеством вершин V и множеством дуг E — упорядоченных пар номеров смежных вершин $[i, j]$, $E = [i_1, j_1], \ldots, [i_n, j_n]$. Общее количество таких пар обозначим как Q.

Одним из способов представления топологии является матрица изоморфности D, в строках которой представлены номера входящих (с плюсом) и выходящих (с минусом) дуг.



	<1,2>	<1,3>	<3,2>	<3,4>	<5,4>	<5,6>	<6,5>
1	–1	–1	0	0	0	0	0
2	1	0	1	0	0	0	0
3	0	1	–1	–1	0	0	0
4	0	0	0	1	1	0	0
5	0	0	0	0	–1	–1	1
6	0	0	0	0	0	1	–1

а



	{1,2}	{1,3}	{1,5}	{2,3}	{2,5}	{3,4}	{4,5}	{4,6}	{5,6}
1	1	1	1	0	0	0	0	0	0
2	1	0	0	1	1	0	0	0	0
3	0	1	0	1	0	1	0	0	0
4	0	0	0	0	0	1	1	1	0
5	0	0	1	0	1	0	1	0	1
6	0	0	0	0	0	0	0	1	1

б

Рис. 8.35. Графы и их матрицы инциденций:
а) ориентированный граф; б) неориентированный граф

Для графа, приведенного на рис. 8.36, матрицы смежности и изоморфности имеют вид:

$$R = \begin{vmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{vmatrix}, \quad D = \begin{vmatrix} +6 & -1 \\ +1, +7 & -2 \\ +2 & -3, -7 \\ +3 & -4 \\ +4 & -5 \\ +5 & -6 \end{vmatrix}.$$

Избыточность хранимой информации в матрице смежности (нулевые значения) компенсируются простотой вычислительных алгоритмов и скоростью получения требуемой информации из матрицы.

Другой способ представления графа — ***вектор смежности***, который выдает списки узлов, с которыми данный узел (вершина) связан непосредственно. Для графа, изображенного на рис. 8.37, такое описание можно представить в виде структуры (таблицы). В колонке S

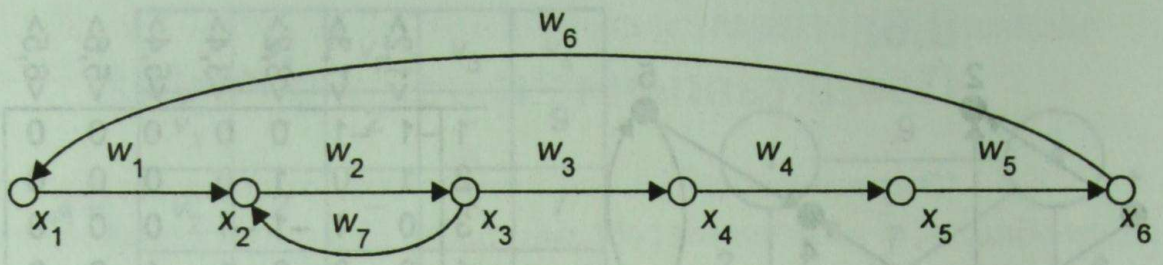
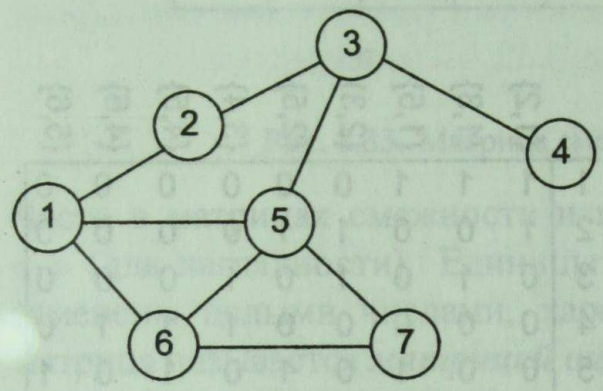


Рис. 8.36. Модель системы в форме графа



а

S	Список смежных узлов			
1	2	5	6	
2	1	3		
3	2	4	5	
4	3			
5	1	3	6	7
6	1	5	7	
7	5	6		

б

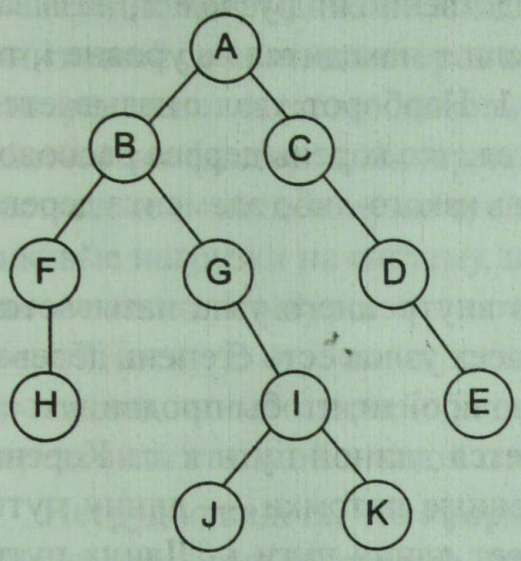
Рис. 8.37. Граф (*а*), представленный вектором смежности (*б*)

представлены номера узлов, далее в строке таблицы следует список соседних узлов. По этой причине число колонок в каждой из строк различно.

Представление древовидных структур. При представлении древовидных структур в машинной памяти в виде связного списка каждый элемент списка может содержать три поля, в одном из которых хранятся данные, соответствующие элементу (базовый тип), а в двух оставшихся полях — указатели левого и правого потомков узла (рис. 8.38).

Контрольные вопросы

1. Дайте определение графа.
2. Что такое степень вершины графа?
3. Чем характеризуется изоморфизм?
4. Какой граф называется регулярным?
5. Что такое инвариант сети?
6. Какой граф называется двудольным?
7. Дайте определение подграфа.



а

Номер строки	Элемент	Левый потомок	Правый потомок
1	*A*	2	3
2	*B*	6	7
3	*C*	0	4
4	*D*	0	5
5	*E*	0	0
6	*F*	8	0
7	*G*	0	9
8	*H*	0	0
9	*I*	10	11
10	*J*	0	0
11	*K*	0	0

б

Рис. 8.38. Представление древовидной структуры:
а — дерево; *б* — табличное представление

8. Что показывает цикломатическое число графа?
9. Чем маршрут отличается от цикла?
10. В чем различие эйлерова и гамильтонова циклов?
11. Опишите способы представления графов в ПЭВМ.
12. Укажите способы представления древовидных структур.

Глава 9
Алгоритмы построения остовного (покрывающего) дерева сети

Древовидная структура характеризуется следующими свойствами.

1) существует единственный элемент, или узел, на который не ссылается никакой другой элемент и который называется корнем;

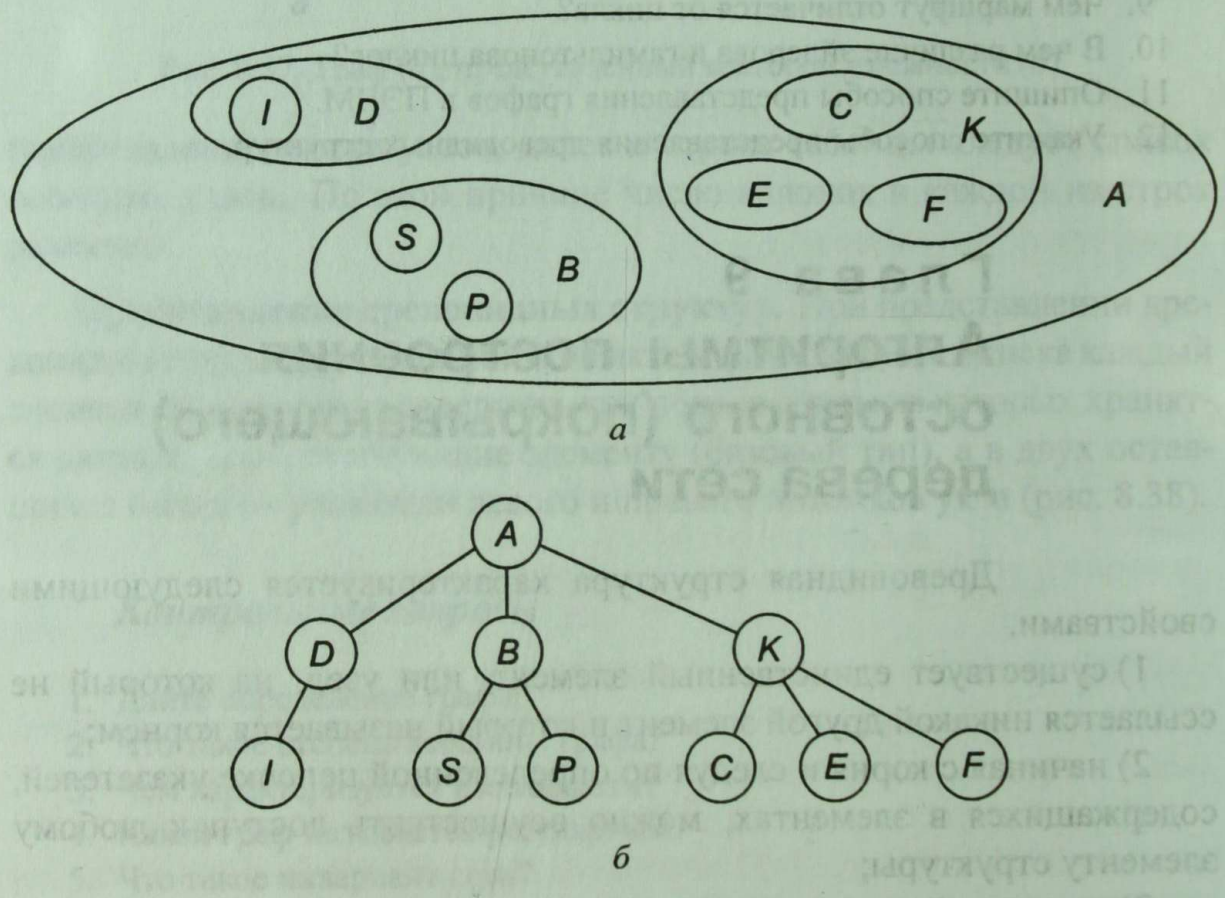
2) начиная с корня и следуя по определенной цепочке указателей, содержащихся в элементах, можно осуществить доступ к любому элементу структуры;

3) на каждый элемент, кроме корня, имеется единственная ссылка, т.е. каждый элемент адресуется единственным указателем.

Узел y, который находится непосредственно под узлом x, называется непосредственным *потомком* x; если x находится на уровне i, то говорят, что y расположен на уровне $i+1$. Наоборот, узел x называется непосредственным предком y. Считается, что корень дерева расположен на уровне 1. Максимальный уровень какого-либо элемента дерева называется его ***глубиной*** или ***высотой***.

Число непосредственных потомков внутреннего узла называется его ***степенью***. Максимальная степень всех узлов есть степень дерева. Число ветвей, или ребер, которые нужно пройти, чтобы продвинуться от корня к некоторому узлу x, называется длиной пути к x. Корень имеет длину пути 1, его непосредственные потомки — длину пути 2 и т.д. Вообще, узел на уровне i имеет длину пути i. Длина пути дерева определяется как сумма длин путей всех его узлов. Она также называется длиной ***внутреннего пути***.

Если элемент не имеет потомков, он называется ***терминальным элементом***, или ***листом***, а элемент, не являющийся терминальным, называется ***внутренним узлом*** (рис. 9.1).



а

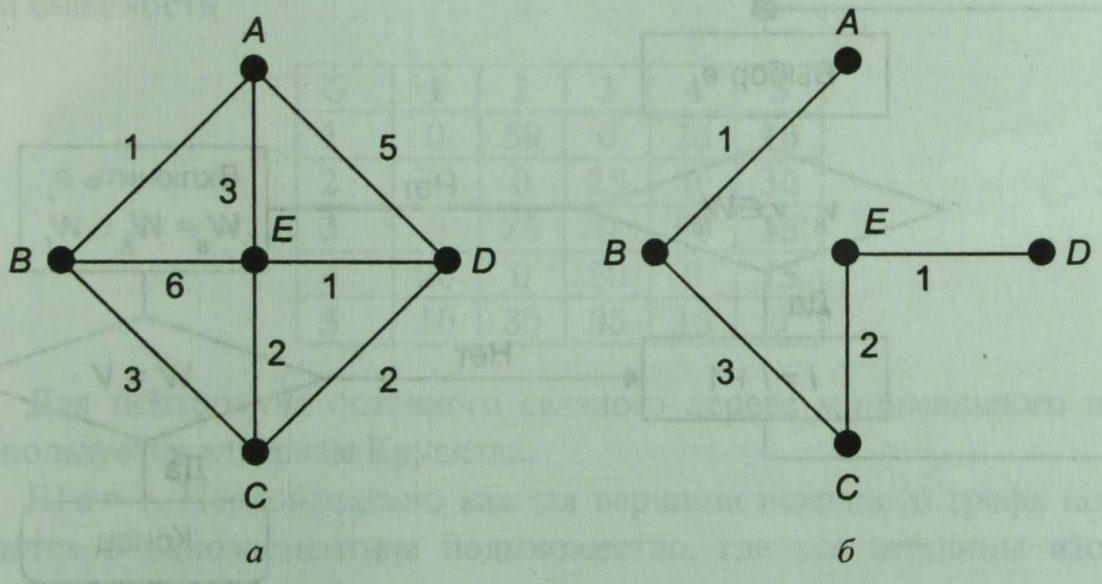
б

Рис. 9.1. Представления древовидной структуры:
а — вложенное множество; *б* — дерево

Предположим, что необходимо принять решение, связанное с организацией сети компьютеров в различных территориальных пунктах. Это решение является довольно сложным и зависит от большого количества факторов, которые включают вычислительные ресурсы, доступные в каждом пункте, соответствующие уровни потребностей, пиковые нагрузки на систему, возможное неэффективное использование основного ресурса в системе и, кроме того, стоимость предлагаемой сети. В эту стоимость входят: приобретение оборудования; прокладка линий связи; обслуживание системы и т.д. Необходимо определить стоимость такой сети.

Нетрудно видеть, что сформулированная здесь задача имеет много других аналогов. Например, требуется соединить несколько населенных пунктов линиями телефонной связи таким образом, чтобы все эти пункты были связаны в сеть и чтобы стоимость прокладки коммуникаций была минимальной. Вместо телефонных линий можно говорить о прокладке водопроводных коммуникаций, о строительстве дорог и т.д. Решение подобных задач возможно с использованием теории графов (сетей).

Пусть $G = (V, E)$ — связный неориентированный граф, содержащий циклы, т.е. замкнутые маршруты, где V — множество вершин, а E — множество ребер. Остовным (покрывающим) деревом называется подграф, не содержащий циклов, включающий все вершины исходного графа, для которого сумма весов ребер минимальна (рис. 9.2).



а *б*

Рис. 9.2. Граф (*а*) и его остовное дерево (*б*)

Цикломатическое число γ показывает, сколько ребер на графе нужно удалить, чтобы в нем не осталось ни одного цикла: $\gamma = n - m + 1$, где n — количество ребер; m — количество вершин. Например, для графа, изображенного на рис. 9.2, цикломатическое число равно $\gamma = n - m + 1 = 8 - 5 + 1 = 4$. Это значит, что если на графе убрать четыре ребра, то в нем не останется ни одного цикла, а суммарный вес ребер будет равен 7.

Для построения остовного дерева графа используются алгоритмы Крускала и Прима.

9.1. Метод Крускала

На рис. 9.3 показана схема алгоритма Крускала.

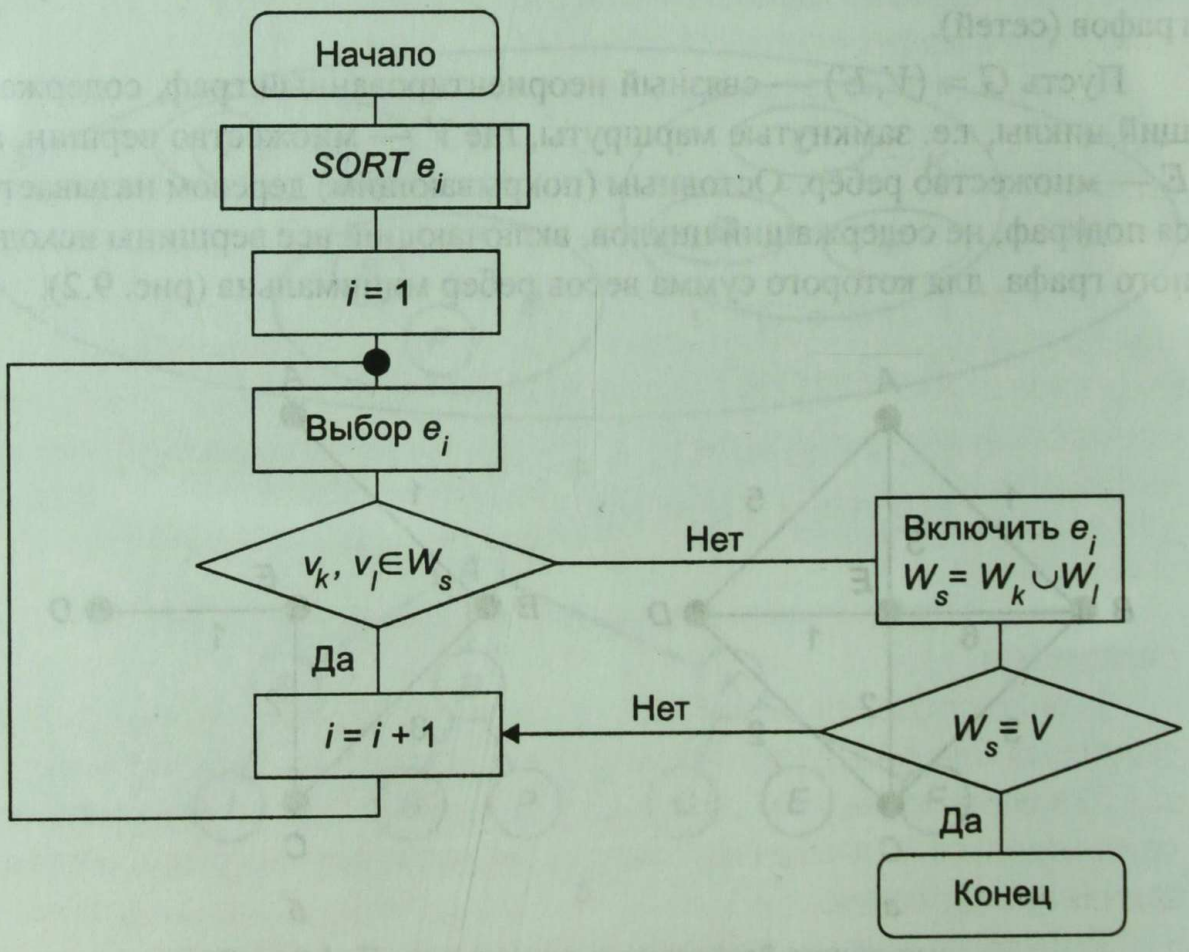


Рис. 9.3. Схема алгоритма Крускала

Блок $SORT\ e_i$ предполагает предварительную сортировку весов ребер в порядке их возрастания. В начале работы алгоритма предполагается, что в искомом остове не проведено ни одно ребро (т.е. остов состоит из изолированного множества вершин $v_1, v_2, \ldots, v_m$, где m — количество вершин графа). Поэтому считается, что множество W_s имеет вид: $W_s = \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}$, где $\{v_i\}$ обозначает множество, состоящее из единственной изолированной вершины v_i. Проверка $v_k, v_l \in W_s$ предполагает установление факта: входят ли вершины v_k, v_l во множество W_s как изолированные или они сами входят в подмножества постепенно увеличивающихся связных вершин W_k, W_l, каждое из которых имеет вид: $W_k = \{\ldots, v_k, \ldots\}$ и $W_l = \{\ldots, v_l, \ldots\}$.

Если обе вершины v_k, v_l содержатся в одном из подмножеств W_k, W_l, то ребро (k, l) в остов не включается, в противном случае данное ребро включается в остов, а множества W_k, W_l объединяются. Работа алгоритма Крускала заканчивается, когда множество W_s совпадет по мощности с множеством V — множеством всех вершин графа. Нетрудно видеть, что это произойдет, когда все вершины графа окажутся связными.

На рис. 9.4 показана работа алгоритма Крускала. Ребра рассматриваются в порядке возрастания весов (текущее ребро показано стрелкой). Ребро включается в остовное дерево, если оно не создает цикла и связывает вершины, принадлежащие разным множествам.

Пример 1. Построить остовное дерево графа, заданного матрицей смежности

G	**1**	**2**	**3**	**4**	**5**
1	0	**50**	**0**	**25**	**10**
2	50	0	**25**	**0**	**30**
3	0	25	0	**50**	**35**
4	25	0	50	0	**15**
5	10	30	35	15	0

Для построения остовного связного дерева минимального веса используется алгоритм Крускала.

Шаг 1. Первоначально каждая вершина исходного графа помещается в одноэлементное подмножество, где все вершины изолированы.

Шаг 2. Ребра сортируются по возрастанию веса.

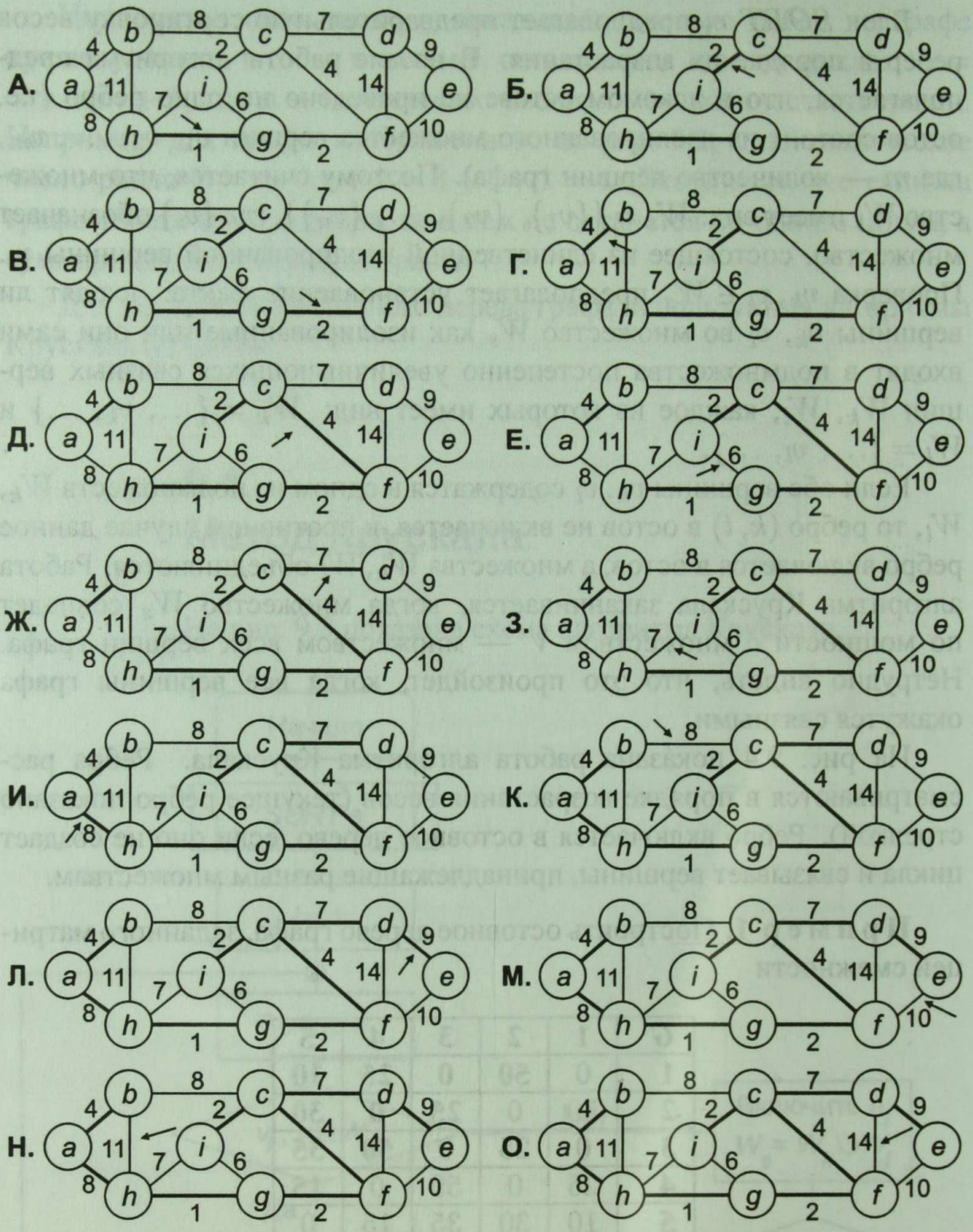


Рис. 9.4. Этапы построения остовного дерева алгоритмом Крускала

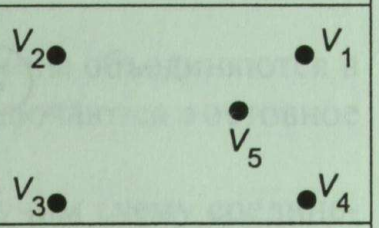
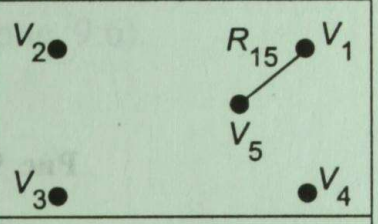
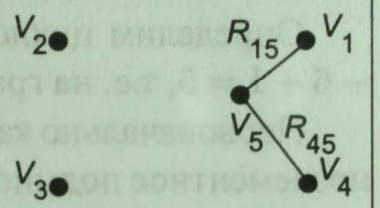
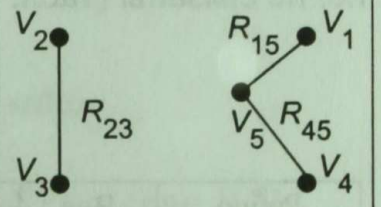
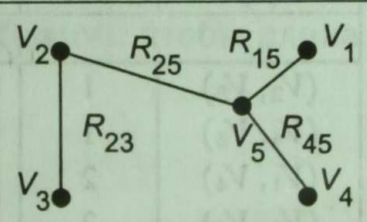
Шаг 3. Ребро включается в остовное дерево, если оно связывает вершины, принадлежащие разным множествам.

Шаг 4. Алгоритм заканчивает работу, когда все вершины объединяются в одно множество, при этом оставшиеся ребра не включаются в остовное дерево.

Рассмотрим реализацию этого алгоритма на примере построения остовного дерева минимального веса для графа G (табл. 9.1).

Таблица 9.1

Этапы построения остовного дерева методом Крускала

№ п/п	Выполняемые действия	Множество вершин	Граф
1	Построим остовной подграф, содержащий только изолированные вершины	Каждая вершина исходного графа помещается в одноэлементные подмножества: $\{V_1\}, \{V_2\}, \{V_3\}, \{V_4\}, \{V_5\}$	
2	Найдем ребро минимального веса (в данном случае R_{15}) и добавим его в остовной подграф	Образуется связное подмножество вершин: $\{V_1, V_5\}$. Одноэлементные подмножества вершин: $\{V_2\}$, $\{V_3\}$, $\{V_4\}$	
3	Среди оставшихся ребер найдем ребро минимального веса (в данном случае R_{45}) и добавим его в остовной подграф	Образуется связное подмножество вершин: $\{V_1, V_5, V_4\}$. Одноэлементные подмножества вершин: $\{V_2\}$, $\{V_3\}$	
4	Среди оставшихся ребер найдем ребро минимального веса (в данном случае R_{23}) и добавим его в остовной подграф	Создается второе связное подмножество: $\{V_2, V_3\}$. Существует также первое связное подмножество: $\{V_1, V_5, V_4\}$	
5	Среди оставшихся ребер найдем ребро минимального веса (в данном случае R_{25}) и добавим его в остовной подграф	Так как одна вершина ребра входит в одно связное подмножество, а другая вершина в другое связное подмножество, эти подмножества объединяются в единое связное множество: $\{V_1, V_5, V_4, V_2, V_3\}$	
6	Получен граф, который является: • *остовным*, так как включает все вершины; • *связным*, так как все вершины в нем можно соединить маршрутами; • *деревом*, так как в нем отсутствуют циклы; • *графом минимального веса*, так как в него последовательно включались ребра, отсортированные по возрастанию.		
7	Полученное остовное связное дерево обладает минимальным весом: $R_{15} + R_{45} + R_{23} + R_{25} = 10 + 15 + 25 + 30 = 80$.		

Пример 2. Пусть дана схема микрорайона. Необходимо соединить дома телефонным кабелем таким образом, чтобы его длина была минимальной. Схему микрорайона представим взвешенным графом (рис. 9.5).

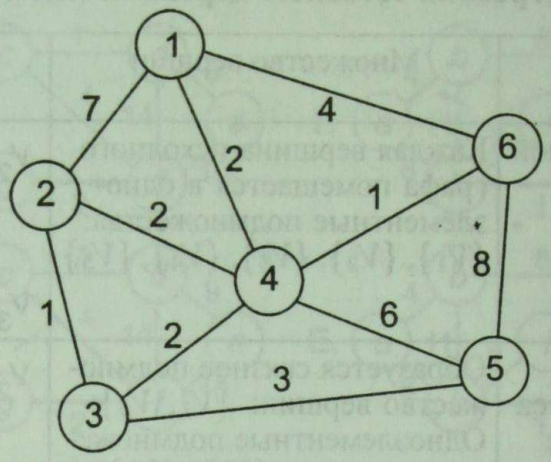


Рис. 9.5. Графическая схема микрорайона

Определим цикломатическое число графа $\gamma = n - m + 1 = 10 - 6 + 1 = 5$, т.е. на графе необходимо удалить пять ребер.

Первоначально каждая вершина исходного графа помещается в одноэлементное подмножество, считаем, что все вершины изолированы, т.е. не связаны (табл. 9.2).

Таблица 9.2

Реализация метода Крускала

Ребро	Вес	Множества вершин	Операция
		$\{V_1\}, \{V_2\}, \{V_3\}, \{V_4\}, \{V_5\}, \{V_6\}$	
(V_2, V_3)	1	$\{V_2, V_3\}, \{V_1\}, \{V_4\}, \{V_5\}, \{V_6\}$	Включение
(V_4, V_6)	1	$\{V_2, V_3\}, \{V_4, V_6\}, \{V_1\}, \{V_5\}$	Включение
(V_1, V_4)	2	$\{V_2, V_3\}, \{V_1, V_4, V_6\}, \{V_5\}$	Включение
(V_3, V_4)	2	$\{V_1, V_2, V_3, V_4, V_6\}, \{V_5\}$	Включение
(V_2, V_4)	2		Исключение
(V_3, V_5)	3	$\{V_1, V_2, V_3, V_4, V_5, V_6\}$	Включение

Ребро включается в остовное дерево, если оно связывает вершины, принадлежащие разным подмножествам, при этом вершины объединяются в новое подмножество.

В таблицу последовательно включаются ребра в порядке возрастания их весов. Ребро (V_2, V_3) связывает две вершины, принадлежащие

разным подмножествам $\{V_2\}$ и $\{V_3\}$. Поэтому ребро включается в остовное дерево, а вершины объединяются в одно подмножество $\{V_2, V_3\}$.

Ребро (V_4, V_6) также связывает вершины из разных подмножеств, оно включается в остовное дерево, а вершины образуют подмножество $\{V_4, V_6\}$.

Вершины V_2 и V_4 находятся в одном подмножестве, поэтому ребро (V_2, V_4) исключается из рассмотрения.

Алгоритм заканчивает работу, когда все вершины объединяются в одно множество, при этом оставшиеся ребра не включаются в остовное дерево.

Последовательно просматривая таблицу, получим схему соединения телефонным кабелем домов в микрорайоне (рис. 9.6).

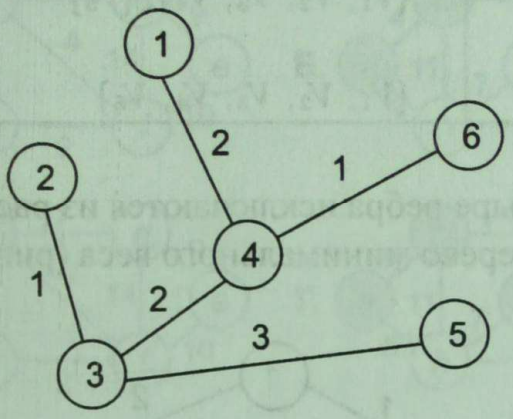


Рис. 9.6. Схема прокладки телефонного кабеля

Пример 3. Пусть дана схема компьютерной сети (рис. 9.7).

Необходимо соединить компьютеры таким образом, чтобы длина проводки была минимальной.

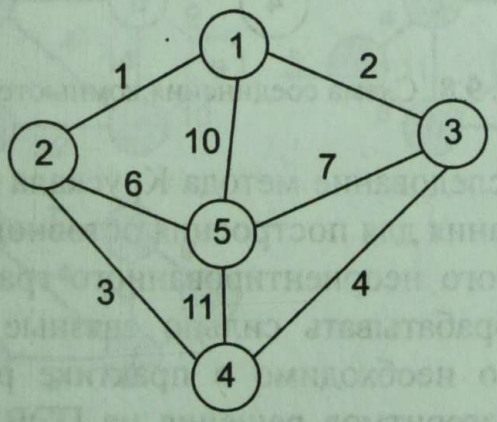


Рис. 9.7. Схема компьютерной сети

Определим цикломатическое число графа:

$$\gamma = n - m + 1 = 8 - 5 + 1 = 4.$$

Это значит, что на графе требуется удалить четыре ребра.

Решение показано в табл. 9.3.

Таблица 9.3

Реализация метода Крускала

Ребро	Вес	Множества вершин	Операция
		$\{V_1\}, \{V_2\}, \{V_3\}, \{V_4\}, \{V_5\}$	
(V_1, V_2)	1	$\{V_1, V_2\}, \{V_3\}, \{V_4\}, \{V_5\}$	Включение
(V_1, V_3)	2	$\{V_1, V_2, V_3\}, \{V_4\}, \{V_5\}$	Включение
(V_2, V_4)	3	$\{V_1, V_2, V_3, V_4\}, \{V_5\}$	Включение
(V_3, V_4)	4		Исключение
(V_2, V_5)	6	$\{V_1, V_2, V_3, V_4, V_5\}$	Включение

Оставшиеся четыре ребра исключаются из рассмотрения. В итоге получим остовное дерево минимального веса (рис. 9.8).

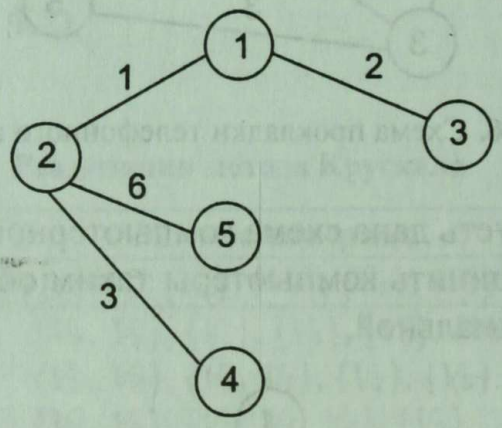


Рис. 9.8. Схема соединения компьютеров

Проведенное исследование метода Крускала показало эффективность его использования для построения остовного дерева минимального веса взвешенного неориентированного графа (сети). Данный метод позволяет обрабатывать сильно связные графы с большим числом вершин, что необходимо в практике разработки и создания эффективных алгоритмов решения на ПЭВМ широкого класса задач.

9.2. Метод Прима

В методе Прима от исходного графа переходим к его представлению в виде матрицы смежности. На графе выбирается произвольная вершина. Выбранная вершина образует первоначальный фрагмент остовного дерева. Затем анализируются веса ребер от выбранной вершины до оставшихся невыбранных вершин. Выбирается минимальное ребро, которое указывает на следующую выбранную вершину, и т.д. Процесс продолжается до тех пор, пока в остовное дерево не будут включены все вершины исходного графа.

Этапы работы алгоритма Прима показаны на рис. 9.9.

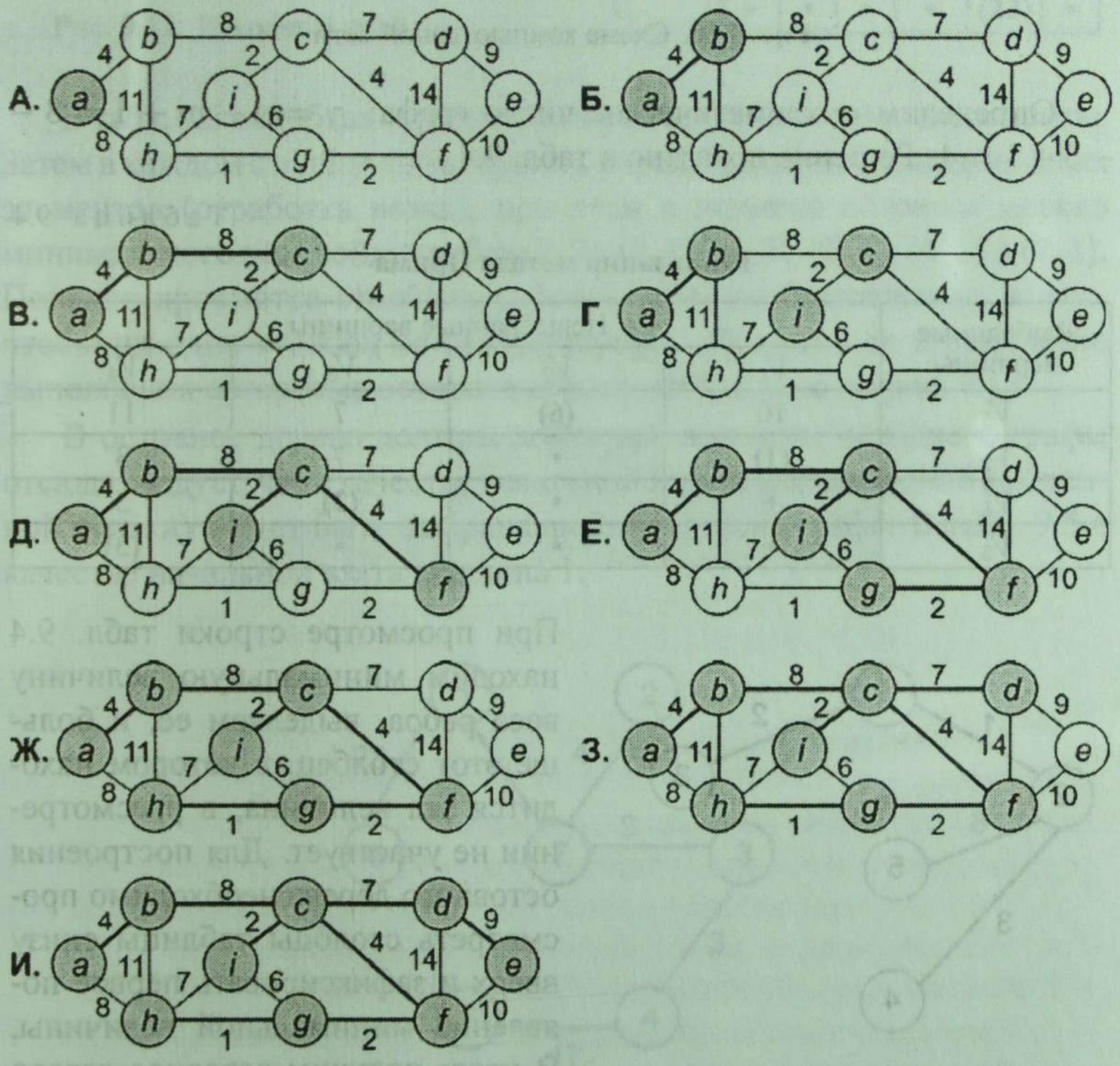


Рис. 9.9. Этапы построения остовного дерева алгоритмом Прима

Пример 1. Пусть дана схема компьютерной сети (рис. 9.10). Необходимо соединить компьютеры таким образом, чтобы длина проводки была минимальной.

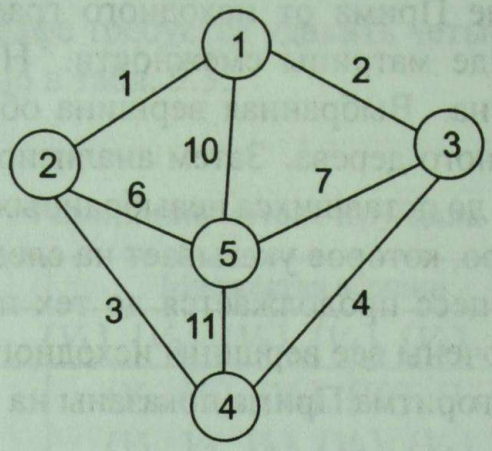


Рис. 9.10. Схема компьютерной сети

Определим цикломатическое число графа: $\gamma = n - m + 1 = 8 - 5 + 1 = 4$. Решение показано в табл. 9.4.

Таблица 9.4

Реализация метода Прима

Выбранные вершины	Невыбранные вершины			
	V_1	V_2	V_3	V_4
V_5	10	**(6)**	7	11
V_2	**(1)**	*	7	3
V_1	*	*	**(2)**	3
V_3	*	*	*	**(3)**

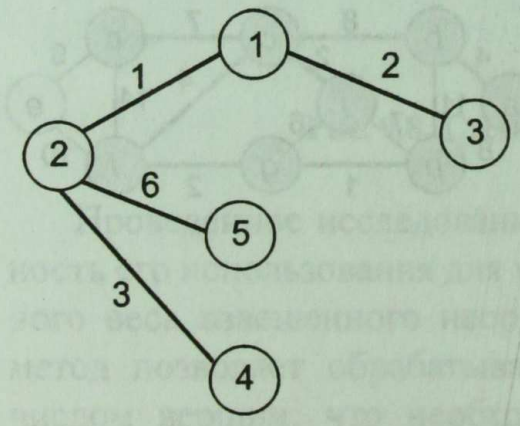


Рис. 9.11. Схема прокладки кабеля

При просмотре строки табл. 9.4 находим минимальную величину веса ребра, выделяем ее, и больше этот столбец, в котором находится эта величина, в рассмотрении не участвует. Для построения остовного дерева необходимо просмотреть столбцы таблицы снизу вверх и зафиксировать первое появление минимальной величины. В итоге получим остовное дерево минимального веса (рис. 9.11).

Пример 2. Построить остовное дерево минимального веса для графа, показанного на рис. 9.12.

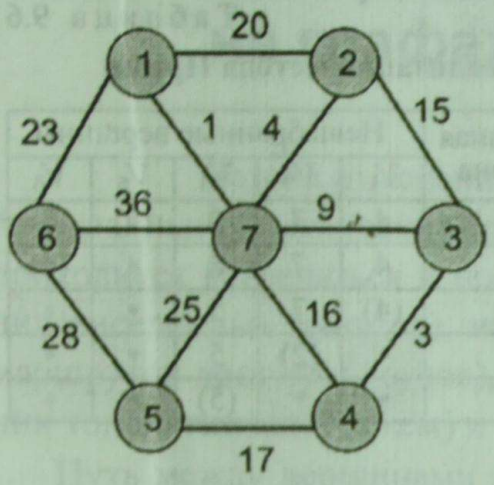


Рис. 9.12. Исходный граф

Таблица 9.5

Реализация метода Прима

Выбранные вершины	V_2	V_3	V_4	V_5	V_6	V_7
V_1	20	–	–	–	23	(1)
V_7	(4)	9	16	25	23	*
V_2	*	(9)	16	25	23	*
V_3	*	*	(3)	25	23	*
V_4	*	*	*	(17)	23	*
V_5	*	*	*	*	(23)	*

В табл. 9.5 в скобках отмечены выбранные минимальные элементы. Затем в каждом столбце фиксируются первые появления минимальных элементов (отработка назад), при этом в искомое остовное дерево минимального веса войдут ребра $(2,7)$, $(3,7)$, $(4,3)$, $(5,4)$, $(6,1)$ и $(7,1)$. Порядок просмотра столбцов рабочей таблицы безразличен; важно, чтобы ни один столбец не был пропущен. Полученное в результате выполнения алгоритма остовное дерево изображено на рис. 9.13.

В остовное дерево должны войти все вершины исходного графа, отсюда следует, что в качестве начальной вершины (т.е. первой выбранной строки) может быть выбрана любая вершина графа. В табл. 9.5 в качестве начальной взята вершина 1.

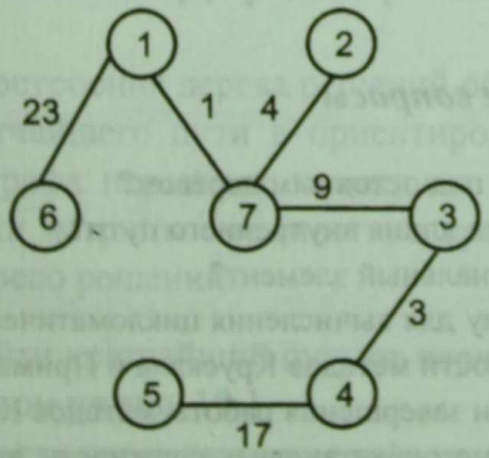


Рис. 9.13. Остовное дерево минимального веса

Пример 3. Построить остовное дерево графа (рис. 9.14) методом Прима.

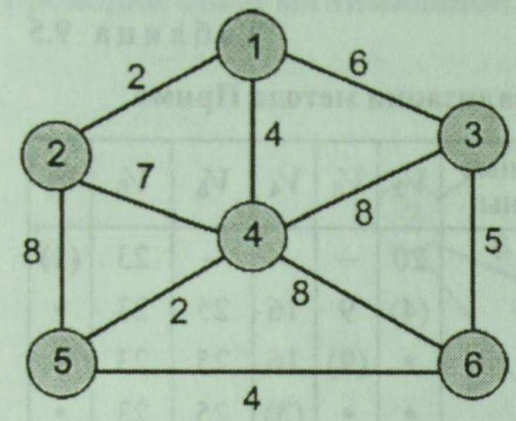


Рис. 9.14. Исходный граф

Таблица 9.6

Реализация метода Прима

Выбранная вершина	Невыбранные вершины				
	V_1	V_2	V_3	V_5	V_6
V_4	4	7	8	(2)	8
V_5	4	7	8	*	(4)
V_6	(4)	7	5	*	*
V_1	*	(2)	5	*	*
V_2	*	*	(5)	*	*

В табл. 9.6 показаны шаги выполнения метода Прима. Просмотр столбцов табл. 9.6 снизу вверх позволяет определить ребра, включенные в остовное дерево мининального веса (рис. 9.15).

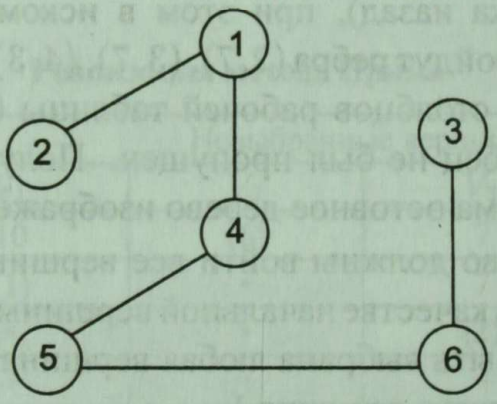


Рис. 9.15. Остовное дерево графа

Контрольные вопросы

1. Что понимается под остовным деревом?
2. Как определяется длина внутреннего пути?
3. Что такое терминальный элемент?
4. Укажите формулу для вычисления цикломатического числа.
5. Каковы особенности методов Крускала и Прима?
6. Каковы критерии завершения работы методов Крускала и Прима?
7. В чем состоит методика анализа сложности алгоритмов построения остовного дерева графа?

Глава 10
Алгоритмы нахождения на графах кратчайших путей

Задача отыскания на графе кратчайшего пути имеет многочисленные практические приложения. С решением подобной задачи приходится встречаться в технике связи (например, при телефонизации населенных пунктов), на транспорте (при выборе оптимальных маршрутов доставки грузов), в микроэлектронике (при проектировании топологии микросхем) и т.д.

Путь между вершинами i и j графа считается кратчайшим, если вершины i и j соединены минимальным числом ребер (случай не взвешенного графа) или если сумма весов ребер, соединяющих вершины i и j, минимальна (для взвешенного графа).

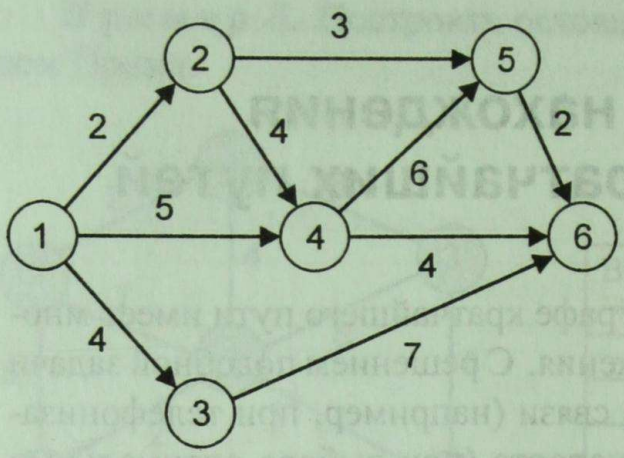
Важным показателем алгоритма является его эффективность. Применительно к поставленной задаче эффективность алгоритма может зависеть в основном от двух параметров графа: числа его вершин и числа весов его ребер. Рассмотрим три алгоритма для определения кратчайшего расстояния между вершинами графа: построение дерева решений, метод динамического программирования и метод Дейкстры.

10.1. Построение дерева решений

Метод построения дерева решений обычно используется для нахождения кратчайшего пути в ориентированном графе, при этом от исходного графа переходят к матрице смежности. Затем просматривают строки матрицы смежности, и граф последовательно «расслаивается» в дерево решений.

Пример 1. Найти кратчайший путь от вершины 1 до вершины 6 в графе, изображенном на рис. 10.1.

В матрице смежности весов данного графа символ «прочерк» обозначает отсутствие связности между соответствующими вершинами.



	V_1	V_2	V_3	V_4	V_5	V_6
V_1	–	2	4	5	–	–
V_2	–	–	–	4	3	–
V_3	–	–	–	–	–	7
V_4	–	–	–	–	6	4
V_5	–	–	–	–	–	2
V_6	–	–	–	–	–	–

Рис. 10.1. Граф и соответствующая ему матрица смежности

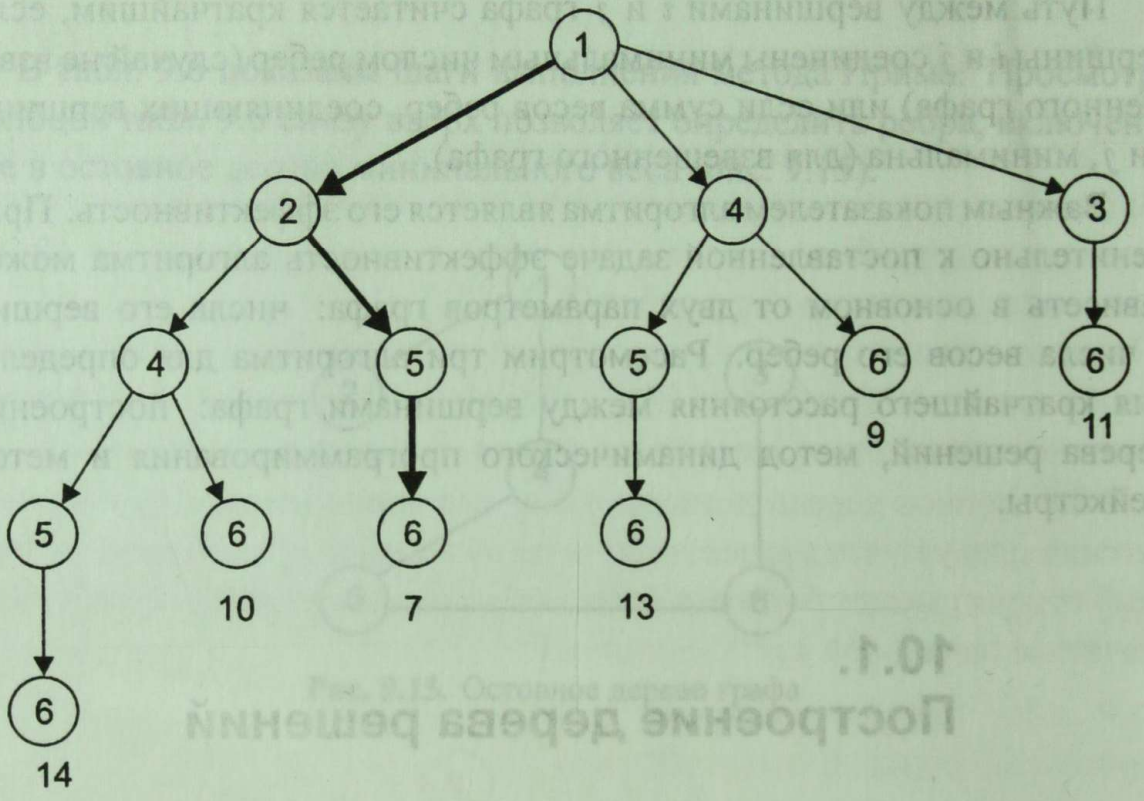
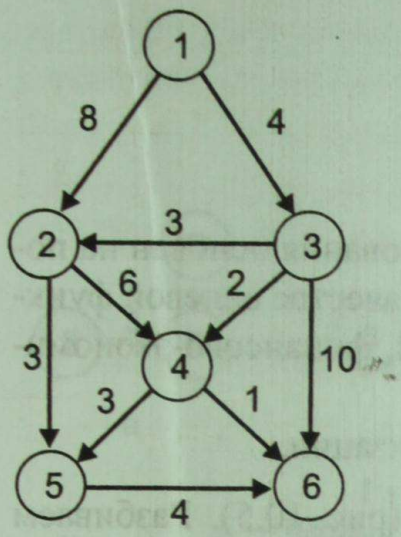


Рис. 10.2. Дерево перебора решений

Полное дерево перебора, построенное по данному алгоритму, изображено на рис. 10.2.

Из рис. 10.2 нетрудно видеть, что кратчайший путь из вершины 1 в вершину 6 равен 7 и имеет вид $(1-2-5-6)$.

Пример 2. Найти кратчайший путь от вершины 1 до вершины 6 в ориентированном графе, изображенном на рис. 10.3.



	V_1	V_2	V_3	V_4	V_5	V_6
V_1	–	8	4	–	–	–
V_2	–	–	–	6	3	–
V_3	–	3	–	2	–	10
V_4	–	–	–	–	3	1
V_5	–	–	–	–	–	4
V_6	–	–	–	–	–	–

Рис. 10.3. Граф и соответствующая ему матрица смежности

Последовательно просматриваем строки матрицы смежности и строим дерево решений. Полное дерево перебора изображено на рис. 10.4.

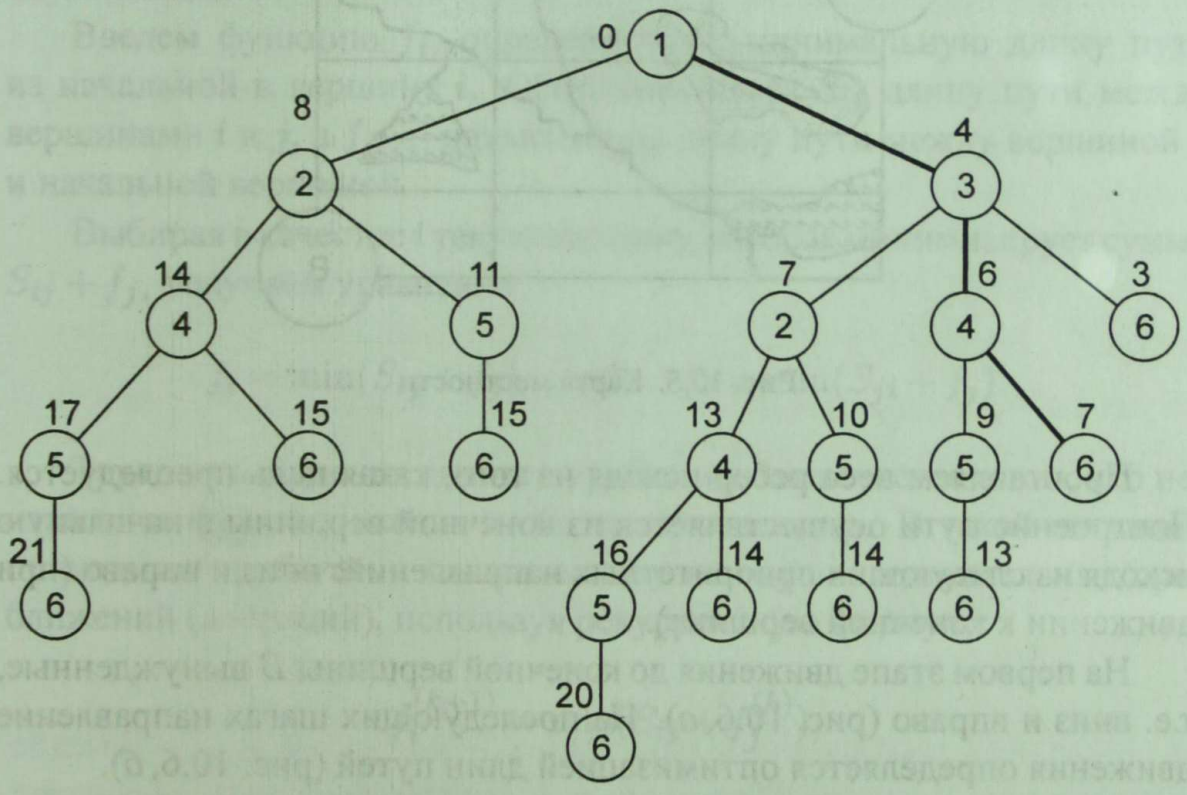


Рис. 10.4. Дерево перебора решений

Дерево решений показывает все возможные пути. Имеется девять вариантов путей от первой до шестой вершины. Кратчайший путь равен $V_1 - V_3 - V_4 - V_6$.

10.2. Метод динамического программирования

Метод динамического программирования основан на пошаговой оптимизации целевой функции, где в качестве целевой функции может быть стоимость, ресурсные затраты, финансово-экономические затраты, а также кратчайшие пути.

Рассмотрим пример топологической оптимизации.

Пример 1. Пусть дана карта местности (рис. 10.5). Разбиваем карту местности координатной сеткой. Шаг сетки выбирается исходя из условия точности. Узлы координатной сетки считаем вершинами графа.

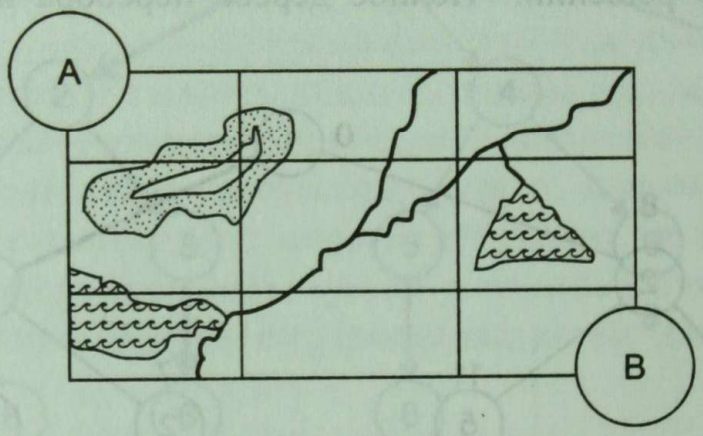


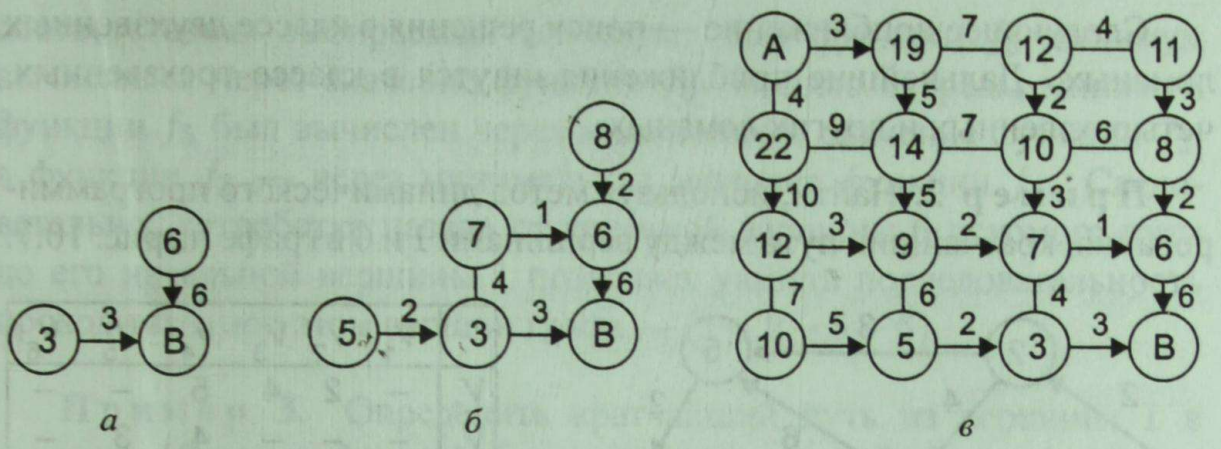
Рис. 10.5. Карта местности

Проставляем веса ребер исходя из того, какая цель преследуется. Построение пути осуществляется из конечной вершины в начальную исходя из следующих приоритетных направлений: вниз и вправо (при движении к конечной вершине).

На первом этапе движения до конечной вершины B вынужденные, т.е. вниз и вправо (рис. 10.6, *а*). На последующих шагах направление движения определяется оптимизацией длин путей (рис. 10.6, *б*).

В результате выполнения операций определяется длина пути между вершинами (22 единицы) и траектория движения (по стрелкам) — рис. 10.6, *в*.

Метод динамического программирования рассматривает многостадийные процессы принятия решения. При постановке задачи



а *б* *в*

Рис. 10.6. Выбор кратчайшего пути:
а — первый шаг; *б* — второй шаг; *в* — траектория пути

динамического программирования формируется некоторый критерий. Процесс разбивается на стадии (шаги), в которых принимаются решения, приводящие к достижению общей поставленной цели. Таким образом, метод динамического программирования — метод пошаговой оптимизации.

Введем функцию f_i, определяющую минимальную длину пути из начальной в вершину i. Обозначим через S_{ij} длину пути между вершинами i и j, а f_j — наименьшую длину пути между вершиной j и начальной вершиной.

Выбирая в качестве i такую вершину, которая минимизирует сумму $S_{ij} + f_j$, получаем уравнение

$$f_i = \min(S_{ij} + f_j) \quad \text{либо} \quad f_i = \min(S_{ji} + f_j).$$

Трудность решения данного уравнения заключается в том, что неизвестная функция входит в обе части равенства. В такой ситуации приходится прибегать к классическому методу последовательных приближений (итераций), используя рекуррентную формулу:

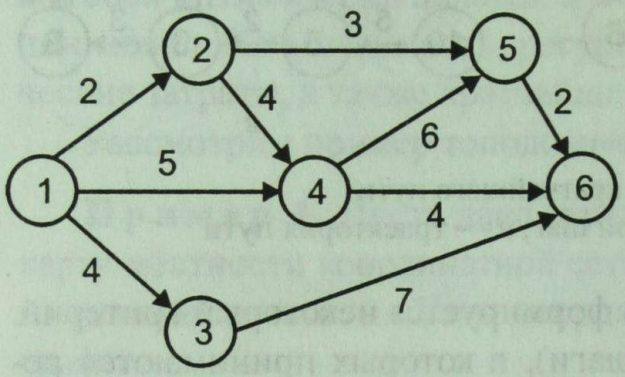
$$f_i^{(k+1)} = \min(S_{ij} + f_j^{(k)}),$$

где $f_j^{(k)}$ — k-е приближение функции.

Возможен другой подход к решению поставленной задачи с помощью метода стратегий. При движении из начальной точки i в конечную k получается приближение $f_i^{(0)} = S_{ik}$, где S_{ik} — длина пути между точками i и k.

Следующее приближение — поиск решения в классе двухзвенных ломаных. Дальнейшие приближения ищутся в классе трехзвенных, четырехзвенных и других ломаных.

Пример 2. Найти, используя метод динамического программирования, кратчайший путь между вершинами 1 и 6 в графе на рис. 10.7.



	V_1	V_2	V_3	V_4	V_5	V_6
V_1	–	2	4	5	–	–
V_2	–	–	–	4	3	–
V_3	–	–	–	–	–	7
V_4	–	–	–	–	6	4
V_5	–	–	–	–	–	2
V_6	–	–	–	–	–	–

Рис. 10.7. Граф и соответствующая ему матрица смежности

Первый этап алгоритма — поиск длины минимального пути. Для вершины 1 имеем $f_1 = 0$, так как для этой вершины никакой путь еще не пройден. Имеем $f_2 = \min(S_{12} + f_1) = 2 + 0 = 2$. Аналогично имеем $f_3 = \min(S_{13} + f_1) = 4 + 0 = 4$. Из рис. 10.7 видно, что из вершины 1 в вершину 4 возможны два пути: $(1-4)$ и $(1-2-4)$, поэтому функция f_4 будет иметь вид

$$f_4 = \min\{S_{14} + f_1,\ S_{24} + f_2\} = \min\{5+0,\ 4+2\} = \min\{5,\ 6\},$$

откуда выбирается минимальное значение функции $f_4 = 5$. Аналогичным способом определяется значение функций f_5 и f_6:

$$f_5 = \min\{S_{25} + f_2,\ S_{45} + f_4\} = \min\{3+2,\ 6+5\} = \min\{5,\ 11\};$$

$$f_6 = \min\{S_{56} + f_5,\ S_{46} + f_4,\ S_{36} + f_3\} = \\ = \min\{2+5,\ 4+5,\ 7+4\} = \min\{7,\ 9,\ 11\}.$$

Из последних соотношений находим, что имеют место следующие минимальные значения: $f_5 = 5$, $f_6 = 7$. Таким образом, в результате выполнения первого этапа алгоритма найдено, что кратчайший путь из вершины 1 в вершину 6 равен 7.

На втором этапе алгоритма будет найдена последовательность вершин, через которые проходит вычисленный минимальный путь. Для этого необходимо найти последовательность тех функций, которым

соответствовал выбираемый минимум. Для функции f_6 минимум вычислялся через значение функции f_5. В свою очередь, минимум функции f_5 был вычислен через минимальное значение функции f_2, а функции f_2 — через минимальное значение функции f_1. Следовательно, «отработка назад» от конечной вершины 6 искомого пути до его начальной вершины 1 позволяет указать последовательность проходимых при этом вершин графа — (1 – 2 – 5 – 6).

Пример 3. Определить кратчайший путь из вершины 1 в вершину 10 для графа, представленного на рис. 10.8.

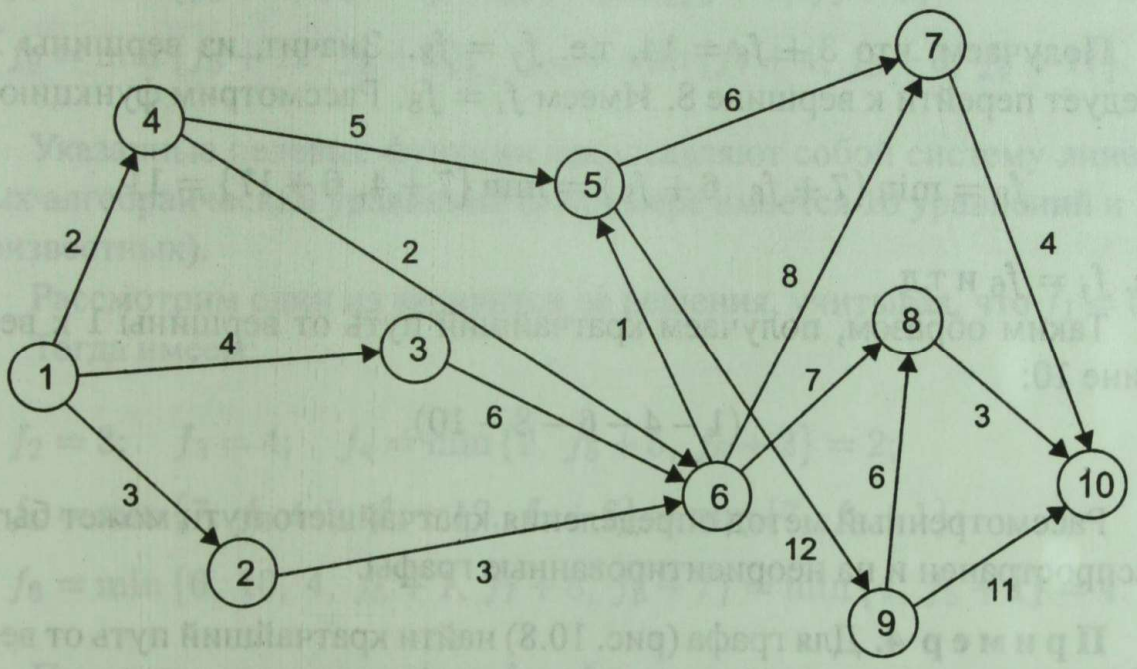


Рис. 10.8. Взвешенный ориентированный граф

Начальные условия: $f_1 = 0$.

Находим последовательно значения функции f_i (в условных единицах) для каждой вершины ориентированного графа:

$$f_2 = \min\{S_{21} + f_1\} = 3 + f_1 = 3 + 0 = 3;$$
$$f_3 = \min\{S_{31} + f_1\} = 4 + f_1 = 4 + 0 = 4;$$
$$f_4 = \min\{S_{41} + f_1\} = 2 + f_1 = 2 + 0 = 2;$$
$$f_5 = \min\{S_{54} + f_4,\ S_{56} + f_6\} = \min\{5 + 2,\ 1 + 4\} = 5;$$
$$f_6 = \min\{S_{64} + f_4,\ S_{63} + f_3,\ S_{62} + f_2\} = \min\{2 + 2,\ 6 + 4,\ 3 + 3\} = 4;$$
$$f_7 = \min\{S_{75} + f_5,\ S_{76} + f_6\} = \min\{6 + 5,\ 8 + 4\} = 11;$$
$$f_9 = \min\{S_{95} + f_5\} = 12 + 5 = 17;$$

$$f_8 = \min\{S_{86} + f_6,\ S_{89} + f_9\} = \min\{7+4,\ 6+17\} = 11;$$
$$f_{10} = \min\{S_{10,7} + f_7,\ S_{10,8} + f_8,\ S_{10,9} + f_9\} =$$
$$= \min\{4+11,\ 3+11,\ 11+17\} = 14.$$

Длина кратчайшего пути составляет 14 условных единиц. Для выбора оптимальной траектории движения следует осуществить просмотр функций f_i в обратном порядке, т.е. с f_{10}. Пусть $f_i = f_{10}$. В данном случае

$$f_{10} = \min\{4+f_7,\ 3+f_8,\ 11+f_9\} = \min\{4+11,\ 3+11,\ 11+17\} = 14.$$

Получаем, что $3 + f_8 = 14$, т.е. $f_j = f_8$. Значит, из вершины 10 следует перейти к вершине 8. Имеем $f_i = f_8$. Рассмотрим функцию

$$f_8 = \min\{7 + f_6,\ 6 + f_9\} = \min\{7+4,\ 6+17\} = 11,$$

т.е. $f_j = f_6$ и т.д.

Таким образом, получаем кратчайший путь от вершины 1 к вершине 10:

$$(1-4-6-8-10).$$

Рассмотренный метод определения кратчайшего пути может быть распространен и на неориентированные графы.

Пример 4. Для графа (рис. 10.8) найти кратчайший путь от вершины 1 до вершины 10, рассматривая граф как неориентированный. Матрица смежности весов графа в этом случае имеет вид:

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

Запишем выражения для функции f_i:

$$f_1 = 0;$$

$$f_2 = \min\{f_1 + 3,\ f_6 + 3\}; \quad f_3 = \min\{f_1 + 4,\ f_6 + 6\};$$

$$f_4 = \min\{f_1 + 2,\ f_5 + 5,\ f_6 + 2\};$$

$$f_5 = \min\{f_4 + 5,\ f_6 + 1,\ f_9 + 12,\ f_7 + 6\};$$

$$f_6 = \min\{f_2 + 3,\ f_3 + 6,\ f_4 + 2,\ f_5 + 1,\ f_7 + 8,\ f_8 + 7\};$$

$$f_7 = \min\{f_5 + 6,\ f_6 + 8\}; \quad f_8 = \min\{f_6 + 7,\ f_9 + 6\};$$

$$f_9 = \min\{f_5 + 12,\ f_8 + 6\}; \quad f_{10} = \min\{f_7 + 4,\ f_8 + 3,\ f_9 + 11\}.$$

Указанные целевые функции представляют собой систему линейных алгебраических уравнений (в примере имеется 10 уравнений и 10 неизвестных).

Рассмотрим один из вариантов ее решения, учитывая, что $f_1 = 0$. Тогда имеем:

$$f_2 = 3; \quad f_3 = 4; \quad f_4 = \min\{2,\ f_5 + 5,\ f_6 + 2\} = 2;$$

$$f_5 = \min\{7,\ f_6 + 1,\ f_9 + 12,\ f_7 + 6\} = \min\{7,\ f_6 + 1\};$$

$$f_6 = \min\{6,\ 10,\ 4,\ f_5 + 1,\ f_7 + 8,\ f_8 + 7\} = \min\{4,\ f_5 + 1\} = 4.$$

Подставив выражение для f_6 в f_5, получим

$$f_5 = \min\{7,\ 4 + 1\} = 5.$$

Тогда

$$f_7 = 11; \quad f_8 = \min\{11,\ f_9 + 6\};$$

$$f_9 = \min\{17,\ f_8 + 6\}; \quad f_{10} = \min\{15,\ f_8 + 3,\ f_9 + 11\}.$$

Подставляя f_9 в f_8, получаем

$$f_8 = \min\{11,\ 17 + 6,\ f_8 + 6 + 6\} = 11.$$

Окончательно имеем: $f_9 = 17$; $f_{10} = 14$.

Таким образом, кратчайший путь равен $(1 - 4 - 6 - 8 - 10)$.

10.3. Метод Дейкстры

Алгоритм Дейкстры предназначен для нахождения кратчайшего пути между вершинами в неориентированном графе.

Идея алгоритма следующая: сначала выберем путь до начальной вершины равным нулю, и заносим эту вершину во множество выбранных вершин, расстояние от которых до оставшихся невыбранных вершин определено. На каждом следующем этапе находим следующую выбранную вершину, расстояние до которой наименьшее и соединенную ребром с какой-нибудь вершиной из множества невыбранных (это расстояние будет равно расстоянию до выбранной вершины плюс длина ребра).

Пример 1. Найти кратчайший путь на графе от вершины L до вершины D (рис. 10.9).

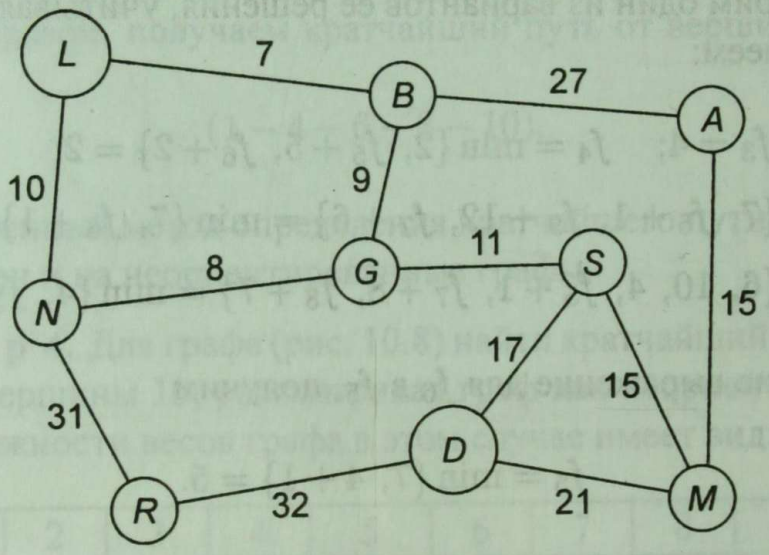


Рис. 10.9. Взвешенный неориентированный граф

Запишем алгоритм в виде последовательности шагов (табл. 10.1).

Таблица 10.1

Метод Дейкстры (первый шаг)

Выбранная вершина	Путь до выбранной вершины	Невыбранные вершины							
		B	*G*	*N*	*R*	*D*	*S*	*M*	*A*
L	0	(7)	∞	10	∞	∞	∞	∞	∞
B	7								

Ш а г 1. Определяются расстояния от начальной вершины L до всех остальных.

Ш а г 2. Выбираем наименьшее расстояние от L до B, найденная вершина B принимается за вновь выбранную. Найденное наименьшее расстояние добавляется к длинам ребер от новой вершины B до всех остальных. Выбирается минимальное расстояние от Bдо N. Новая вершина N принимается за выбранную (табл. 10.2).

Таблица 10.2

Метод Дейкстры (второй шаг)

Выбранная вершина	Путь до выбранной вершины	Невыбранные вершины							
		B	G	N	R	D	S	M	A
L	0	(7)	∞	10	∞	∞	∞	∞	∞
B	7	*	16	**(10)**	∞	∞	∞	∞	34
N	10								

Для наглядности в дальнейшем вместо знака ∞ будем ставить знак «–».

Ш а г 3. Определяются расстояния от вершины N до всех оставшихся (за исключением L и B) (табл. 10.3).

Таблица 10.3

Метод Дейкстры (третий шаг)

Выбранная вершина	Путь до выбранной вершины	Невыбранные вершины							
		B	G	N	R	D	S	M	A
L	0	(7)	∞	10	∞	∞	∞	∞	∞
B	7	*	16	**(10)**	∞	∞	∞	∞	34
N	10	*	**(16)**	*	41	∞	∞	∞	34
G	16								

Расстояние от вершины L через вершину N до вершины G равно 18. Это расстояние больше, чем расстояние $LB + BG = 16$, поэтому оно не учитывается в дальнейшем. Продолжая аналогичные построения, построим табл. 10.4 (величины, указанные в квадратных скобках, не учитываются). Таким образом, найдена длина кратчайшего пути между вершинами L и D (44 условные единицы).

Траектория пути определяется следующим образом. Осуществляем обратный просмотр от конечной вершины к начальной. Просматриваем столбец, соответствующий вершине, снизу вверх и фиксируем

первое появление минимальной величины. В столбце, соответствующем вершине D, впервые минимальная длина 44 появилась снизу в четвертой строке. В этой строке указана вершина S, к которой следует перейти, т.е. следующим нужно рассматривать столбец, соответствующий вершине S.

Таблица 10.4

Табличная реализация метода Дейкстры

Выбранная вершина	Путь до выбранной вершины	Невыбранные вершины							
		B	G	N	R	D	S	M	A
L	0	(7)	–	10	–	–	–	–	–
B	7	*	16	**(10)**	–	–	–	–	34
N	10	*	**(16)**	*	41	–	–	–	34
G	16	*	*	*	41	–	16+11 =(**27**)	–	34
S	27	*	*	*	41	27+17 =44	*	27+15 =42	**(34)**
A	34	*	*	*	**(41)**	44	*	42 [34+15]	*
R	41	*	*	*	*	44 [41+32]	*	**(42)**	*
M	42	*	*	*	*	(**44**) [42+21]	*	*	*

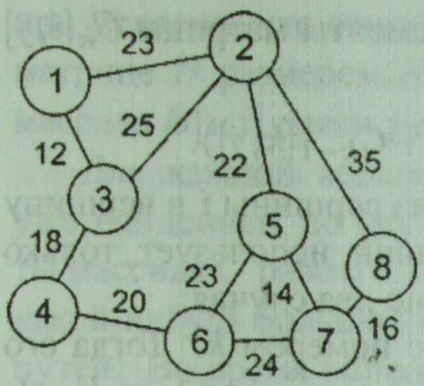
В этом столбце минимальная длина, равная 27, указывает следующую вершину G, к которой следует перейти, и т.д. Таким образом, получаем траекторию пути: $(L - B - G - S - D)$.

Пример 2. Найти кратчайший путь на графе между 1-й и 7-й вершинами (рис. 10.10).

Определяем следующую выбранную вершину, расстояние до которой наименьшее, и соединенную ребром с одной из вершин, принадлежащих множеству невыбранных. Это расстояние будет равно расстоянию до выбранной вершины плюс длина ребра (табл. 10.5).

Осуществляем обратный просмотр от конечной вершины к начальной. Просматриваем столбец, соответствующий вершине, снизу вверх и фиксируем первое появление минимальной величины.

Кратчайший путь при этом будет равен $(V_7 - V_5 - V_2 - V_1)$.



0	23	12	∞	∞	∞	∞	∞
23	0	25	∞	22	∞	∞	35
12	25	0	18	∞	∞	∞	∞
∞	∞	18	0	∞	20	∞	∞
∞	22	∞	∞	0	23	14	∞
∞	∞	∞	20	23	0	24	∞
∞	∞	∞	∞	14	24	0	16
∞	35	∞	∞	∞	∞	16	0

Рис. 10.10. Граф и соответствующая ему матрица смежности

Таблица 10.5

Табличная реализация метода Дейкстры

Выбранная вершина	Путь до выбранной вершины	Невыбранные вершины						
		V_2	V_3	V_4	V_5	V_6	V_7	V_8
V_1	0	23	**(12)**	–	–	–	–	–
V_2	23	*	*	**(30)**	45	–	–	58
V_3	12	**(23)**	*	30	–	–	–	–
V_4	30	*	*	*	**(45)**	50	–	58
V_5	45	*	*	*	*	**(50)**	59	58
V_6	50	*	*	*	*	*	59	**(58)**
V_8	58	*	*	*	*	*	**(59)**	*

10.4. Алгоритм Флойда

Пусть в матрице $A[i,j]$ записаны длины ребер графа (элемент $A[i,j]$ равен весу ребра, соединяющего вершины с номерами i и j. Если же такого ребра нет, то в соответствующем элементе записано некоторое очень большое число. Построим новые матрицы $C_k[i,j]$ ($k = 0, \ldots, N$). Элемент матрицы $C_k[i,j]$ будет равен минимально возможной длине такого пути из i в j, что в качестве промежуточных вершин в этом пути используются вершины с номерами от 1 до k, т.е. рассматриваются пути, которые могут проходить через вершины с номерами от 1 до k, но заведомо не проходят через вершины с номерами от $k+1$ до N. В матрицу записывается длина кратчайшего из таких путей. Если таких путей не существует, записывается то же большое число, которым обозначается отсутствие ребра.

Если вычислена матрица $C_{k-1}[i,j]$, то элементы матрицы $C_k[i,j]$ можно вычислить по следующей формуле

$$C_k[i,j] = \min(C_{k-1}[i,j], C_{k-1}[i,k] + C_{k-1}[k,j]).$$

В самом деле, рассмотрим кратчайший путь из вершины i в вершину j, который в качестве промежуточных вершин использует только вершины с номерами от 1 до k. Тогда возможно два случая:

• этот путь не проходит через вершину с номером k. Тогда его промежуточные вершины — это вершины с номерами от 1 до $k-1$. Но тогда длина этого пути уже вычислена в элементе $C_{k-1}[i,j]$;

• этот путь проходит через вершину с номером k. Но тогда его можно разбить на две части: сначала из вершины i доходим оптимальным образом до вершины k, используя в качестве промежуточных вершины с номерами от 1 до $k-1$ (длина такого оптимального пути вычислена в $C_{k-1}[i,k]$), а потом от вершины k идем в вершину j опять же оптимальным способом, используя в качестве промежуточных вершин только вершины с номерами от 1 до k ($C_{k-1}[k,j]$).

Выбирая из этих двух вариантов минимальный, получаем $C_k[i,j]$.

Последовательно вычисляя матрицы C_0, C_1, C_2 и т.д., получим искомую матрицу C_N кратчайших расстояний между всеми парами вершин в графе.

Алгоритм Флойда можно свести к последовательности шагов.

Присвоить $c_{ij} = 0$ для всех $i = 1, 2, \ldots, n$ и $c_{ij} = \infty$, если в графе отсутствует дуга $[x_i, x_j]$.

Присвоение начальных значений.

Шаг 1. Присвоить $k = 0$.

Шаг 2. $k = k + 1$.

Шаг 3. Для всех $i \neq k$, таких, что $c_{ik} \neq \infty$, и для всех $j \neq k$, таких, что $c_{ik} \neq \infty$, $c_{ij} = \min\{c_{ij},\ c_{ik} + c_{kj}\}$.

Проверка на окончание.

Шаг 4. Если $k = n$, то матрица $[c_{ij}]$ дает длины всех кратчайших путей. Остановка. Иначе перейти к шагу 2.

10.5. Алгоритм Йена

Пусть граф задан матрицей $A[i,j]$. Вершина s выбрана как начальная. Найдем длины k наименьших путей до каждой вершины от вершины s (ребра в одном пути могут повторяться неоднократно),

при условии, что эти пути существуют. Результат будет храниться в матрице B размером $N \times k$, где N — количество вершин. Элемент массива $B[i,j]$ равен j-му по длине пути до вершины i.

Для каждой вершины в массиве C будем хранить количество уже найденных до нее наименьших путей. Изначально все элементы массива C равны нулю. Все элементы матрицы B делаем равными какой-нибудь большой константе, заведомо бóльшей всех возможных путей. Во время исполнения алгоритма в матрице B будем хранить лучшие k путей до каждой вершины, найденные во время исполнения, при этом первые $C[i]$ путей для вершины i уже найдены окончательно (элементы матрицы B — $B[i,1]$, $B[i,2]$, ..., $B[i,k]$ для всех i упорядочены в возрастающем порядке).

Следовательно, можно действовать следующим образом: пусть уже найдены какие-то кратчайшие пути, тогда чтобы найти следующий по длине, попробуем удлинить каждый из уже полученных на одно ребро. Найдем наикратчайший из них, причем оканчивающийся на вершину, до которой найдено менее k путей. Его можно вносить в таблицу результата.

Алгоритм Йена можно свести к последовательности шагов.

Присвоение начальных значений.

Шаг 1. Найти P^1. Присвоить $k = 2$. Если существует только один кратчайший путь P^1, включить его в список L_0 и перейти к шагу 2. Если таких путей несколько, но меньше, чем K, включить один и них в список L_c, а остальные в список L_1. Перейти к шагу 2. Если существует K или более кратчайших путей P^1, то задача решена. Остановка.

Нахождение отклонений.

Шаг 2. Найти все отклонения P_i^k $(k-1)$-го кратчайшего пути P^{k-1} для всех $i = 1, 2, \ldots, q_{k-1}$, выполняя для каждого i шаги с 3-го по 6-й.

Шаг 3. Проверить, совпадает ли подпуть, образованный первыми i вершинами любого из P^j путей $(j = 1, 2, \ldots, k-1)$. Если да, то присвоить $c(x_i^{k-1}, x_{i+1}^j) = \infty$; иначе ничего не изменять. (При выполнении алгоритма вершина x_1 обозначается s.)

Шаг 4. Найти кратчайшие пути S_i^k от x_i^{k-1} к t, исключая из рассмотрения вершины s, x_i^{k-1}, x_3^{k-1}, ..., x_i^{k-1}. Если существует несколько кратчайших путей, то взять в качестве S_i^k один из них.

Шаг 5. Построить P_i^k и поместить P в список L_1.

Ш а г 6. Заменить элементы матрицы весов, измененные на шаге их первоначальными значениями и возвратиться к шагу 3.

Выбор кратчайших отклонений.

Ш а г 7. Найти кратчайший путь в списке L_1. Обозначить этот путь P^k и переместить его из L_1 в L_0. Если $k = K$, то остановка. Список L_0 — список K-кратчайших путей.

Если $k < K$, то присвоить $k = k + 1$, перейти к шагу 2. Если в L_1 имеется более чем один кратчайший путь (h-путь), то поместить в L_0 любой из них и продолжать вычисления до тех пор, пока увеличенное на h число путей, уже находящихся в L_1, не совпадет с K или не превысит его. Тогда — остановка.

10.6. Алгоритм Беллмана — Форда

Пусть в матрице $A[i,j]$ записаны длины ребер графа. Найдем кратчайшие расстояния от заданной вершины s до всех остальных вершин графа. Алгоритм Беллмана — Форда решает эту задачу даже при наличии ребер отрицательного веса. Обозначим через МинСт(s,ν,k) наименьшую стоимость проезда из s в ν менее чем с k пересадками:

$$\text{МинСт}(s,\nu,k+1) = \min\{\text{МинСт}(s,\nu,k),\ \text{МинСт}(s,i,k) + A[i,\nu]\}.$$

Искомым ответом является МинСт(s,i,n) для всех $i = 1..n$.

Чтобы найти не только длины наименьших путей до всех вершин, но и сами эти пути, используем следующий прием. Параллельно с вычислением массива x будем вычислять матрицу $D[i,j]$. Если между вершинами s и j существует путь, то в элементе массива $D[j,0]$ будет хранится количество вершин в этом пути, а цепочка вершин, составленная из элементов с $D[j,1]$ по $D[j,D[j,0]]$, будет этим самым путем. Путь до вершины s содержит единственную вершину ($D[s,0] = 1$; $D[s,1] = s$). Для вычисления матрицы D потребуется дополнить текст процедуры:

```
k:= 1;
for i := 1 to n do begin x[i] := a[s][i]; end;
{инвариант: x[i] := МинСт(s,i,k)}
```

Обозначим через МинСт$(s, i, к)$ наименьшую стоимость проезда из s в i менее чем с k пересадками. Тогда выполняется такое соотношение:

```
for i := 1 to n do begin D[i,0]:=2; D[i,1]:=s; D[i,2]:=i; End;
D[s,0]:=1; while k <> n do begin
    for i := 1 to n do begin
        for j := 1 to n do begin
            if x[i] > x[j]+a[j][i] then begin
                x[i] := x[j]+a[j][i];
                D[i]:=D[j];
                D[i,0]:=D[j,0]+1;
                D[i,D[i,0]]:=i;
            end;
        end
        {x[i] = МинСт(s,i,k+1)}
    end;
    k := k + 1;
end;
```

Контрольные вопросы

1. В чем особенность построения дерева решений?
2. Что показывает дерево решений?
3. Каким образом используется метод оптимизации для определения кратчайшего пути по карте местности?
4. В решении каких прикладных задач используются алгоритмы определения в графе кратчайших расстояний между заданными вершинами?
5. Может ли быть применен алгоритм Дейкстры к определению кратчайшего расстояния в ориентированном графе?
6. Как работает алгоритм Дейкстры?
7. Как работает алгоритм динамического программирования применительно к задаче определения в графе кратчайших расстояний между вершинами?
8. В чем особенность алгоритма Флойда?
9. В чем особенность алгоритма Йена?
10. Укажите основной принцип построения алгоритма Беллмана — Форда.

Глава 11
Эвристические алгоритмы

Эвристический алгоритм — это алгоритм, в котором на определенном этапе используется интуиция разработчика. Если решение, принятое разработчиком, окажется неверным, результат все равно будет получен, но за большее число шагов. Таким образом, в эвристических алгоритмах можно увеличить скорость получения правильного результата.

К эвристическим алгоритмам относятся: волновой, двухлучевой, четырехлучевой, маршрутный, алгоритмы составления расписания.

11.1. Волновой алгоритм

Волновой алгоритм, или алгоритм Ли, первоначально использовался для поиска пути в лабиринте или в игровых задачах. В настоящее время алгоритм Ли (волновой) является основным в микроэлектронике для трассировки (соединения) элементов интегральных схем. Особенность алгоритма состоит в следующем:

- в лабиринте (на подложке) выбираются две точки A (начальная) и B (конечная). Из начальной точки в четырех направлениях выходит волна (рис. 11.1). Цифрами обозначается номер фронта волны или ее путевые координаты;
- путевые координаты определяют шаг распространения волны. Каждый элемент первого фронта волны является источником вторичной волны (рис. 11.2).

	1	
1	A	1
	1	

Рис. 11.1. Направления распространения лучей

		2		
	2	1	2	
2	1	A	1	2
	2	1	2	
		2		

Рис. 11.2. Второй фронт волны

Элементы второго фронта генерируют третий фронт и т.д. От запрещенных элементов волна не распространяется. Процесс продолжается до тех пор, пока не будет достигнут конечный элемент.

Траектория пути определяется обратным просмотром, от конечного к начальному. При этом задаются приоритеты движения:

вверх, вниз, влево, вправо.

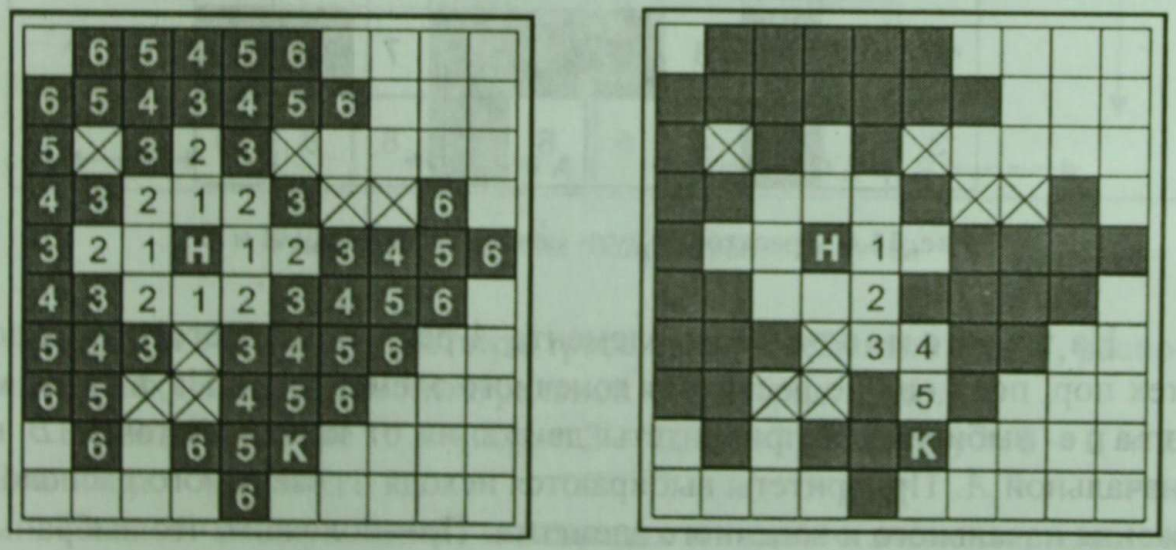
От того, в каком порядке заданы приоритеты, зависит скорость решения задачи.

При построении траектории используются два принципа:

- движение осуществляется строго по заданным приоритетам;
- при построении трассы, т.е. траектории движения, значения путевых координат должны уменьшаться.

Пример 1. Пусть задан лабиринт, где запрещенные элементы заштрихованы. Найти путь между элементами H и K.

На первом этапе от начального элемента трассы исходит волна, которая останавливается, достигнув конечного элемента (рис. 11.3). Волна распространяется по четырем направлениям: вверх, вниз, вправо и влево. Первый фронт волны представляет собой четыре единицы. Следующий фронт представляют собой распространение волны на одну клетку от элементов предыдущего фронта в стороны от начального элемента. При распространении волны элементам присваивают значения номера фронта (2, 3, 4 и т.д.). Если элемент запрещенный, то



а *б*

Рис. 11.3. Распространение волны (*а*) и построение трассы (*б*)

к нему это не относится — волна распространяется только по незапрещенным элементам сетки. Также волна не распространяется за границы координатной сетки.

Следующим этапом является нахождение пути, причем поиск ведется от конечного элемента (рис. 11.3). Выбираются приоритетные направления движения от конечного элемента к начальному: вверх, влево, вниз, вправо. Трасса прокладывается с учетом уменьшения путевых координат.

Пример 2. Пусть задан лабиринт, где запрещенные элементы заштрихованы. Найти путь между элементами A и B волновым алгоритмом (рис. 11.4).

6		10	9	8		10			
5				7		9	10		
4			5	6		8	9	B	
3	2		4	5		7	8		10
	1	2	3	4		6			9
1	A	1	2	3	4	5	6	7	8
2	1		3				7		
3	2		4	5	6		8	9	10

Рис. 11.4. Траектория пути между элементами A и B

На первом этапе от элемента A распространяется волна до тех пор, пока она не достигнет конечного элемента B. На втором этапе выбираются приоритеты движения от конечной точки B к начальной A. Приоритеты выбираются исходя из взаимного расположения начального и конечного элемента. Предположим, что выбраны парадоксальные (логически неверные) приоритеты : вверх, вправо, вниз, влево. В этом случае трасса все равно будет построена, но за большее число шагов (сравнений).

11.2. Двухлучевой алгоритм

В двухлучевом алгоритме из начального и конечного элементов одновременно выходят по два луча, трасса считается проведенной, если пересекаются два разноименных луча (от разных источников). Если на пути луча встречается запрещенный элемент, его обход осуществляется по второму приоритетному направлению, характерному для лучей, выходящих из одной точки.

Если же оба направления оказываются заблокированными запрещенными элементами либо достигнут край координатной сетки, то движение луча прекращается.

Выбор направления движения лучей происходит исходя из интуиции разработчика. Существуют четыре варианта распространения лучей, которые определяются после вычисления значений α и β:

$$\alpha = \begin{cases} 1, & \text{если} \quad X_A - X_B \geq 0; \\ 0, & \text{если} \quad X_A - X_B < 0; \end{cases} \qquad \beta = \begin{cases} 1, & \text{если} \quad Y_A - Y_B \geq 0; \\ 0, & \text{если} \quad Y_A - Y_B < 0, \end{cases}$$

где (X_A, Y_A) — координаты начального элемента;
(X_B, Y_B) — координаты конечного элемента.

Исходя из значений α и β выбирают направления лучей (рис. 11.5).

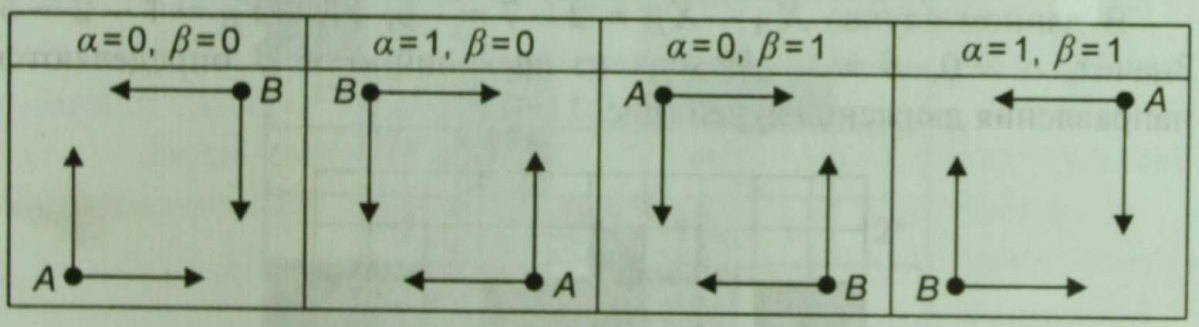


Рис. 11.5. Варианты распространения лучей

Пример 1. Осуществить трассировку элементов A и B, расположенных на подложке интегральной схемы (рис. 11.6).

В данном случае $X_A - X_B = 3 - 10 = -7$, $Y_A - Y_B = 9 - 2 = 7$. Значит, $\alpha = 0$, $\beta = 1$.

Каждый из двух лучей, выходящих из одного элемента, движется по заданному приоритетному направлению (рис. 11.5). Если луч встречает на пути запрещенный элемент, то его обход осуществляется по второму приоритетному направлению. Если оба направления

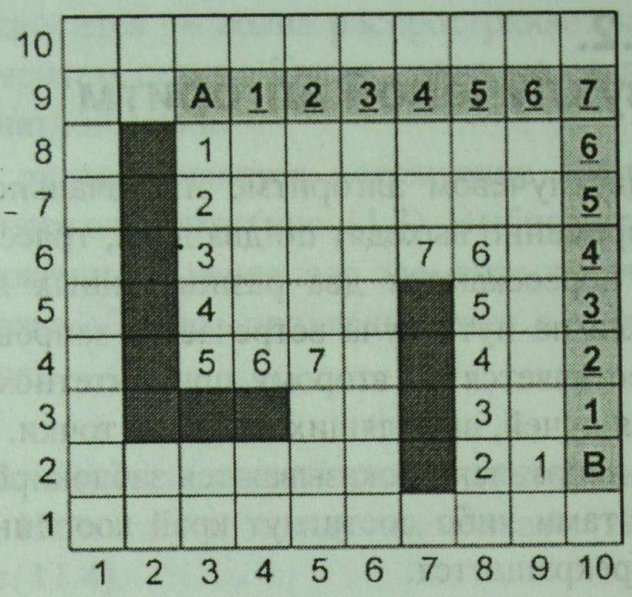


Рис. 11.6. Построение трассы двухлучевым алгоритмом

оказываются заблокированными запрещенными элементами, то движение луча прекращается. При достижении лучом края координатной сетки он затухает.

Трасса считается найденной, если встречаются два разноименных луча, т.е. лучи от разных источников.

Пример 2. Осуществить трассировку элементов A и B, расположенных на подложке интегральной схемы (рис. 11.7).

В данном случае $X_A - X_B = 2 - 7 = -5$, $Y_A - Y_B = 7 - 2 = 5$. Значит, $\alpha = 0$, $\beta = 1$. Исходя из значений α и β определяются направления движения лучей (рис. 11.5).

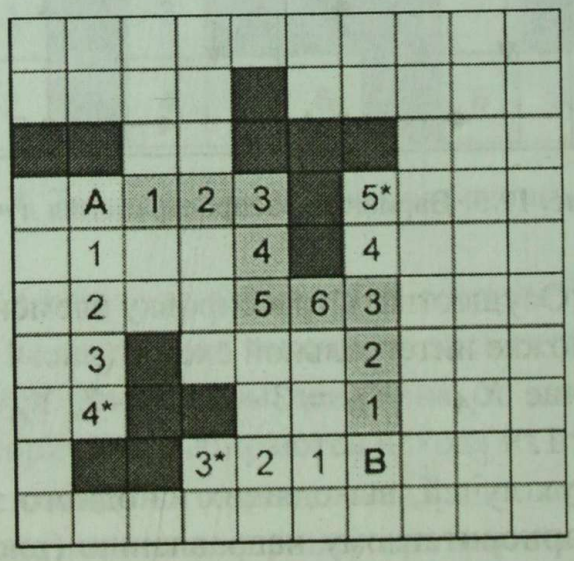


Рис. 11.7. Трассировка двухлучевым алгоритмом
(* — луч прекращает движение)

11.3. Четырехлучевой алгоритм

Из начальной и конечной точек выходят одновременно по четыре луча (рис. 11.8). Лучи движутся строго по заданным направлениям и «затухают», если достигли края координатной сетки либо встретили запрещенный элемент.

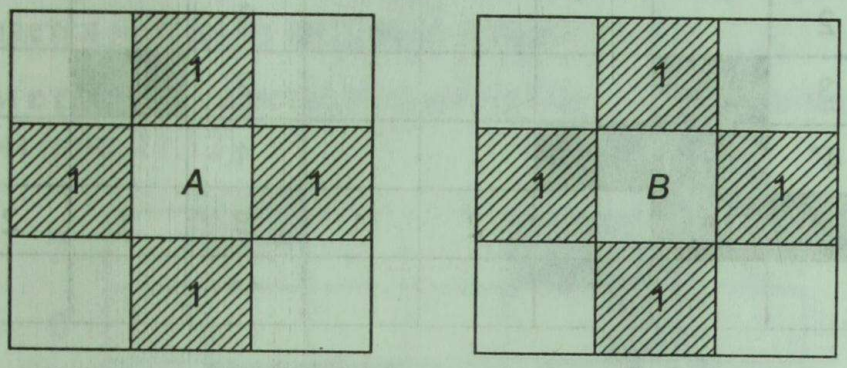


Рис. 11.8. Направления движения лучей из точек *A* и *B*

Пример 1. Осуществить трассировку (соединение) элементов интегральной схемы четырехлучевым алгоритмом (рис. 11.9).

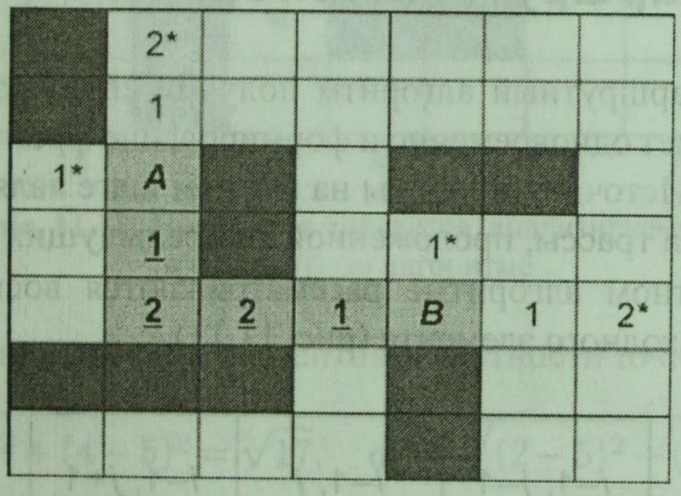


Рис. 11.9. Трассировка четырехлучевым алгоритмом

Если луч встречает на пути запрещенный элемент или достигает края координатной сетки, то он затухает. Трасса считается найденной, если встречаются два разноименных луча.

Пример 2. Осуществить трассировку элементов четырехлучевым алгоритмом (рис. 11.10). В данном примере трассу провести нельзя.

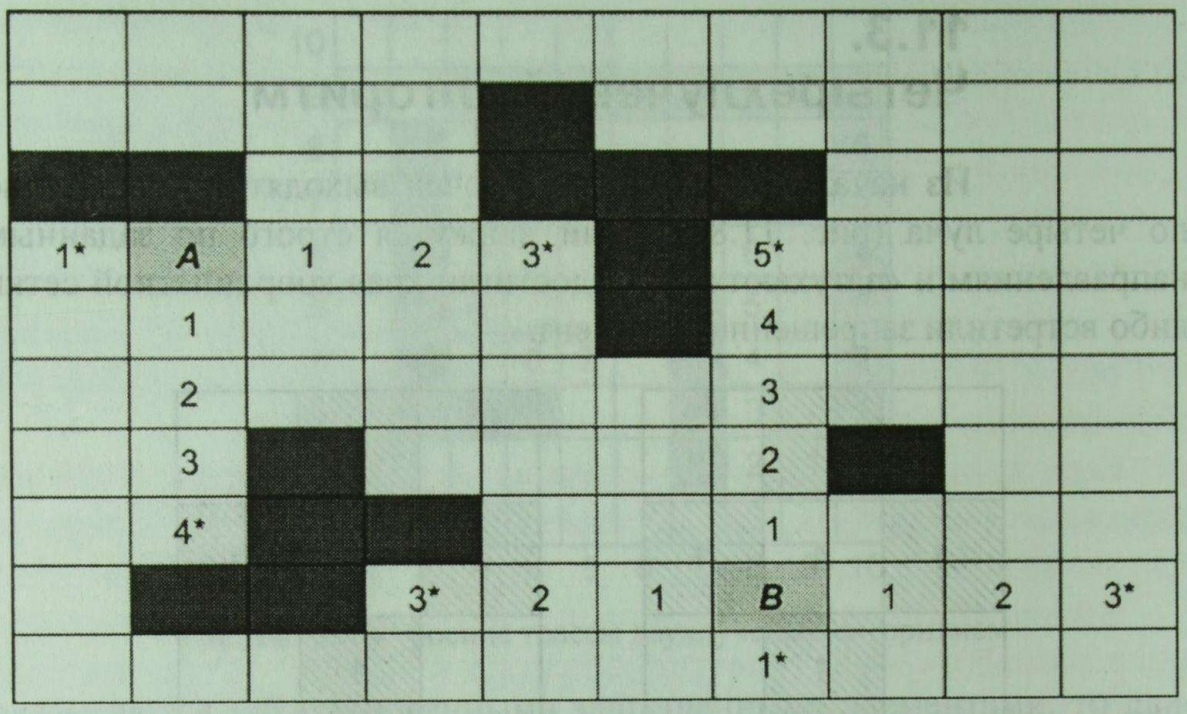


Рис. 11.10. Трассировка четырехлучевым алгоритмом

11.4. Маршрутный алгоритм

Маршрутный алгоритм получил свое название, потому что осуществляет одновременно и формирование фронта, и прокладывание трассы. Источником волны на каждом шаге является конечный элемент участка трассы, проложенной на предыдущих шагах.

В маршрутном алгоритме рассматриваются восьмиэлементная окрестность исходного элемента (рис. 11.11).

$i-1, j-1$	$i-1, j$	$i-1, j+1$
$i, j-1$	**A**	$i, j+1$
$i+1, j-1$	$i+1, j$	$i+1, j+1$

Рис. 11.11. Восьмиэлементная окрестность

От каждого элемента окружения оценивается расстояние d до конечного элемента B: $d = \sqrt{(x_i - x_b)^2 + (y_i - y_b)^2}$ или $d = |x_i - x_b| + |y_i - y_b|$.

Таким образом определяются восемь значений расстояний, из которых выбирается минимальное. Элемент, для которого d оказалось минимальным, считаем элементом трассы. Процесс повторятся до тех пор, пока расстояние не будет равным нулю ($d = 0$), т.е. пока не будет достигнут конечный элемент. Обход запрещенных элементов осуществляется исходя из интуиции разработчика.

Пример 1. Осуществить трассировку элементов маршрутным алгоритмом (рис. 11.12).

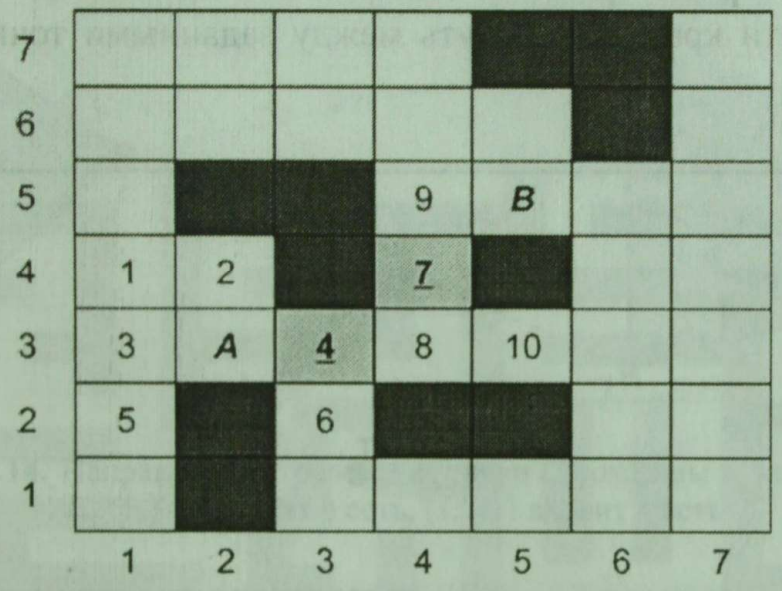


Рис. 11.12. Нумерация элементов, анализируемых в маршрутном алгоритме

Определим расстояния элементов окрестности точки A до точки B:

$$d_1 = \sqrt{(1-5)^2 + (4-5)^2} = \sqrt{17}, \quad d_2 = \sqrt{(2-5)^2 + (4-5)^2} = \sqrt{10},$$

$$d_3 = \sqrt{(1-5)^2 + (3-5)^2} = \sqrt{20}, \quad d_4 = \sqrt{(3-5)^2 + (3-5)^2} = \sqrt{8},$$

$$d_5 = \sqrt{(1-5)^2 + (2-5)^2} = \sqrt{25}, \quad d_6 = \sqrt{(3-5)^2 + (2-5)^2} = \sqrt{13}.$$

Минимальным является расстояние $d_4 = \sqrt{8}$. Далее считаем расстояния от точки B до точек окрестности найденной точки:

$$d_7 = \sqrt{2}, \quad d_8 = \sqrt{5}.$$

Минимальное расстояние d_7. Проводим вычисления далее:

$$d_9 = 1, \quad d_{10} = \sqrt{2}, \quad d_B = 0.$$

Получили маршрут $(A - 4 - 7 - B)$.

11.5. Геометрическая модель задачи о лабиринте

Пример 1. В лабиринте с произвольными препятствиями найти кратчайший путь между заданными точками S и T (рис. 11.13).

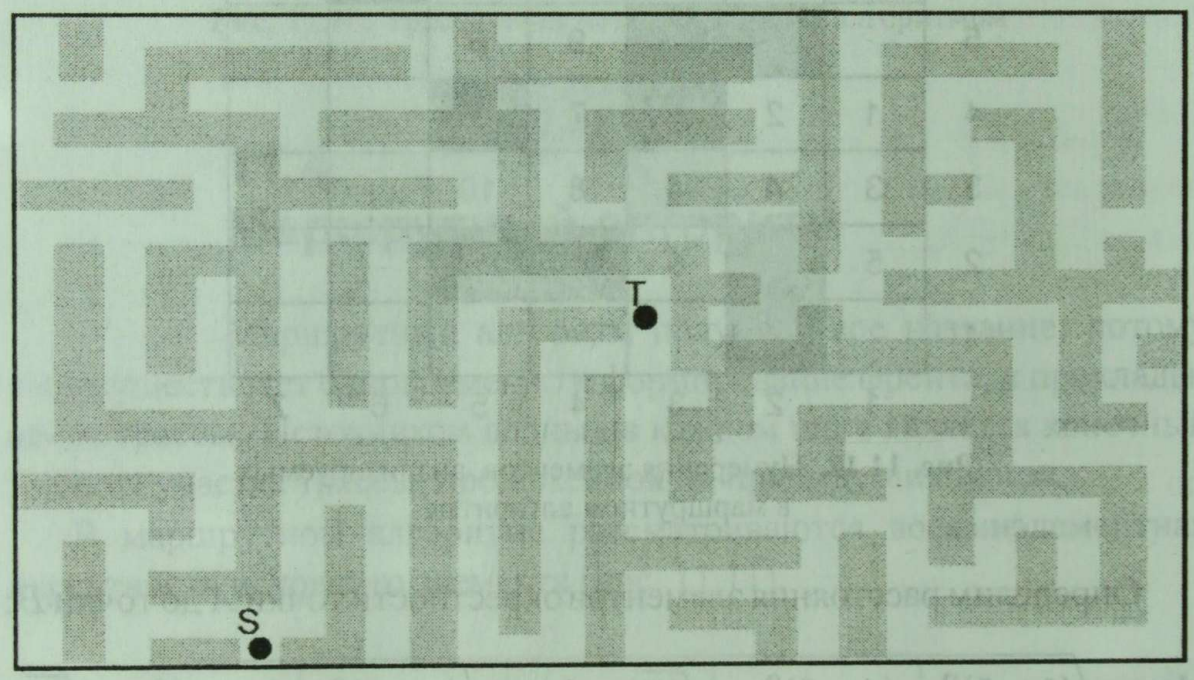


Рис. 11.13. Исходный лабиринт

Решение. Так как препятствия на местности образуют многоугольники или какие-либо другие геометрические фигуры (которые с некоторыми погрешностями тоже можно изобразить в виде многоугольников), то кратчайшая трасса будет являться ломаной с узлами в вершинах этих многоугольников. Звено ломаной — это либо сторона многоугольника, либо прямолинейный отрезок, проходящий вне многоугольников и соединяющий две вершины одного и того же или

разных многоугольников. Для решения этой задачи нужно построить сеть (ломаную), а также соединить точки s и t с простреливаемыми из них вершинами, если эти точки не являются вершинами многоугольников.

Формирование сети, т.е. матрицы расстояний C размером $n \times n$ (n — общее число вершин всех многоугольников плюс два для учета старта и финиша) представляет собой тройной цикл: внешний по i — перебор вершин, откуда стреляют; средний по j (j от $i+1$ до n, чтобы не повторяться) — это перебор вершин, куда стреляют; и внутренний по k — это проверка, не пересекает ли k-я сторона какого-либо многоугольника отрезок соединения (рис. 11.14).

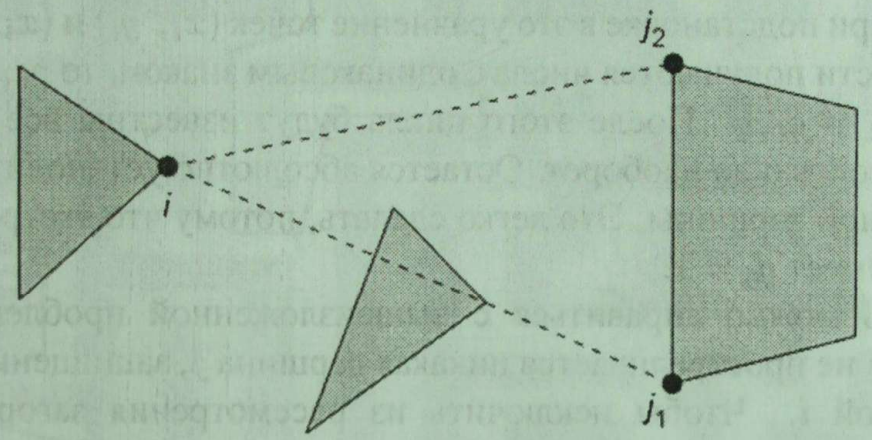


Рис. 11.14. Направления стрельбы из точки i (показаны пунктиром): $(i,\ j_1)$ не входит в сеть, $(i,\ j_2)$ входит в сеть

Последнее условие проверяется по стандартным формулам аналитической геометрии: записывается уравнение прямой, проходящей через $(i,\ j)$, и уравнение прямой, проходящей через концы отрезка k. Решением системы из этих двух уравнений находится точка пересечения и устанавливается, лежит ли точка пересечения внутри рассматриваемых отрезков (рис. 11.15). Если да, то $d_{ij} = \infty$, конец цикла по k; если нет пересечения по окончании цикла по k, то вычисляется евклидово расстояние d_{ij}.

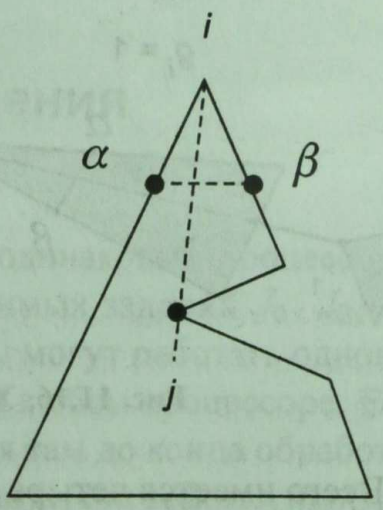


Рис. 11.15. Определение точки пересечения отрезков

Пусть между вершинами i и j не проходит никакой стены, а j из i не простреливается. Чтобы преодолеть эту трудность, нужно ввести характеристику i угла препятствия g_i, присвоив $g_i = 0$, если «вогнутый» угол, или $g_i = 1$, если «выпуклый» угол. Так, для угла с вершиной i $g_i = 1$, а для угла с вершиной j $g_i = 0$.

Если крайние вершины x_i и x_{i+3} ($x_i, x_{i+1}, x_{i+2}, x_{i+3}$ — последовательные вершины многоугольника) лежат по одну сторону от прямой, проходящей через соседние вершины x_{i+1}, x_{i+2}, то $g_{i+1} = g_{i+2}$, иначе $g_{i+1} \neq g_{i+2}$:

$$(x - x_{i+1})(y_{i+2} - y_{i+1}) - (x_{i+2} - x_{i+1})(y - y_{i+1}) = 0.$$

Если при подстановке в это уравнение точек $(x_i,\ y_i)$ и $(x_{i+3},\ y_{i+3})$ в левой части получаются числа с одинаковым знаком, то $g_{i+1} = g_{i+2}$, иначе $g_{i+1} \neq g_{i+2}$. После этого цикла будут известны все g_i точно или с точностью до наоборот. Остается абсолютно установить g_i хотя бы для одной вершины. Это легко сделать, потому что экстремальная вершина имеет $g_0 = 1$.

Теперь можно справиться с вышеизложенной проблемой. Из вершины i не простреливается никакая вершина j, защищенная углом с вершиной i. Чтобы исключить из рассмотрения загороженные вершины, нужно отступить от вершины i по сторонам угла на величину ε, заведомо меньшую, чем длина стороны, построив таким образом точки α и β. После этого нужно ввести бинарную величину B, полагая $B = 1$, если отрезки $\alpha\beta$ и ij пересекаются, и $B = 0$, если отрезки $\alpha\beta$ и ij не пересекаются (рис. 11.16).

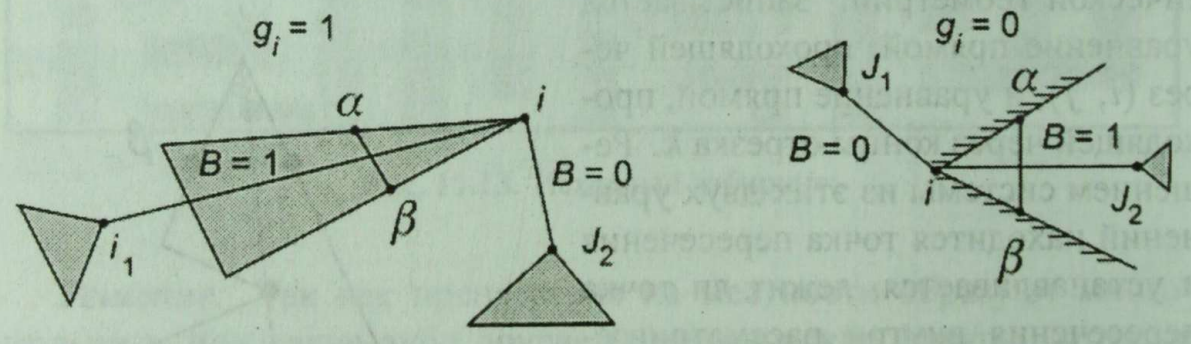


Рис. 11.16. Характеристики угла препятствия

Всего имеется четыре возможности:

1) $B = 1$ и $g_i = 0$;

2) $B = 0$ и $g_i = 1$;

3) $B = 1$ и $g_i = 0$;
4) $B = 1$ и $g_i = 1$.

Ясно, что вершина j не простреливается в случаях 2 и 3 (при нечетном $B + g$). Теперь можно построить сеть.

После того как сеть построена, можно приступать к нахождению кратчайших путей, воспользовавшись любым из рассмотренных выше алгоритмов (в зависимости от поставленной задачи) — рис. 11.17.

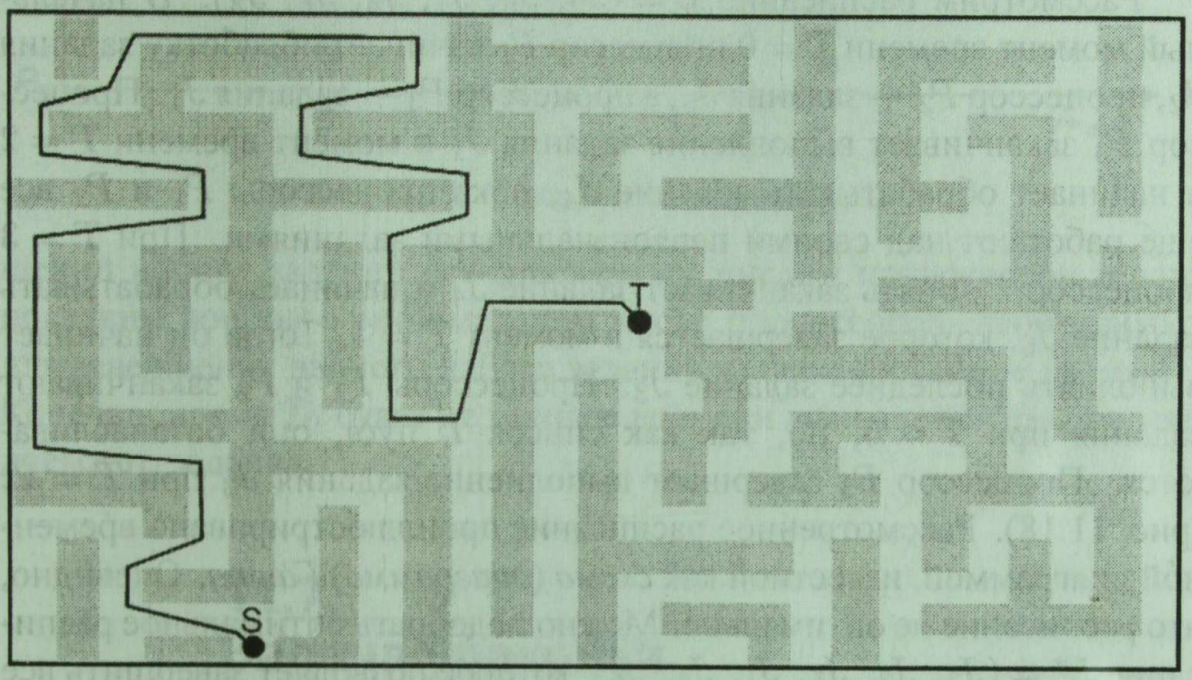


Рис. 11.17. Траектория пути в лабиринте

11.6. Алгоритмы составления расписания

Предположим, что имеется n одинаковых процессоров, обозначенных P_1, P_2, ..., P_n, и m независимых заданий J_1, J_2, ..., J_m, которые нужно выполнить. Процессоры могут работать одновременно, и любое задание можно выполнять на любом процессоре. Если задание загружено в процессор, оно остается там до конца обработки. Время обработки задания J_i известно и равно t_i, $i = 1, 2, \ldots, m$. Организовать обработку заданий таким образом, чтобы выполнение всего набора заданий было завершено как можно быстрее.

Система работает следующим образом: первый освободившийся процессор берет из списка следующее задание. Если одновременно освобождаются два или более процессоров, то выполнять очередное задание из списка будет процессор с наименьшим номером.

Пример. Пусть имеются три процессора и шесть заданий, время выполнения каждого из которых $t_1 = 2$, $t_2 = 5$, $t_3 = 8$, $t_4 = 1$, $t_5 = 5$, $t_6 = 1$.

Рассмотрим расписание $L = (J_2, J_5, J_1, J_4, J_6, J_3)$. В начальный момент времени $T = 0$ процессор P_1 начинает обработку задания J_2, процессор P_2 — задания J_5, а процессор P_3 — задания J_1. Процессор P_3 заканчивает выполнение задания J_1 в момент времени $T = 2$ и начинает обрабатывать задание J_4, пока процессоры P_1 и P_2 все еще работают над своими первоначальными заданиями. При $T = 3$ процессор P_3 опять заканчивает задание J_4 и начинает обрабатывать задание J_6, которое завершается в момент $T = 4$. Тогда он начинает выполнять последнее задание J_3. Процессоры P_1 и P_2 заканчивают задания при $T = 5$, но, так как список L пуст, они останавливаются. Процессор P_3 завершает выполнение задания J_3 при $T = 12$ (рис. 11.18). Рассмотренное расписание проиллюстрировано временно́й диаграммой, известной как *схема* (*диаграмма*) *Ганта*. Очевидно, что расписание не оптимально. Можно подобрать оптимальное расписание $L^* = (J_3, J_2, J_5, J_1, J_4, J_6)$, которое позволяет завершить все задания за минимальное время $T^* = 8$ единиц (рис. 11.19).

Можно рассмотреть другой тип задач по составлению расписания для многопроцессорных систем. Вместо вопроса о быстрейшем завер-

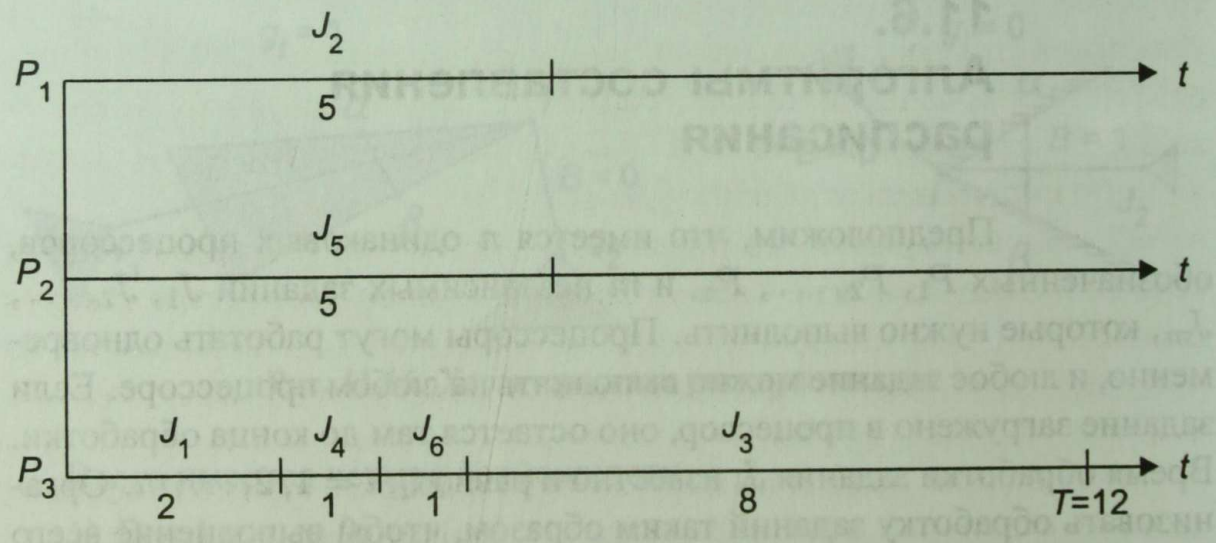


Рис. 11.18. Схема Ганта: расписание L

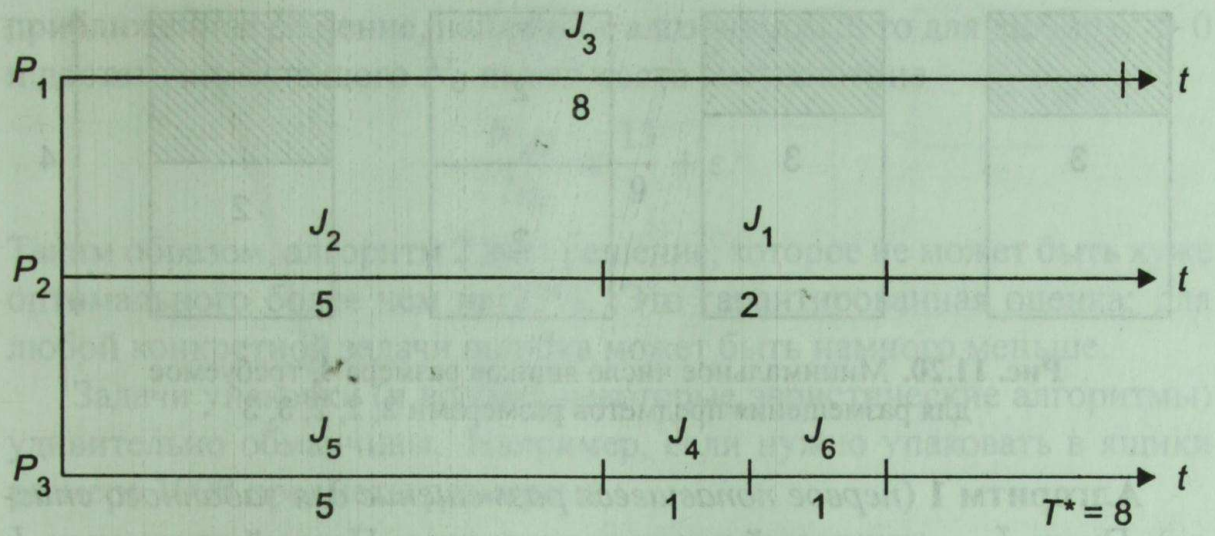


Рис. 11.19. Схема Ганта: расписание L^*

шении набора заданий фиксированным числом процессоров теперь поставим вопрос о минимальном числе процессоров, необходимых для завершения данного набора заданий за фиксированное время T_0. Конечно, время T_0 будет не меньше времени выполнения самого трудоемкого задания.

11.7. Задача упаковки

Задача составления расписания эквивалентна следующей задаче упаковки. Пусть каждому процессору P_j соответствует ящик B_j размера T_0. Пусть каждому заданию J_i соответствует предмет размера t_i, равного времени выполнения задания J_i, где $i = 1, 2, \ldots, n$. Теперь для решения задачи по составлению расписания нужно построить алгоритм, позволяющий разместить все предметы в минимальном количестве ящиков. Конечно, нельзя заполнять ящики сверх их объема T_0, и предметы нельзя дробить на части.

Пусть дано множество предметов: два из них имеют размер 3 и три — размер 2. Какое требуется минимальное количество ящиков размера 4, чтобы поместить в них все предметы? Ответ показан на рис. 11.20, где заштрихованные участки обозначают пустые места в ящиках.

Для решения этой задачи существует несколько простых эвристических алгоритмов. Каждый из них может быть описан в нескольких строках. Рассмотрим четыре таких алгоритма.

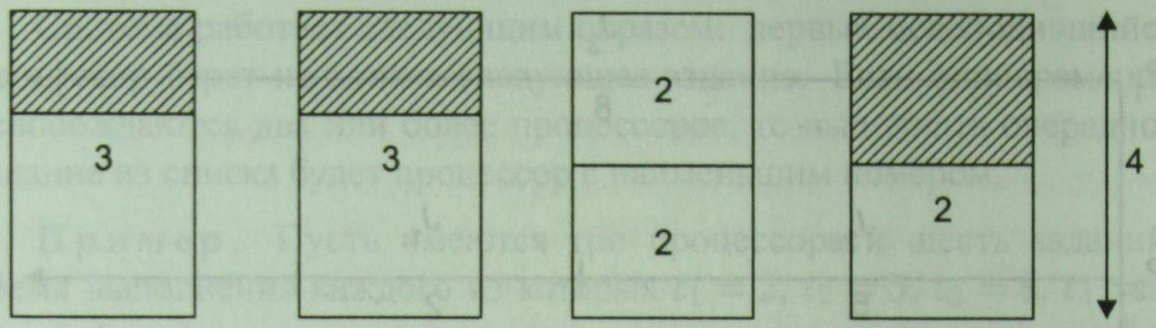


Рис. 11.20. Минимальное число ящиков размера 4, требуемое для размещения предметов размерами 2, 2, 2, 3, 3

Алгоритм 1 (*первое попавшееся размещение для заданного списка*). Пусть L — некоторый порядок предметов. Первый предмет из L кладем в ящик B_1. Второй предмет из L кладем в B_1, если он туда помещается. В противном случае кладем его в следующий ящик B_2. Вообще, очередной предмет из L кладем в ящик B_i, куда этот предмет поместится, причем индекс i наименьший среди всех возможных. Если предмет не помещается ни в один из k частично заполненных ящиков, кладем его в ящик B_{k+1}. Этот основной шаг повторяется, пока список L не будет исчерпан.

Алгоритм 2 (*первое попавшееся размещение с убыванием*). Этот алгоритм отличается от алгоритма 1 тем, что список L упорядочен от больших предметов к меньшим.

Алгоритм 3 (*лучшее размещение для заданного списка*). Пусть L — заданный список заданий. Основной шаг тот же, что и в алгоритме 1, но очередной предмет кладется в тот ящик, где остается наименьшее неиспользованное пространство. Таким образом, если очередной предмет имеет, например, размер 3 и есть четыре частично заполненных ящика размера 6, в которых осталось 2, 3, 4 и 5 единиц незаполненного пространства, тогда предмет кладется во второй ящик, и тот становится полностью заполненным.

Алгоритм 4 (*лучшее размещение с убыванием*). Этот алгоритм такой же, как и алгоритм 3, но список L упорядочен от больших предметов к меньшим.

Для задачи упаковки было найдено несколько хороших верхних оценок. Одна из лучших, принадлежащая Грэхему, выглядит следующим образом. Пусть N_0 — минимальное число необходимых ящиков, полученное при помощи точного алгоритма. Если N_{A2} обозначает

приближенное решение, найденное алгоритмом 2, то для любого $\varepsilon > 0$ и достаточно большого N_0 имеет место соотношение

$$\frac{N_{A2}}{N_0} < \frac{11}{9} + \varepsilon.$$

Таким образом, алгоритм 2 дает решение, которое не может быть хуже оптимального более чем на 23%. Это гарантированная оценка; для любой конкретной задачи ошибка может быть намного меньше.

Задачи упаковки (и вообще некоторые эвристические алгоритмы) удивительно обманчивы. Например, если нужно упаковать в ящики размера 1000 предметы размерами

$$L = (760,\ 395,\ 395,\ 379,\ 379,\ 241,\ 200,\ 105,\ 105,\ 40),$$

то эвристический алгоритм 2 потребует для этого $N_{A2} = 3$ ящика. Легко проверить, что это на самом деле оптимальное решение. Но если уменьшить размеры каждого предмета на 1, что приводит к списку

$$L' = (759,\ 394,\ 394,\ 378,\ 378,\ 240,\ 199,\ 104,\ 104,\ 39),$$

то эвристический алгоритм 2 уже даст $N_{A2} = 4$ ящика. Это очевидно не оптимальное и совершенно неожиданное решение.

Обратим внимание на другую аномалию. Применяя эвристический алгоритм 1, упакуем $L = (7,\ 9,\ 7,\ 1,\ 6,\ 2,\ 4,\ 3)$ в ящики размера 13. Находим, что $N_{A1} = 3$, как показано на рис. 11.21; очевидно, что этот результат оптимальный.

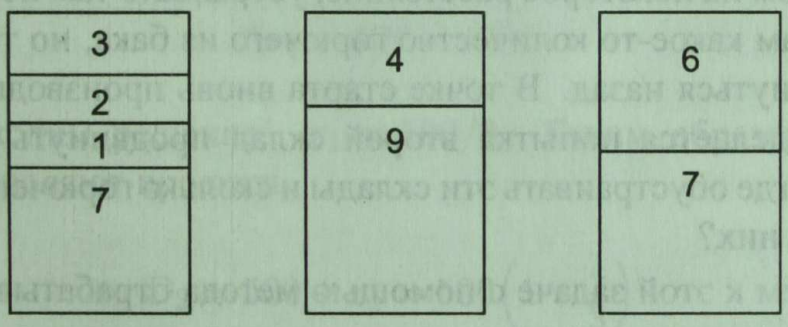


Рис. 11.21. Оптимальная упаковка

Уберем из списка предмет размера 1, что приводит к списку $L' = (7,\ 9,\ 7,\ 6,\ 2,\ 4,\ 3)$. Казалось бы должен получиться результат не хуже предыдущего. Однако нетрудно убедиться, что решение ухудшается. Это решение показано на рис. 11.22.

На рис. 11.22 заштрихованные участки показывают неиспользованные объемы тары.

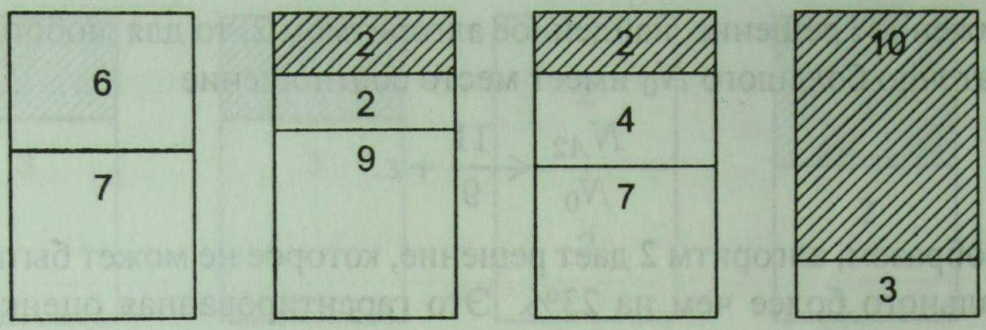


Рис. 11.22. Использование эвристического алгоритма первого попавшегося размещения

11.8. Задача о джипе

Пусть необходимо пересечь на джипе 1000-километровую пустыню, израсходовав при этом минимум горючего. Объем топливного бака джипа 500 л, горючее расходуется равномерно, по одному литру на километр. При этом в точке старта имеется неограниченный резервуар с топливом. Так как в пустыне нет складов с горючим, необходимо установить свои собственные хранилища и наполнять их топливом из бака машины.

Итак, идея задачи ясна: нужно из точки старта отъезжать с полным баком на некоторое расстояние, устраивать там первый склад, оставлять там какое-то количество горючего из бака, но такое, чтобы хватило вернуться назад. В точке старта вновь производится полная заправка и делается попытка второй склад продвинуть в пустыню дальше. Но где обустраивать эти склады и сколько горючего оставлять в каждом из них?

Подойдем к этой задаче с помощью метода отрабатывания назад. С какого расстояния от конца можно пересечь пустыню, имея запас горючего в точности k баков? Рассмотрим этот вопрос для $k = 1, 2, 3, \ldots$, пока не будет найдено такое целое n, что n полных баков позволяет пересечь всю 1000-километровую пустыню.

Для $k = 1$ ответ, очевидно, равен 500 км, как показано на рис. 11.23. Можно заправить машину в точке B и пересечь оставшиеся 500 км пустыни. Ясно, что это наиболее отдаленная точка, стартуя из которой можно преодолеть пустыню, имея в точности 500 л горючего.

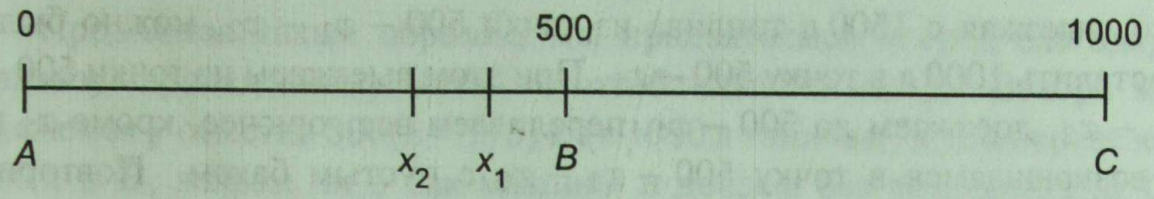


Рис. 11.23. Схема пути через пустыню

Вначале была поставлена частная цель: какое расстояние можно проехать на заданном количестве топлива?

Предположим, что $k = 2$, т.е. имеется два полных бака (1000 л). Будем рассматривать этот случай, опираясь на результат для $k = 1$. Данная ситуация иллюстрируется на рис. 11.23. Каково максимальное значение x_1, такое, что, отправляясь с 1000 л горючего из точки $500 - x_1$, можно перевезти достаточно горючего в точку, чтобы завершить поездку, как в случае $k = 1$?

Один из способов определения приемлемого значения x_1 состоит в следующем. Заправляемся в точке $500 - x_1$, едем x_1 километров до B и переливаем в хранилище все горючее, кроме литров, которые потребуются для возвращения в точку $500 - x_1$. В этой точке бак становится пустым. Теперь наполняем второй бак, проезжаем x_1 километров до B, забираем в B горючее, оставленное там, и из B едем в C с полным баком. Общее пройденное расстояние состоит из трех отрезков по x_1 километров и одного отрезка BC длиной 500 км. Тогда x_1 находим из уравнения

$$3x_1 + 500 = 1000.$$

Отсюда находим решение: $x_1 = 500/3$. Таким образом, два бака (1000 *л*) позволяют проехать

$$D_2 = 500 + x_1 = 500\left(1 + \frac{1}{3}\right).$$

Заметим, что исходная предпосылка является недальновидной и грубой. Когда имеются в распоряжении k баков горючего, мы просто стараемся продвинуться назад как можно дальше от точки, найденной для $k - 1$ баков.

Рассмотрим $k = 3$. Из какой точки можно выехать с 1500 л топлива так, что машина сможет доставить 1000 л в точку $500 - x_1$? Возвращаясь к рис. 11.23, найдем наибольшее значение x_2, такое,

что, выезжая с 1500 л топлива из точки $500 - x_1 - x_2$, можно было доставить 1000 л в точку $500 - x_1$. При этом выезжаем из точки $500 - x_1 - x_2$, доезжаем до $500 - x_1$, переливаем все горючее, кроме x_2 л, и возвращаемся в точку $500 - x_1 - x_2$ с пустым баком. Повторив эту процедуру, затратим $4x_2$ л на проезд и оставим $1000 - 4x_2$ л в точке $500 - x_1$. Теперь в точке $500 - x_1 - x_2$ осталось ровно 500 л. Заправляемся последними 500 л и едем в точку $500 - x_1$, израсходовав на это x_2 л.

Если находимся в точке $500 - x_1$, то на проезд будет затрачено $5x_2$ л топлива. Здесь оставлено в общей сложности $1500 - 5x_2$ л. Это количество должно быть равно 1000 л, т.е. $x_2 = 500/5$. Из этого заключаем, что 1500 литров позволяют проехать

$$D_3 = 500 + x_1 + x_2 = 500\left(1 + \frac{1}{3} + \frac{1}{5}\right) \text{ км.}$$

Продолжая индуктивно процесс отрабатывания назад, получаем, что n баков горючего позволяют нам проехать D_n км, где

$$D_n = 500\left(1 + \frac{1}{3} + \frac{1}{5} + \ldots + \frac{1}{2n-1}\right).$$

Нужно найти наименьшее значение n, при котором $D_n \geq 1000$. Простые вычисления показывают, что для $n = 7$ имеем $D_7 = 997{,}5$ км, т.е. семь баков, или 3500 л, топлива дадут возможность проехать 977,5 км. Полный восьмой бак — это было бы уже больше, чем нам потребуется, чтобы перевезти 3500 л из точки A в точку, отстоящую на 22,5 км (1000 – 977,5) от A. При этом для доставки 3500 л топлива к отметке 22,5 км достаточно 337,5 л. Таким образом, для того чтобы пересечь на машине пустыню из A в C, нужно 3837,5 л горючего.

Теперь алгоритм транспортировки горючего может быть представлен следующим образом. Стартуем из A, имея 3837,5 л. Здесь как раз достаточно топлива, чтобы постепенно перевезти 3500 л к отметке 22,5 км, где окажемся с пустым баком и запасом горючего на семь полных заправок. Этого топлива достаточно, чтобы перевезти 3000 л к точке, отстоящей на $22{,}5 + 500/13$ км от A, где бак машины будет опять пуст. Последующие перевозки приведут нас к точке, отстоящей на $22{,}5 + 500/13 + 500/11$ км от A, с пустым баком машины и 2500 л на складе.

Продолжая таким образом, мы продвигаемся вперед благодаря анализу, проведенному методом отрабатывания назад. Вскоре мы окажемся у отметки $500(1-1/3)$ км с 1000 л топлива. Затем перевезем 500 л в B, зальем их в бак машины и доедем без остановки до C. Рис. 11.24 иллюстрирует этот процесс.

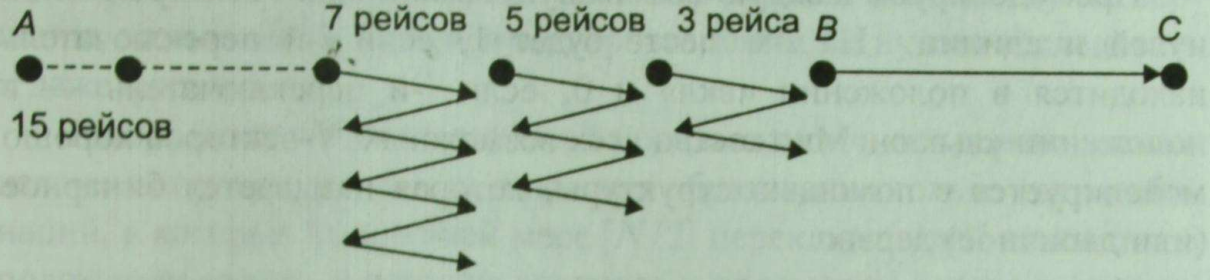


Рис. 11.24. Схема решения задачи о джипе

Возникает вопрос, можно ли проехать 1000 км, затратив меньше чем 3837,5 л горючего. Оказывается, что нельзя. Для этого можно высказать следующий, довольно правдоподобный довод. Очевидно, действуя наилучшим образом для $k=1$, при $k=2$ используем план для $k=1$ и затем вводим в действие второй бак горючего для того, чтобы оказаться как можно дальше от B. Исходная предпосылка для k баков заключается в том, чтобы определить, как действовать наилучшим образом в случае с $k-1$ баками, и отодвигаться как можно дальше назад с помощью k-го бака.

11.9. Задача о кодовом замке

Пусть кодовый замок состоит из набора N переключателей, каждый из которых может быть в положении «вкл» или «выкл». Замок открывается только при одном наборе положений переключателей, из которых не менее $[N/2]$ (целая часть от $N/2$), например, находятся в положении «вкл». Предположим, что забыта комбинация, но необходимо отпереть замок. Предположим также, что можно перепробовать все комбинации, что для замка с N переключателями приведет к перебору 2^N возможных комбинаций. Неплохие будут шансы решить задачу полным перебором всех комбинаций, если, скажем, $N \leq 10$. А если значительно боль-

ше 10? Здесь и пригодится использование условия $[N/2]$, которое позволит многие комбинации не просматривать. Важно лишь так построить алгоритм перебора комбинаций, чтобы не пропустить нужную и не набирать ту, которая заведомо к успеху не приведет.

Промоделируем каждую возможную комбинацию вектором из N нулей и единиц. На i-м месте будет 1, если i-й переключатель находится в положении «вкл» и 0, если i-й переключатель — в положении «выкл». Множество всех возможных N-векторов хорошо моделируется с помощью структуры, которая называется бинарное (или двоичное) дерево.

Каждая вершина k-го уровня этого дерева будет соответствовать определенному набору первых k компонент N-вектора. Две ветви, идущие вниз из вершины этого уровня, соответствуют двум возможным значениям $(k+1)$-й компоненты в N-векторе. Если количество переключателей в замке равно N, то в дереве просмотра будет N уровней. На рис. 11.25 конструкция бинарного дерева изображена для $N = 4$.

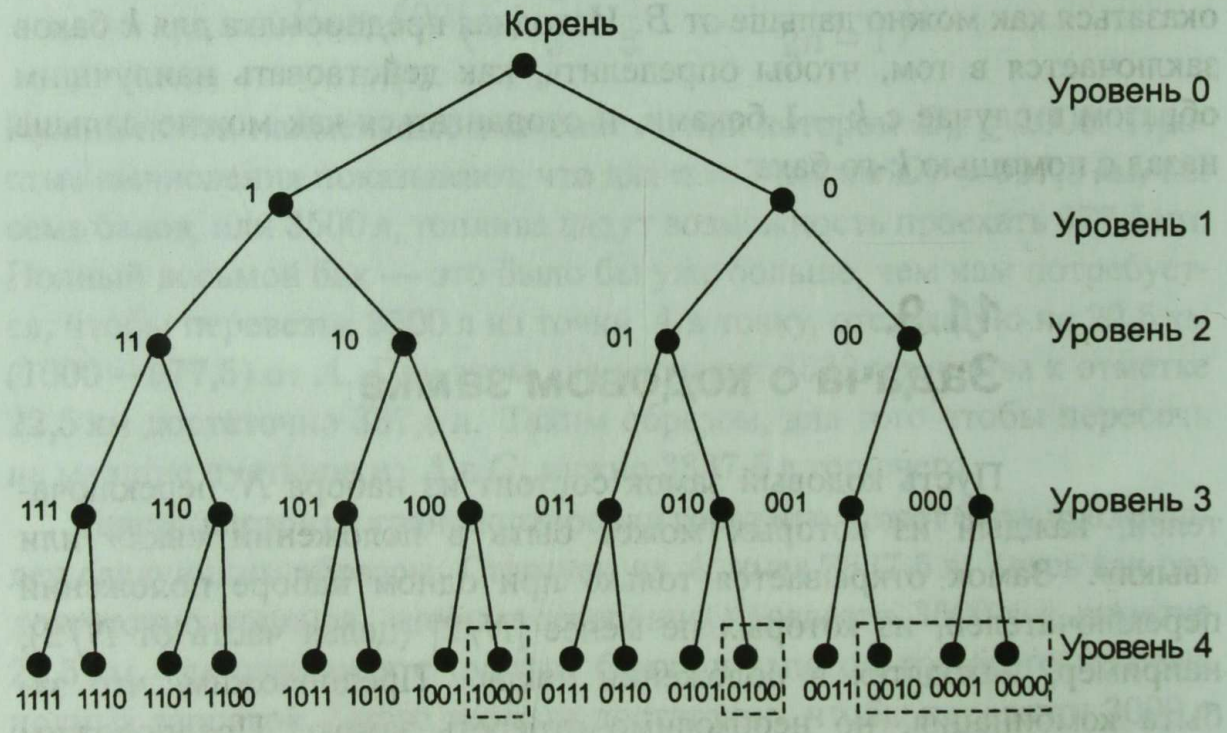


Рис. 11.25. Бинарное дерево

Условие, заключающееся в том, что число переключателей в положении «вкл» должно быть не меньше $[N/2]$, позволяет не рассма-

тривать те части дерева, которые не могут привести к правильной комбинации. Например, рассмотрим вершину 00 на рис. 11.25. Так как правая ветвь 000 не может привести к допустимой комбинации, нет необходимости ее формировать. Если какие-то вершины, следующие за рассматриваемой вершиной, не удовлетворяют ограничению задачи, то эти вершины не надо рассматривать. В данном случае вершины, находящиеся внутри пунктирных линий, не нужно исследовать и даже формировать.

Теперь, воспользовавшись этой моделью двоичного дерева, можно изложить процедуру отхода назад для образования только тех комбинаций, в которых по крайней мере $[N/2]$ переключателей находятся в положении «вкл». Алгоритм сводится к просмотру дерева в определенном порядке. Движемся вниз по дереву от корня, придерживаясь левой ветви, до тех пор, пока это возможно. Достигнув конечной вершины, анализируем соответствующую комбинацию. Если она не подходит, поднимаемся на один уровень и проверяем, можно ли спуститься вниз по другой ветви.

Алгоритм заканчивает работу, когда не остается не просмотренных ветвей.

Этот пример иллюстрирует основные свойства, общие для всех алгоритмов с отходом назад. Если можно сформулировать задачу так, что все возможные решения могут быть образованы построением N-векторов, то ее можно решить при помощи процедуры с отходом.

Контрольные вопросы

1. Какова теоретическая сложность алгоритмов, рассмотренных в данной работе?
2. Назовите особенности работы волнового и лучевых алгоритмов?
3. Назовите особенности работы маршрутного алгоритма?
4. По какой формуле осуществляется вычисление расстояния между двумя точками?
5. Как влияет выбор приоритетов на длину трассы?
6. В чем заключаются принципы составления оптимального расписания работы параллельных процессоров?
7. Укажите основные особенности задачи упаковки?
8. Приведите принципы решения задачи о джипе?
9. Как построить дерево решений в задаче о кодовом замке?

Глава 12
Метод ветвей и границ. Задача коммивояжера

Пусть $M = \{m_1, \ldots, m_r\}$ — конечное множество и $f\colon M \to \Re$ — вещественнозначная функция на нем; требуется найти минимум этой функции и элемент множества, на котором этот минимум достигается.

Когда имеется та или иная дополнительная информация о множестве, решение этой задачи иногда удается осуществить без полного перебора элементов всего множества M. Но чаще всего полный перебор производить приходится. В этом случае обязательно возникает задача, как лучше организовать перебор.

Метод ветвей и границ применим в том случае, когда выполняются специфические дополнительные условия на множество M и минимизируемую на нем функцию, а именно, предположим, что имеется вещественнозначная функция φ на множестве подмножеств множества M со следующими двумя свойствами:

1) для любого i $\varphi(\{m_i\}) = f(m_i)$ (здесь $\{m_i\}$ — множество, состоящее из единственного элемента m_i);

2) если $U \subseteq V \subseteq M$, то $\varphi(U) \geq \varphi(V)$.

В этих условиях можно организовать перебор элементов множества M с целью минимизации функции на этом множестве так: разобьем множество M на части (любым способом) и выберем ту из его частей Ω_1, на которой функция φ минимальна; затем разобьем на несколько частей множество Ω_1 и выберем ту из его частей Ω_2, на которой функция φ минимальна; затем разобьем Ω_2 на несколько частей и выберем ту из них, где минимальна φ, и т.д., пока не придем к какому-либо одноэлементному множеству $\{m_i\}$.

Это одноэлементное множество $\{m_i\}$ называется ***рекордом***.

Функция φ, которая используется при этом выборе, называется *оценочной*. Очевидно, что рекорд не обязан доставлять минимум функции f; однако возникает возможность сократить перебор при благоприятных обстоятельствах.

Описанный выше процесс построения рекорда состоял из последовательных этапов, на каждом из которых фиксировалось несколько множеств и выбиралось затем одно из них. Пусть $A_1, \ldots, A_s$ —

подмножества множества M, возникшие на предпоследнем этапе построения рекорда, и пусть множество A_1 оказалось выбранным с помощью оценочной функции. Именно при разбиении A_1 и возник рекорд, который для определенности обозначим через $\{m_1\}$. Согласно сказанному выше, $\varphi(A_1) \leq \varphi(A_i)$, $i = 1, \ldots, s$; кроме того, по определению оценочной функции, $\varphi(A_1) \leq \varphi(\{m_1\}) = f(m_1)$.

Предположим, что $f(m_1) \leq \varphi(A_2)$; тогда для любого элемента m множества M, принадлежащего множеству A_2, будут верны неравенства $f(m_1) \leq \varphi(A_2) \leq f(m)$; это значит, что при полном переборе элементов из M элементы из A_2 уже вообще не надо рассматривать. Если же неравенство $f(m_1) \leq \varphi(A_2)$ не будет выполнено, то все элементы из A_2 надо последовательно сравнить с найденным рекордом и как только отыщется элемент, дающий меньшее значение оптимизируемой функции, надо им заменить рекорд и продолжить перебор. Последнее действие называется *улучшением рекорда*.

Метод ветвей и границ связан с естественной графической интерпретацией всего изложенного: строится многоуровневое дерево, на нижнем уровне которого располагаются элементы множества M, на котором ветви ведут к рекорду и его улучшениям и на котором часть ветвей остается «отсеченной», потому что их развитие оказалось нецелесообразным.

Идея метода ветвей и границ заключается в создании некоторой процедуры построения ограниченного дерева перебора. Правда, в каких-то особых случаях «ограниченный» перебор может совпасть с полным: в этом проявляется эвристичность метода ветвей и границ.

Пусть имеется конечное множество M вариантов решения и функция F, принимающая различные значения от выбранного варианта. Требуется среди множества вариантов найти оптимальный, т.е. такой, на котором функция F принимает максимальное или минимальное (в зависимости от требований задачи) значения. Метод ветвей и границ отыскания оптимального варианта состоит из двух основных этапов:

- ветвления, т.е. построения дерева перебора;
- отсечения ветвей, т.е. прекращения построения дерева в тех ветвях, которые заведомо не содержат оптимального решения.

Рассмотрим основные принципы метода ветвей и границ на примере решения задачи расшифровки криптограмм и о радиоактивном шаре.

12.1. Расшифровка криптограмм

Пусть дана криптограмма

$$BEST + MADE = MASER.$$

Здесь каждая буква шифрует какую-то одну цифру из множества $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Требуется расшифровать данную криптограмму. Если воспользоваться деревом полного перебора, то можно показать, что восемь букв и десять цифр могут быть сопоставлены друг с другом различными способами, число которых достигает значения 1814400. Однако если воспользоваться определенными свойствами десятичной системы счисления, правилами сложения десятичных чисел, а также операциями ветвления и отсечения ветвей, то пространство поиска в данной задаче можно качественным образом сузить.

Для удобства рассуждений введем следующие обозначения: столбцы зашифрованных цифр пронумеруем слева направо от 1 до 5. Символы P_1, P_2, P_3, P_4 будут обозначать соответствующую цифру переноса из одного разряда в другой. Если цифра i-го разряда больше или равна 10, то $P_i = 1$, в противном случае $P_i = 0$ (рис. 12.1). Очевидно, что при сложении двух десятичных цифр величина переноса не может превышать одной единицы следующего разряда.

P_4	P_3	P_2	P_1		Переносы
	B	E	S	T	
	M	A	D	E	
M	A	S	E	R	
5	4	3	2	1	Номера разрядов

Рис. 12.1. Задача о криптограмме

Начнем анализ с пятого разряда. Можно уверенно утверждать, что $M = 1$, так как M не может равняться нулю, поскольку не

принято писать нуль в начальной позиции, также очевидно, что M не может быть больше единицы. Так как в пятом разряде M целиком образована значением переноса P_4, то получаем $M = 1$. Определен корень ограниченного дерева поиска (рис. 12.2, *а*).

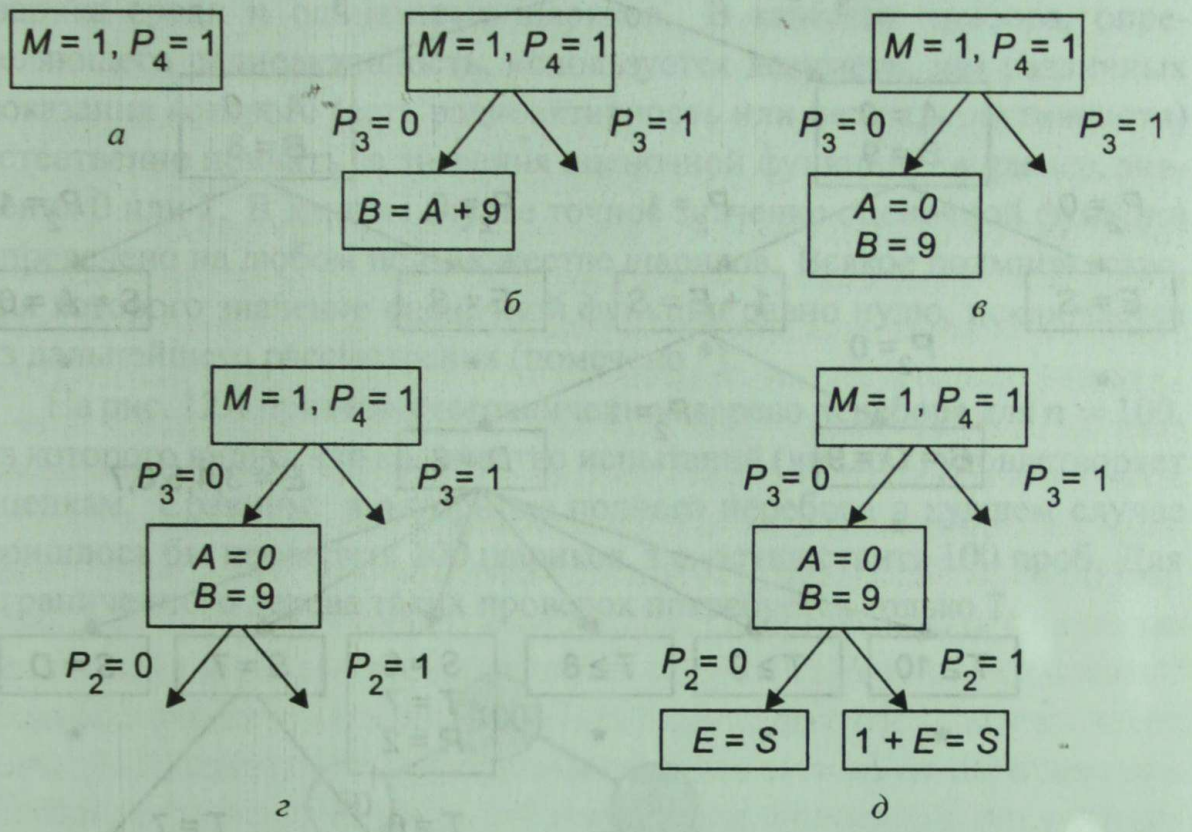


Рис. 12.2. Этапы построения дерева перебора для расшифровки криптограммы

Обращаясь к четвертому разряду, нельзя с уверенностью сказать, чему равно P_3 — единице или нулю. Отметим оба варианта на дереве поиска (рис. 12.2, *б*). Левая ветвь: $P_3 = 0$, так как $P_4 = 1$ и $P_3 = 0$, то $B + M$ должно быть равно $A + 10$. Но из-за того, что $M = 1$, получается, что $B = A + 9$. Но B не может быть больше 9, поэтому $A = 0$ (рис. 12.2, *в*). Теперь проанализируем фрагмент дерева, изображенный на рис. 12.2, *г*.

Если рассматривать ветвь $P_2 = 0$, то нетрудно видеть, что $P_2 + E = S$. Поскольку $P_2 = 0$, $A = 0$, то имеем $E = S$, что запрещено правилами построения криптограммы. Следовательно, путь $P_3 = 0$, $P_2 = 0$ тупиковый, его дальше продолжать не следует. Если же двигаться по ветви $P_2 = 1$, получаем $1 + E = S$, и этот путь, возможно, следует просматривать далее (рис. 12.2, *д*). Продолжая аналогичное построение, получим ограниченное дерево поиска для данной крип-

тографической задачи (рис. 12.3). Звездочками обозначены ветви, не имеющие продолжения.

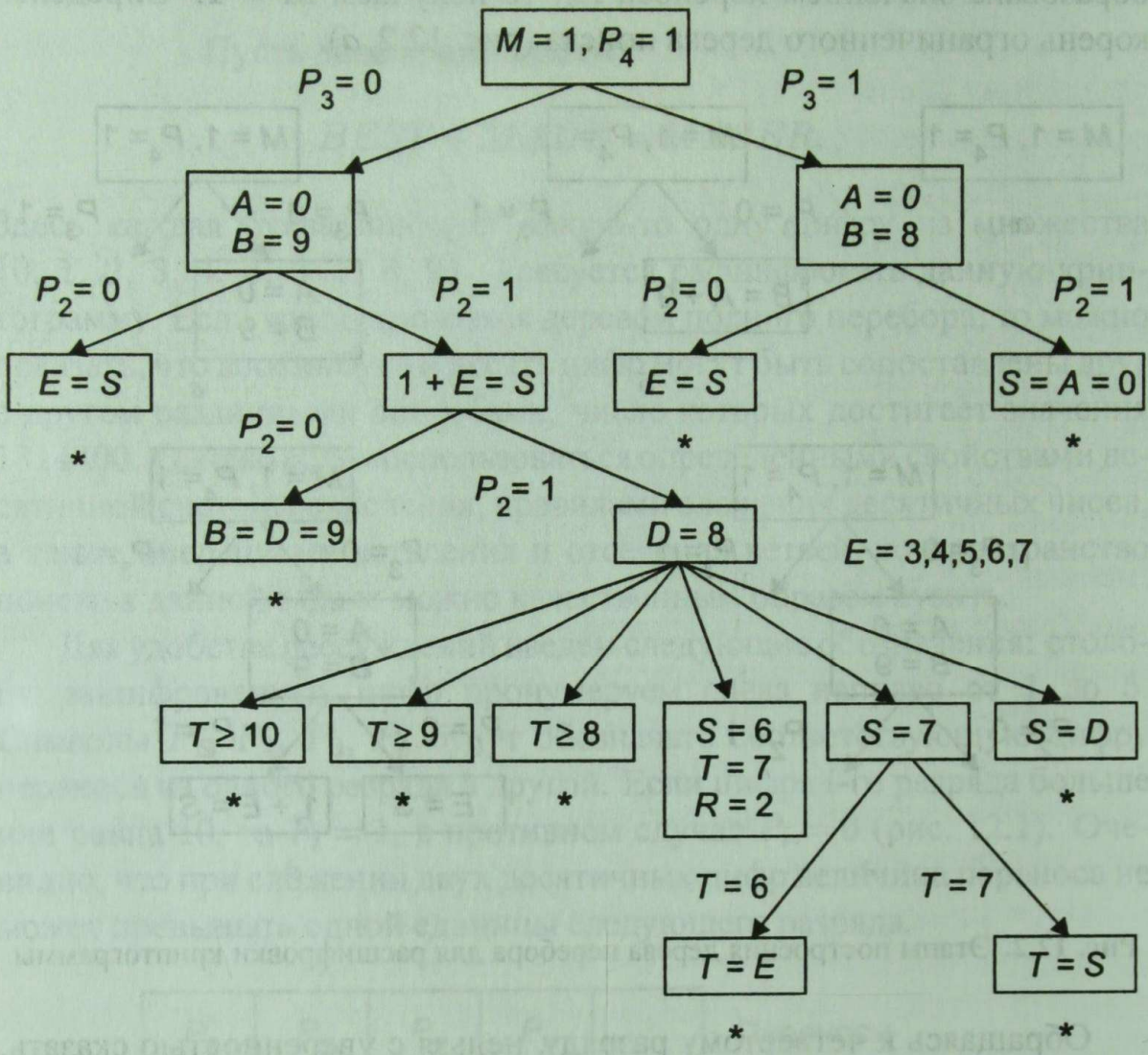


Рис. 12.3. Ограниченное дерево перебора для расшифровки криптограммы

Следует отметить, что сначала в качестве принципа ветвления использовался признак наличия или отсутствия переноса, а затем, когда этот признак был исчерпан, пришлось сделать полный перебор по всем возможным значениям буквы E (перечислены на рисунке слева направо). Таким образом, вместо полного дерева перебора, содержащего около 1814400 листьев, удается построить ограниченное дерево, рассмотрение которого позволяет легко найти нужное решение. В результате можно сделать следующий вывод. Многие комбинаторные задачи существенно упрощаются, если вместо комбинаторно полного дерева перебора удается построить ограниченное дерево, обязательно включающее пространство, содержащее решение.

12.2. Задача о радиоактивном шаре

Рассмотрим задачу об отыскании одного радиоактивного шарика среди n одинаковых шариков. В качестве прибора, определяющего радиоактивность, используется дозиметр, два различных показания которого (есть радиоактивность или нет радиоактивности) естественно принять за значения оценочной функции, например, значения 0 или 1. В данном случае точное значение оценочной функции определено на любом подмножестве шариков. Всякое подмножество, для которого значение оценочной функции равно нулю, исключается из дальнейшего рассмотрения (помечено *).

На рис. 12.4 приведено ограниченное дерево перебора для $n = 100$, из которого видно, что количество испытаний (число 7) удовлетворяет оценкам. Сравним: в алгоритме полного перебора в худшем случае пришлось бы проверить 100 шариков, т.е. осуществить 100 проб. Для ограниченного дерева таких проверок потребуется только 7.

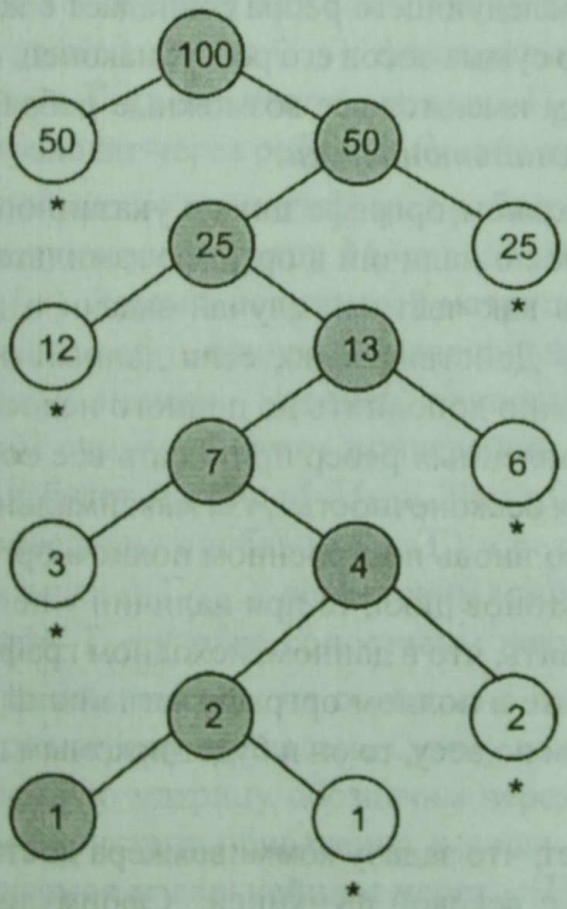


Рис. 12.4. Ограниченное дерево перебора

12.3. Задача коммивояжера

Имеется несколько городов, соединенных дорогами с известной длиной; требуется установить, имеется ли путь, двигаясь по которому можно побывать в каждом городе только один раз и при этом вернуться в город, откуда путь был начат («обход коммивояжера»), и, если такой путь имеется, установить кратчайший из них.

Формализуем условие в терминах теории графов. Города будут вершинами графа, а дороги между городами — ориентированными (направленными) ребрами графа, на каждом из которых задана весовая функция: вес ребра — это длина соответствующей дороги. Путь, который требуется найти, — это ориентированный остовный простой цикл минимального веса в ориентированном графе (орграфе). (Напомним: цикл называется *остовным*, если он проходит по всем вершинам графа; цикл называется *простым*, если он проходит по каждой своей вершине только один раз; цикл называется *ориентированным*, если начало каждого последующего ребра совпадает с концом предыдущего; *вес* цикла — это сумма весов его ребер; наконец, орграф называется *полным*, если в нем имеются все возможные ребра); такие циклы называются также ***гамильтоновыми***.

Очевидно, в полном орграфе циклы указанного выше типа есть. Заметим, что вопрос о наличии в орграфе гамильтонова цикла достаточно рассмотреть как частный случай задачи о коммивояжере для полных орграфов. Действительно, если данный орграф не является полным, то его можно дополнить до полного недостающими ребрами и каждому из добавленных ребер приписать вес ∞, считая, что ∞ — это «компьютерная бесконечность», т.е. максимальное из всех возможных чисел. Если во вновь построенном полном орграфе найти теперь легчайший гамильтонов цикл, то при наличии у него ребер с весом ∞ можно будет говорить, что в данном, исходном графе «цикла коммивояжера» нет. Если же в полном орграфе легчайший гамильтонов цикл окажется конечным по весу, то он и будет искомым циклом в исходном графе.

Отсюда следует, что задачу коммивояжера достаточно решить для полных орграфов с весовой функцией. Сформулируем теперь это в окончательном виде: пусть $G = (A, B)$ — полный ориентированный

граф и $v\colon B \to \mathfrak{R}$ — весовая функция; найти простой остовный ориентированный цикл («цикл коммивояжера») минимального веса.

Пусть $A = \{a_1, \ldots, a_p\}$ — конкретный состав множества вершин и $M = (m_{ij})$, $i, j = 1, \ldots, p$ — весовая матрица данного орграфа, т.е. $m_{ij} = v(a_i, a_j)$, причем для любого i $m_{ii} = \infty$.

Рассмотрим использование метода ветвей и границ для решения задачи коммивояжера.

Шаг 1. Фиксируем множество всех обходов коммивояжера (т.е. всех простых ориентированных остовных циклов). Поскольку граф — полный, это множество заведомо непустое. Сопоставим ему число, которое будет играть роль значения на этом множестве оценочной функции: это число равно сумме констант приведения данной матрицы весов ребер графа. Если множество всех обходов коммивояжера обозначить через Γ, то сумму констант приведения матрицы весов обозначим через $\varphi(\Gamma)$. Приведенную матрицу весов данного графа следует запомнить; обозначим ее через M_1; таким образом, итог первого шага: множеству Γ всех обходов коммивояжера сопоставлено число $\varphi(\Gamma)$ и матрица M_1.

Шаг 2. Выберем в матрице M_1 самый тяжелый ноль (функцию штрафа); пусть он стоит в клетке (i, j); фиксируем ребро графа (i, j) и разделим множество Γ на две части: на часть $\Gamma_{(i,j)}$, состоящую из обходов, которые проходят через ребро (i, j), и на часть $\Gamma_{\overline{(i,j)}}$, состоящую из обходов, которые не проходят через ребро (i, j). Сопоставим множеству $\Gamma_{(i,j)}$ следующую матрицу $M_{1,1}$: в матрице M_1 заменим на ∞ число в клетке (j, i). Затем в полученной матрице вычеркнем строку номер i и столбец номер j, причем у оставшихся строк и столбцов сохраним их исходные номера. Наконец, приведем эту последнюю матрицу и запомним сумму констант приведения. Полученная приведенная матрица и будет матрицей $M_{1,1}$. Только что запомненную сумму констант приведения прибавим к $\varphi(\Gamma)$ и результат, обозначаемый в дальнейшем через $\varphi(\Gamma_{(i,j)})$, сопоставим множеству $\Gamma_{(i,j)}$.

Теперь множеству $\Gamma_{\overline{(i,j)}}$ тоже сопоставим некую матрицу $M_{1,2}$. Для этого в матрице M_1 заменим на ∞ число в клетке (i, j) и полученную в результате матрицу приведем. Сумму констант приведения запомним, а полученную матрицу обозначим через $M_{1,2}$. Прибавим запомненную сумму констант приведения к числу $\varphi(\Gamma)$, и полученное число, обозначаемое в дальнейшем через $\varphi(\Gamma_{\overline{(i,j)}})$, сопоставим множеству $\Gamma_{\overline{(i,j)}}$.

Выберем между множествами $\Gamma_{(i,j)}$ и $\Gamma_{\overline{(i,j)}}$ то множество, на котором минимальна функция φ (т.е. то из множеств, которому соответствует меньшее из чисел $\varphi(\Gamma_{(i,j)})$ и $\varphi(\Gamma_{\overline{(i,j)}})$.

Заметим теперь, что в проведенных рассуждениях использовался в качестве исходного только один фактический объект — приведенная матрица весов данного орграфа. По ней было выделено определенное ребро графа и были построены новые матрицы, к которым, конечно, можно все то же самое применить.

При каждом таком повторном применении будет фиксироваться очередное ребро графа. Условимся о следующем действии: перед тем, как в очередной матрице вычеркнуть строку и столбец, в ней надо заменить на ∞ числа во всех тех клетках, которые соответствуют ребрам, заведомо не принадлежащим тем гамильтоновым циклам, которые проходят через уже отобранные ранее ребра.

К выбранному множеству с сопоставленными ему матрицей и числом φ применим до тех пор, пока это возможно, указанный выше алгоритм.

Доказывается, что в результате получится множество, состоящее из единственного обхода коммивояжера, вес которого равен очередному значению функции φ; таким образом, оказываются выполненными все условия, обсуждавшиеся при описании метода ветвей и границ.

После этого осуществляется улучшение рекорда вплоть до получения окончательного ответа.

Жадный алгоритм. Найти приближенное значение кратчайшего тура ($TOUR$) со стоимостью $COST$ для задачи коммивояжера с N городами и матрицей стоимости C, начиная с вершины U.

Жадный алгоритм — алгоритм нахождения кратчайшего расстояния путем выбора самого короткого, еще не выбранного ребра, при условии, что оно не образует цикла с уже выбранными ребрами. Алгоритм использует стратегию «иди в ближайший город». «Жадным» этот алгоритм назван потому, что на последних шагах приходится жестоко расплачиваться за жадность.

Шаг 1 (Инициализация). $TOUR := 0$; $COST := 0$. Пометить вершину U как «выбранную», т.е. $V := U$, где V — текущая переменная, а все другие вершины — как «не выбранные».

Шаг 2 (Посещение всех городов). FOR $i := 1$ TO $N - 1$ DO.

Ш а г 3 (Выбор следующего ребра). Пусть (V,W) — ребро с наименьшей стоимостью, ведущее из V в любую «невыбранную» вершину W. $TOUR := TOUR + (V,W)$; $COST := COST + C(V,W)$. Помечаем W как «выбранную», т.е. $V := W$.

Ш а г 4 (Завершение тура). $TOUR := TOUR + (V,U)$; $COST := COST + C(V,U)$.

На рис. 12.5 проиллюстрирована последовательность выполнения данного алгоритма для графа, имеющего пять вершин (тур начинается с вершины 1). Пройденные вершины обозначены черным квадратиком, а не пройденные — кружком. Жадный алгоритм для данного графа нашел тур со стоимостью 14, хотя оптимальный тур имеет стоимость 13. Ясно, что «грубый алгоритм» применительно к задаче коммивояжера не всегда находит тур с минимальной стоимостью.

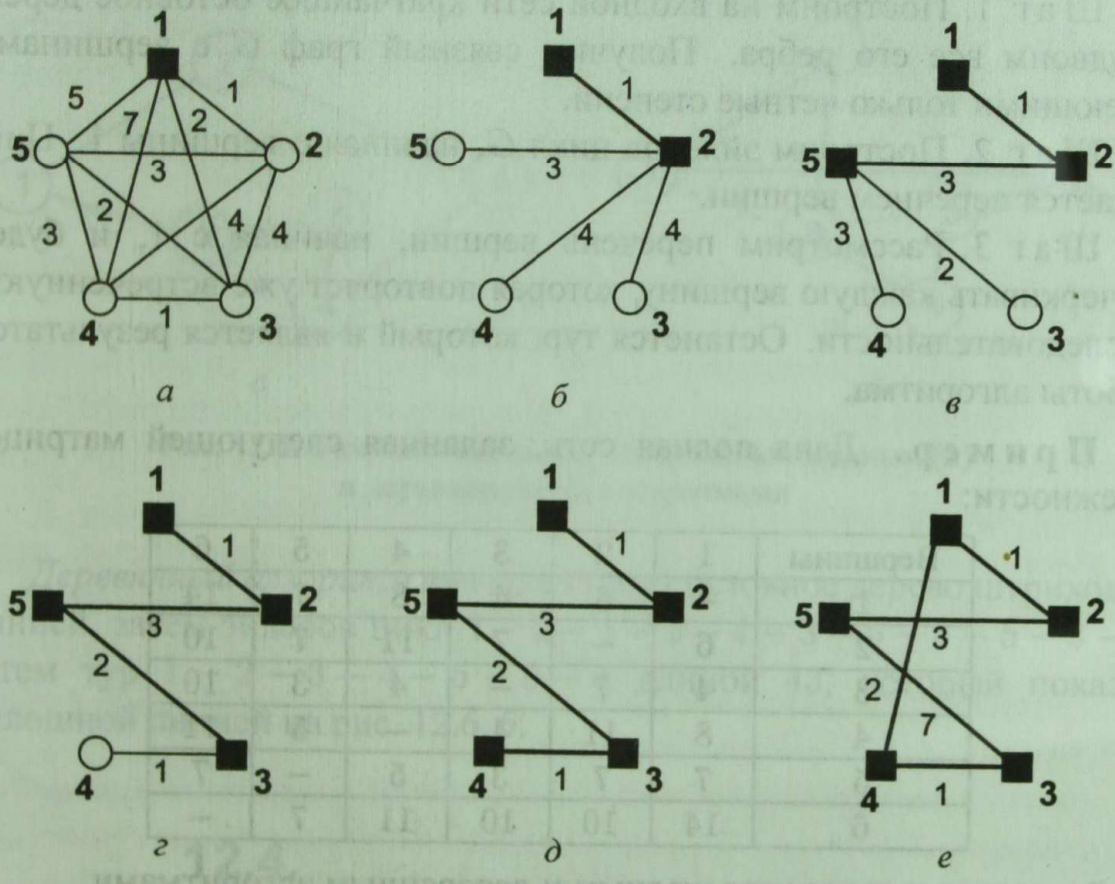


Рис. 12.5. Иллюстрация работы жадного алгоритма:
а — стоимость = 0; *б* — стоимость = 1; *в* — стоимость = 4;
г — стоимость = 6; *д* — стоимость = 7; *е* — стоимость = 14

Жадный алгоритм основан на идее подъема. Цель — найти тур с минимальной стоимостью. Задача сведена к набору частных целей — найти на каждом шаге «самый дешевый» город, чтобы посетить

его следующим. Алгоритм не строит плана вперед; текущий выбор делается безотносительно к следующим выборам.

Для произвольной задачи коммивояжера с n городами требуется $O(n^2)$ операций, чтобы обработать матрицу стоимостей C. Поэтому нижняя граница сложности любого алгоритма, способного дать нетривиальное возможное решение этой задачи, равна $O(n^2)$. Нетрудно проверить, что для любой разумной реализации шагов 1–3 требуется не больше чем $O(n^2)$ операций.

Деревянный алгоритм. В основе деревянного алгоритма лежит построение остовного дерева, которое, с учетом эвристических подходов, преобразуется в тур коммивояжера.

Рассмотрим этапы решения задачи коммивояжера деревянным алгоритмом.

Шаг 1. Построим на входной сети кратчайшее остовное дерево и удвоим все его ребра. Получим связный граф G с вершинами, имеющими только четные степени.

Шаг 2. Построим эйлеров цикл G, начиная с вершины 1. Цикл задается перечнем вершин.

Шаг 3. Рассмотрим перечень вершин, начиная с 1, и будем вычеркивать каждую вершину, которая повторяет уже встреченную в последовательности. Останется тур, который и является результатом работы алгоритма.

Пример. Дана полная сеть, заданная следующей матрицей смежности:

Вершины	1	2	3	4	5	6
1	–	6	4	8	7	14
2	6	–	7	11	7	10
3	4	7	–	4	3	10
4	8	11	4	–	5	11
5	7	7	3	5	–	7
6	14	10	10	11	7	–

Найти тур коммивояжера жадным и деревянным алгоритмами.

Решение. Если справедливо неравенство треугольника, то

$$d[1,3] \le d[1,2] + d[2,3] \quad \text{и} \quad d[3,5] \le d[3,4] + d[4,5].$$

Сложив эти два неравенства, получим

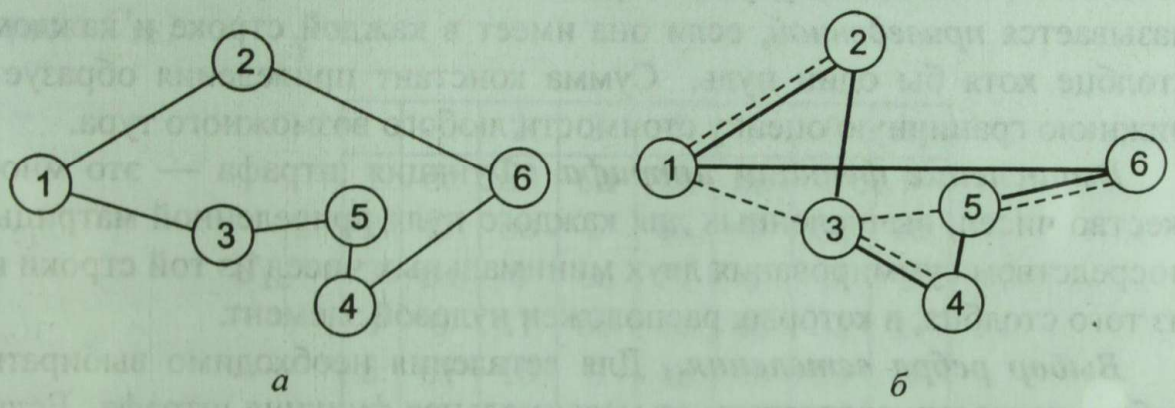
$$d[1,3] + d[3,5] \le d[1,2] + d[2,3] + d[3,4] + d[4,5].$$

По неравенству треугольника получим, $d[1,5] \le d[1,3] + d[3,5]$. Окончательно имеем

$$d[1,5] \le d[1,2] + d[2,3] + d[3,4] + d[4,5].$$

Итак, если справедливо неравенство треугольника, то для каждой цепи верно, что расстояние от начала до конца цепи меньше (или равно) суммарной длине всех ребер цепи. Это обобщение расхожего убеждения, что прямая короче кривой.

Жадный алгоритм (иди в ближайший город) дает тур 1 – (4) – 3 – (3) – 5 – (5) – 4 – (11) – 6 – (10) – 2 – (6) – 1, где без скобок показаны номера вершин, а в скобках — длины ребер. Длина тура равна 39, тур показан на рис. 12.6, *а*.



а *б*

Рис. 12.6. Гамильтонов цикл, построенный жадным (*а*) и деревянным (*б*) алгоритмами

Деревянный алгоритм вначале строит остовное дерево штриховой линией, затем эйлеров цикл 1 – 2 – 1 – 3 – 4 – 3 – 5 – 6 – 5 – 3 – 1, затем тур 1 – 2 – 3 – 4 – 5 – 6 – 1 длиной 43, который показан сплошной линией на рис. 12.6, *б*.

12.4. Примеры решения задачи коммивояжера

Пример 1. Решить методом ветвей и границ задачу коммивояжера для графа, содержащего 6 вершин. Пусть исходный ориентированный граф задан матрицей стоимости:

$C_1 =$

	1	2	3	4	5	6
1	∞	68	73	24	70	9
2	58	∞	16	44	11	92
3	63	9	∞	86	13	18
4	17	34	76	∞	52	70
5	60	18	3	45	∞	58
6	16	82	11	60	48	∞

Начальное приведение матрицы стоимости. Пусть имеется некоторая числовая матрица. *Привести строку* этой матрицы означает выделить в строке минимальный элемент (его называют *константой приведения*) и вычесть его из всех элементов этой строки. В результате в этой строке на месте минимального элемента окажется нуль, а все остальные элементы будут неотрицательными. Матрица стоимости называется *приведенной*, если она имеет в каждой строке и каждом столбце хотя бы один нуль. Сумма констант приведения образует нижнюю граничную оценку стоимости любого возможного тура.

Вычисление функции штрафа. Функция штрафа — это множество чисел, вычисленных для каждого нуля приведенной матрицы посредством суммирования двух минимальных чисел из той строки и из того столбца, в которых расположен нулевой элемент.

Выбор ребра ветвления. Для ветвления необходимо выбирать ребро, которому соответствует максимальная функция штрафа. Если существует несколько одинаковых максимальных значений функции штрафа, выбор среди них может быть произвольным.

Весом элемента матрицы называют сумму констант приведения матрицы, которая получается из данной матрицы заменой анализируемого элемента на ∞. Следовательно, выражение *самый тяжелый нуль* в матрице означает, что в матрице подсчитан вес каждого нуля, а затем фиксирован нуль с максимальным весом.

Вычисление граничной оценки для ветви, соответствующей не включению ребра в тур. Эта оценка вычисляется как сумма граничной оценки, соответствующей предыдущему узлу дерева перебора, и выбранному значению функции штрафа.

Вычисление граничной оценки для ветви, соответствующей включению ребра в тур. Для вычисления граничной оценки необходимо:

- вычеркнуть в матрице стоимости строку и столбец, соответствующие выбранному ребру;

• скорректировать полученную матрицу таким образом, чтобы устранить возможность досрочного завершения тура (устранить циклы);

• сделать приведение (если необходимо) полученной матрицы, и если константа приведения отлична от нуля, сложить эту константу с граничной оценкой предыдущего узла.

Проверка на окончание решения. Если скорректированная матрица имеет размер 2×2, а узел дерева, которому соответствует эта матрица, имеет минимальную граничную оценку, то решение задачи заканчивается: два оставшихся нуля этой матрицы соответствуют двум последним ребрам, которые включаются в тур непосредственно, при этом стоимость тура не изменяется.

Решение. Сначала приводим исходную матрицу по строкам (матрица C_{1c}):

$C_{1c} =$

	1	2	3	4	5	6	h_i
1	∞	59	64	15	61	0	9
2	47	∞	5	33	0	81	11
3	54	0	∞	77	4	9	9
4	0	17	59	∞	35	53	17
5	57	15	0	42	∞	55	3
6	5	71	0	49	37	∞	11

В последнем столбце этой матрицы h_i записаны константы приведения по строкам.

Полученную матрицу необходимо привести по столбцам. В результате получаем матрицу C_1', в которой в строке h_j записаны константы приведения по столбцам:

$C_1' =$

	1	2	3	4	5	6
1	∞	59	64	0	61	0
2	47	∞	5	18	0	81
3	54	0	∞	62	4	9
4	0	17	59	∞	35	53
5	57	15	0	27	∞	55
6	5	71	0	34	37	∞
h_j	0	0	0	15	0	0

Сумма констант приведения H дает нижнюю граничную оценку стоимости всех туров:

$$H = \sum_i h_i + \sum_j h_j = 9 + 11 + 9 + 17 + 3 + 11 + 15 = 75.$$

Вычисляем для ребер, помеченных нулем в матрице C_1', значения функций штрафа, которые обозначим символом D_{ij}:

$$D_{16} = 0 + 9 = 9; \qquad D_{41} = 17 + 5 = 22;$$
$$D_{25} = 5 + 4 = 9; \qquad D_{53} = 15 + 0 = 15;$$
$$D_{32} = 4 + 15 = 19; \qquad D_{63} = 5 + 0 = 5.$$

Максимальное значение функции штрафа равно 22, на основании чего в качестве ребра ветвления выбираем ребро $(4, 1)$. Разбиваем множество всех туров R на два подмножества: $\{4, 1\}$ — подмножество всех туров, в которое входит ребро $(4, 1)$, и подмножество всех туров $\{4, 1^*\}$, в которое ребро $(4, 1)$ не входит. Строим первый фрагмент ограниченного дерева перебора (рис. 12.7). При этом граничная оценка в ветви $\{4, 1^*\}$ находится непосредственно как сумма $75 + 22 = 97$.

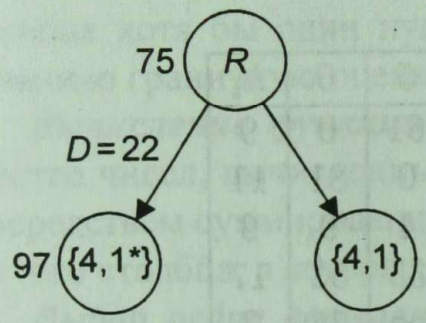


Рис. 12.7. Первый фрагмент ограниченного дерева перебора

Вычеркиваем в матрице C_1' четвертую строку и первый столбец, а чтобы запретить досрочное завершение тура по ребру графа $(1, 4)$, весу этого ребра присвоить значение ∞. В результате подобной корректировки получаем матрицу

$C_2 =$

	2	3	4	5	6
1	59	64	∞	61	0
2	∞	5	18	0	81
3	0	∞	62	4	9
5	15	0	27	∞	55
6	71	0	34	37	∞
h_j	0	0	18	0	0

Ее необходимо привести по четвертому столбцу (константа приведения $h_4 = 18$), в результате чего получаем приведенную матрицу C_2'

и вычисляем граничную оценку в ветви $\{4,\ 1\}$, которая будет равна $75 + 18 = 93$ (рис. 12.8).

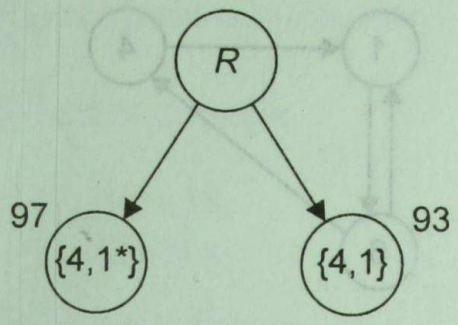


Рис. 12.8. Первый фрагмент ограниченного дерева перебора с граничными оценками в листьях

Сравниваем оценки в листьях и приходим к выводу, что ветвление целесообразно продолжать в листе с оценкой 93. Так как этому листу соответствует матрица C_2', то с ней необходимо поступать как с исходной матрицей, т.е. вычислять для нее значения функций штрафа, выбирать максимальное значение штрафа и на его основе выбирать следующее ребро ветвления:

$C_2' =$

	2	3	4	5	6
1	59	64	∞	61	0
2	∞	5	0	0	81
3	0	∞	44	4	9
5	15	0	9	∞	55
6	71	0	16	37	∞

Для матрицы C_2' имеем следующие значения функций штрафа:

$$D_{16} = 59 + 9 = 68; \qquad D_{32} = 4 + 15 = 19;$$
$$D_{24} = 0 + 9 = 9; \qquad D_{53} = 9 + 0 = 9;$$
$$D_{25} = 0 + 4 = 4; \qquad D_{32} = 16 + 0 = 16.$$

Выбираем максимальное значение 68, которому соответствует ребро ветвления $(1, 6)$, после чего можно построить второй фрагмент ограниченного дерева перебора (рис. 12.9).

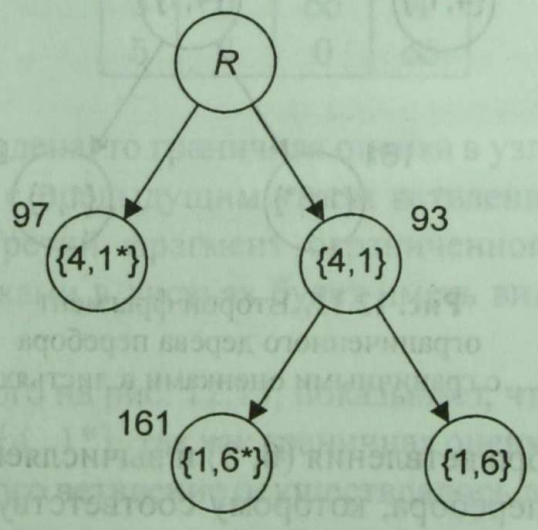


Рис. 12.9. Второй фрагмент ограниченного дерева перебора

Затем приступаем к вычислению граничной оценки в узле дерева $\{1,\ 6\}$. Для этого вычеркиваем в матрице C_2' первую строку и шестой столбец, а также предпринимаем меры для исключения досрочного завершения тура матрицы C_3.

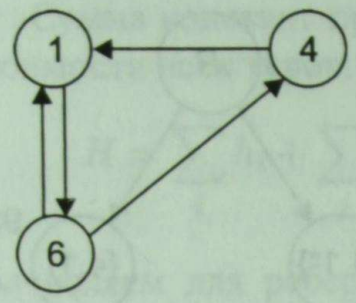


Рис. 12.10. Исключение досрочного завершения тура

В самом деле, к данному моменту в тур включено ребро $(4, 1)$ и есть предложение включить в него ребро $(1, 6)$. Из рис. 12.10 видно, что досрочно завершить тур могли бы ребра $(6, 1)$ и $(6, 4)$. Однако элемент $(6,1)$ в матрице C_3 уже отсутствует, следовательно, эту «роль» может выполнить лишь элемент $(6,4)$, поэтому заменяем его символом ∞. После этого окончательно матрица C_3 будет иметь вид:

$$C_3 = C_3' =$$

	2	3	4	5
2	∞	5	0	0
3	0	∞	44	4
5	15	0	9	∞
6	71	0	∞	37

Матрица C_3 оказалась приведенной (поэтому ее можно, как это было ранее принято для приведенных матриц, пометить штрихом), а следовательно, оценка в узле $(1,6)$ остается неизменной (рис. 12.11).

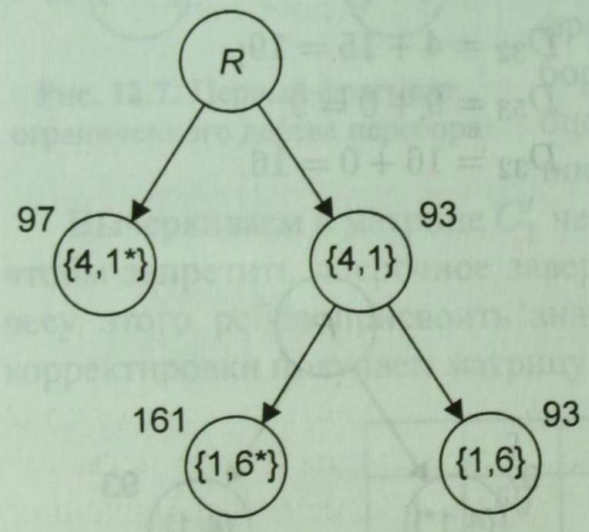


Рис. 12.11. Второй фрагмент ограниченного дерева перебора с граничными оценками в листьях

Вычисляем значения функций штрафа для матрицы C_3':

$$D_{24} = 0 + 9 = 9;$$

$$D_{25} = 0 + 4 = 4;$$

$$D_{32} = 4 + 15 = 19;$$

$$D_{53} = 9 + 0 = 9;$$

$$D_{63} = 37 + 0 = 37.$$

На основе вычисленных значений определяем следующее ребро ветвления $(6, 3)$ и вычисляем граничную оценку в том узле дерева перебора, которому соответствуют туры, не содержащие ребра $(6, 3)$ (рис. 12.12). При этом ветвление продолжается от узла с минимальной оценкой, равной 93.

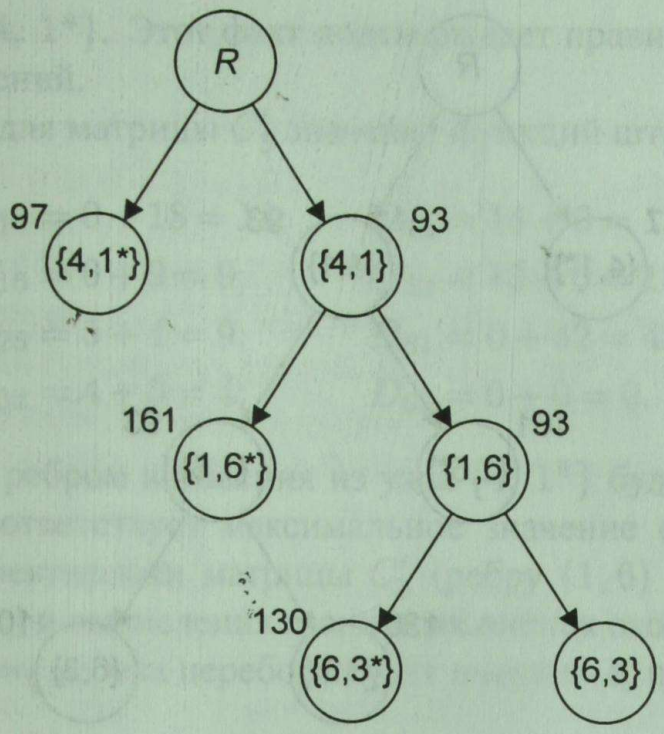


Рис. 12.12. Третий фрагмент ограниченного дерева перебора

Вычеркиваем в матрице C'_3 шестую строку и третий столбец и корректируем ее на исключение досрочного завершения тура. Это достигается присваиванием символа бесконечности элементу $(3,4)$. После этого получаем матрицу C_4, а после ее приведения — матрицу C'_4:

$$C_4 = \begin{array}{|c|c|c|c|} \hline & 2 & 4 & 5 \\ \hline 2 & \infty & 0 & 0 \\ 3 & 0 & \infty & 4 \\ 5 & 15 & 9 & \infty \\ \hline \end{array}, \qquad C'_4 = \begin{array}{|c|c|c|c|} \hline & 2 & 4 & 5 \\ \hline 2 & \infty & 0 & 0 \\ 3 & 0 & \infty & 4 \\ 5 & 6 & 0 & \infty \\ \hline \end{array}.$$

Так как матрица C_4 не была приведена, то граничная оценка в узле $\{6,\ 3\}$ увеличивается по сравнению с предыдущим узлом ветвления на 9 и становится равной 102. Третий фрагмент ограниченного дерева перебора с граничными оценками в листьях будет иметь вид, изображенный на рис. 12.13.

Рассмотрение дерева, приведенного на рис. 12.13, показывает, что ветвление нужно продолжать из узла $\{4,\ 1^*\}$, так как граничная оценка там меньше (97), чем в узле, из которого ветвление осуществлялось до сих пор. Подобная ситуация называется *перескоком*, и в этом случае может оказаться, что все полученные до сих пор данные будут утеряны.

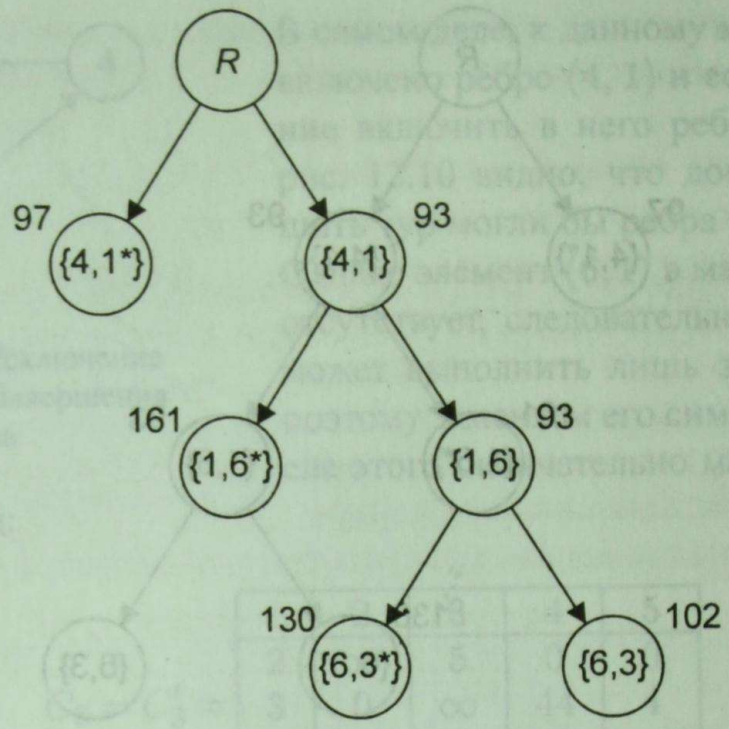


Рис. 12.13. Третий фрагмент ограниченного дерева перебора с граничными оценками в листьях

Итак, переходим к узлу $\{4, 1^*\}$, которому соответствует матрица

$C_5 =$

	1	2	3	4	5	6
1	∞	68	73	24	70	9
2	58	∞	16	44	11	92
3	63	9	∞	86	13	18
4	∞	34	76	∞	52	70
5	60	18	3	45	∞	58
6	16	82	11	60	48	∞

Матрица C_5 не приведена, и после выполнения операции приведения получаем матрицу C_5':

$C_5' =$

	1	2	3	4	5	6
1	∞	59	64	0	61	0
2	42	∞	5	18	0	81
3	49	0	∞	62	4	9
4	∞	0	42	∞	18	36
5	52	15	0	27	∞	55
6	0	71	0	34	37	∞

У матрицы C_5 сумма констант приведения будет равна 97, что соответствует вычисленной ранее через функцию штрафа граничной

оценке в узле $\{4,\ 1^*\}$. Этот факт подтверждает правильность выполненных вычислений.

Вычисляем для матрицы C'_5 значения функций штрафа:

$$D_{14} = 0 + 18 = 18; \qquad D_{42} = 18 + 0 = 18;$$
$$D_{16} = 0 + 9 = 9; \qquad D_{33} = 15 + 0 = 15;$$
$$D_{25} = 5 + 4 = 9; \qquad D_{61} = 0 + 42 = 42;$$
$$D_{32} = 4 + 0 = 4; \qquad D_{63} = 0 + 0 = 0.$$

Очередным ребром ветвления из узла $\{4,\ 1^*\}$ будет ребро $(6, 1)$, так как ему соответствует максимальное значение функции штрафа. После корректировки матрицы C'_5 (ребру $(1, 6)$ приписывается бесконечный вес) и вычисления граничных оценок окончательный четвертый фрагмент дерева перебора будет иметь вид, представленный на рис. 12.14.

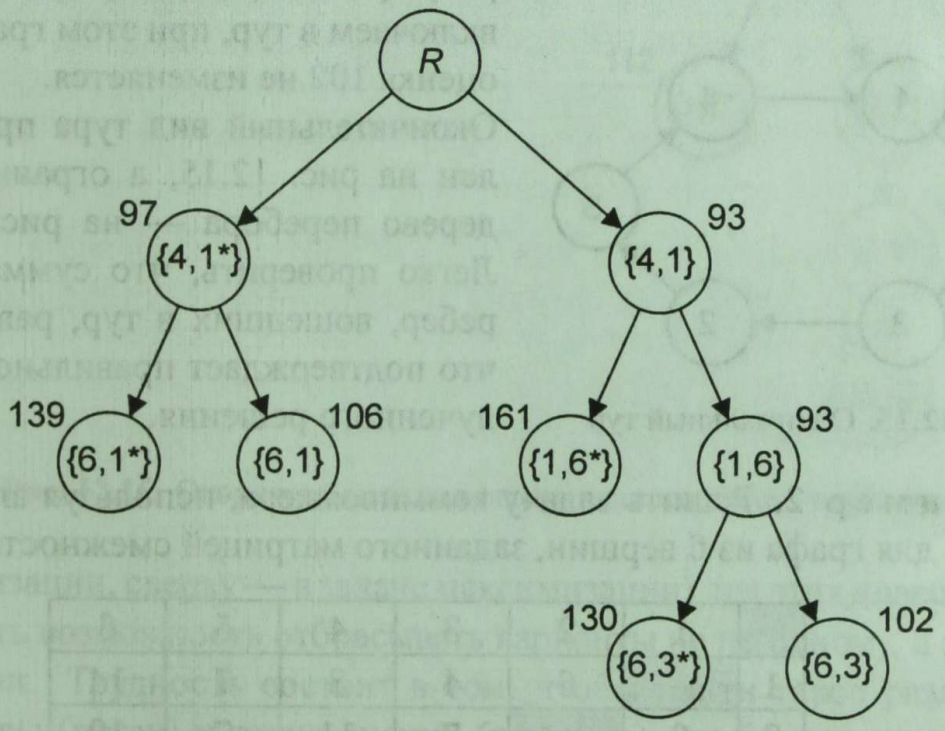


Рис. 12.14. Четвертый фрагмент ограниченного дерева перебора с граничными оценками в листьях

Из этого рисунка видно, что минимальная граничная оценка (102) имеет место в узле $\{6,\ 3\}$, поэтому целесообразно осуществить обратный «перескок» в этот узел и продолжать дальнейшее ветвление из него. Этому узлу соответствует ранее вычисленная матрица C'_4 ; дадим ей для удобства сквозной нумерации новое обозначение $C_6 = C'_6$.

Вычислим для матрицы C_6' значения функций штрафа:

$$D_{24} = 0 + 0 = 0; \qquad D_{32} = 4 + 6 = 10;$$
$$D_{25} = 0 + 4 = 4; \qquad D_{54} = 6 + 0 = 6.$$

Максимальное значение функции штрафа, равное 10, связано с ребром $(3, 2)$, поэтому выбираем его в качестве ребра ветвления, продолжая построение дерева из листа с граничной оценкой 102.

После исключения из матрицы C_6' третьей строки, второго столбца, а также ее корректировки, получаем приведенную матрицу C_7:

$C_7 =$

	4	5
2	∞	0
5	0	∞

Так как матрица C_7 уже приведена и имеет размер 2×2, то ребра $(2, 5)$ и $(5, 4)$ непосредственно включаем в тур, при этом граничная оценка 102 не изменяется.

Окончательный вид тура представлен на рис. 12.15, а ограниченное дерево перебора — на рис. 12.16. Легко проверить, что сумма весов ребер, вошедших в тур, равна 102, что подтверждает правильность полученного решения.

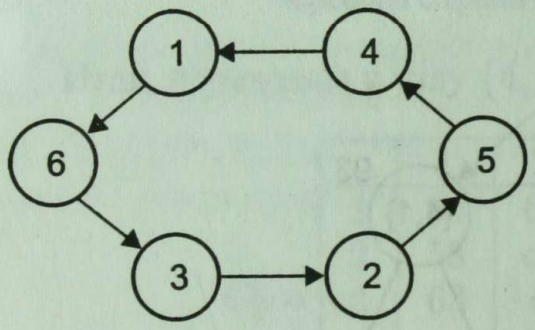


Рис. 12.15. Оптимальный тур

Пример 2. Решить задачу коммивояжера, используя алгоритм Литтла, для графа из 6 вершин, заданного матрицей смежности

$C =$

	1	2	3	4	5	6
1	–	6	4	8	7	14
2	6	–	7	11	7	10
3	4	7	–	4	3	10
4	8	11	4	–	5	11
5	7	7	3	5	–	7
6	14	10	10	11	7	–

Общая идея метода: нужно разделить огромное число перебираемых вариантов на классы и получить оценки (снизу — в задаче

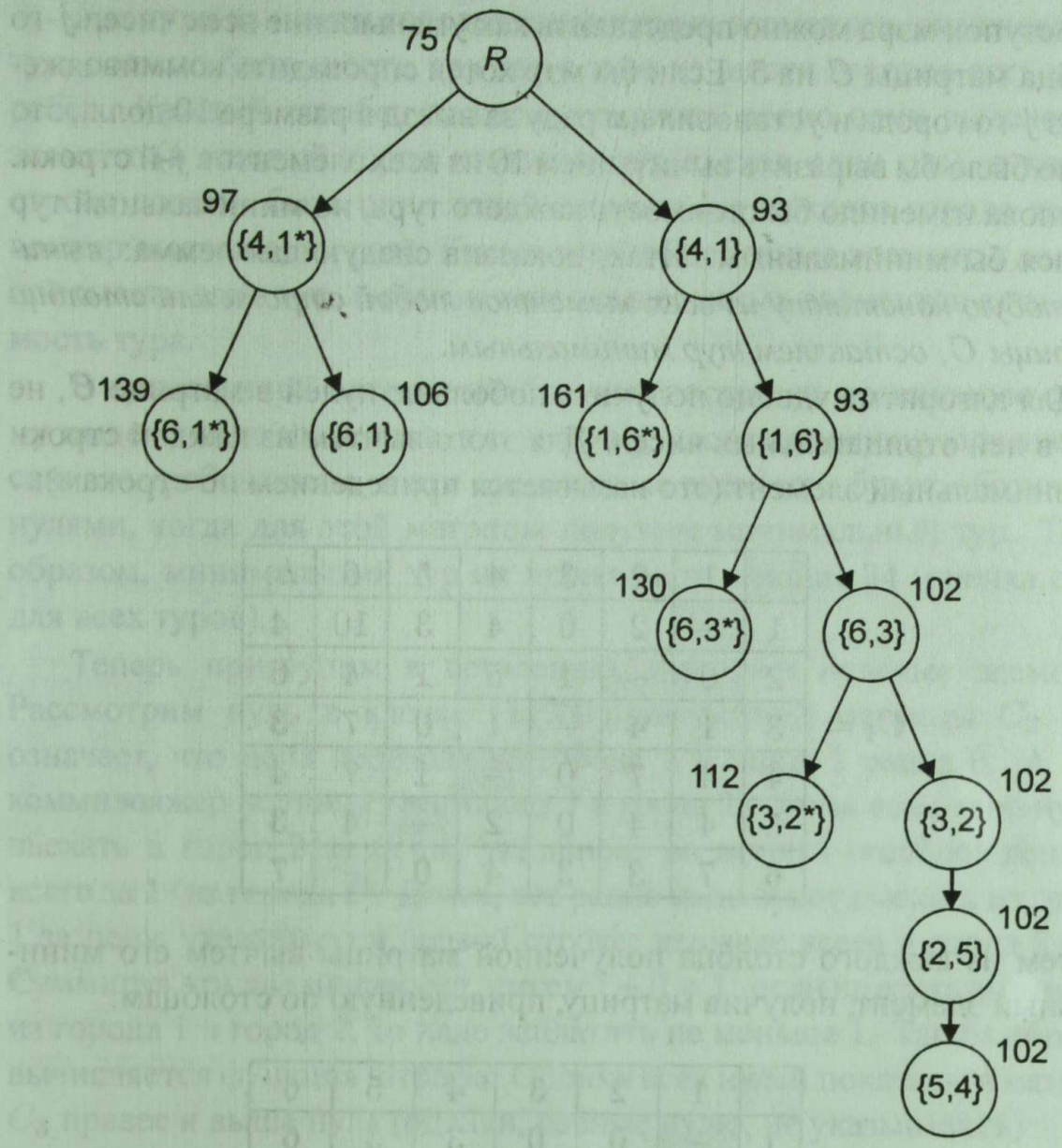


Рис. 12.16. Окончательный вид ограниченного дерева перебора

минимизации, сверху — в задаче максимизации) для этих классов, чтобы иметь возможность отбрасывать варианты не по одному, а целыми классами. Трудность состоит в том, чтобы найти такое разделение на классы (ветви) и такие оценки (границы), чтобы процедура была эффективной.

Будем трактовать C_{ij} как стоимость проезда из города i в город j. Допустим, что добрый мэр города j издал указ выплачивать каждому въехавшему в город коммивояжеру 5 долл. Это означает, что любой тур подешевеет на 5 долл., поскольку в любом туре нужно въехать в город j. Но поскольку все туры равномерно подешевели, то прежний минимальный тур будет и теперь стоить меньше всех. Добрый

же поступок мэра можно представить как уменьшение всех чисел j-го столбца матрицы C на 5. Если бы мэр хотел спровадить коммивояжеров из j-го города и установил награду за выезд в размере 10 долл., это можно было бы выразить вычитанием 10 из всех элементов j-й строки. Это снова изменило бы стоимость каждого тура, но минимальный тур остался бы минимальным. Итак, доказана следующая лемма: *вычитая любую константу из всех элементов любой строки или столбца матрицы C, оставляем тур минимальным.*

Для алгоритма удобно получить побольше нулей в матрице C, не имея в ней отрицательных чисел. Для этого вычтем из каждой строки ее минимальный элемент (это называется приведением по строкам):

$C_1 =$

	1	2	3	4	5	6	h_i
1	–	2	0	4	3	10	**4**
2	0	–	1	5	1	4	**6**
3	1	4	–	1	0	7	**3**
4	4	7	0	–	1	7	**4**
5	4	4	0	2	–	4	**3**
6	7	3	3	4	0	–	**7**

а затем из каждого столбца полученной матрицы вычтем его минимальный элемент, получив матрицу, приведенную по столбцам:

$C_2 =$

	1	2	3	4	5	6
1	–	0	0	3	3	6
2	0	–	1	4	1	0
3	1	2	–	0	0	3
4	4	5	0	–	1	3
5	4	2	0	1	–	0
6	7	1	3	3	0	–
h_j	0	**2**	0	**1**	0	**4**

Прочерки по диагонали означают, что из города i в город i проехать нельзя. Заметим, что сумма констант приведения по строкам равна 27, сумма по столбцам 7,

$$R = \sum_i h_i + \sum_j h_j = 27 + 7 = 34.$$

Для тура из шести городов выделенных элементов, включенных в тур, должно быть шесть, так как в туре из шести городов есть шесть ребер. Каждый столбец должен содержать ровно один выделенный элемент (в каждый город коммивояжер въехал один раз), в каждой строке должен быть один такой элемент (из каждого города коммивояжер выехал один раз). Кроме того, выделенные элементы должны описывать один тур, а сумма чисел выделенных элементов есть стоимость тура.

В приведенной матрице необходимо построить правильную систему выделенных элементов, т.е. систему, удовлетворяющую вышеописанным требованиям, и эти выделенные элементы будут обозначены нулями, тогда для этой матрицы получим минимальный тур. Таким образом, минимальный тур не может быть меньше 34 (оценка снизу для всех туров).

Теперь приступим к ветвлению, оценивая нулевые элементы. Рассмотрим нуль в клетке (1, 2) приведенной матрицы C_2. Он означает, что цена перехода из города 1 в город 2 равна 0. А если коммивояжер не поедет из города 1 в город 2? Тогда все равно нужно въехать в город 2 за цены, указанные во втором столбце; дешевле всего за 1 (из города 6). Далее, все равно надо будет выехать из города 1 за цену, указанную в первой строке; дешевле всего в город 3 за 0. Суммируя эти два минимума, имеем $1+0=1$: если не ехать «по нулю» из города 1 в город 2, то надо заплатить не меньше 1. Таким образом вычисляется функция штрафа. Оценки всех нулей показаны в матрице C_3 правее и выше нуля (оценки, равные нулю, не указывались):

$C_3 =$

	1	2	3	4	5	6
1	–	0^1	0	3	3	6
2	0^1	–	1	4	1	0
3	1	2	–	0^1	0	3
4	4	5	0^1	–	1	3
5	4	2	0	1	–	0
6	7	1	3	3	0^1	–

Выберем максимальную из этих оценок (в примере есть несколько оценок, равных единице, выберем первую из них в клетке (1, 2)).

Итак, выбрано нулевое ребро (1, 2). Разобьем все туры на два множества — включающие ребро (1, 2) и не включающие ребро (1, 2).

В первом случае необходимо рассмотреть матрицу C_4 с вычеркнутой первой строкой и вторым столбцом:

$C_4 =$

	1	3	4	5	6
2	0^1	1	4	1	0
3	1	–	0^1	0	3
4	4	0^1	–	1	3
5	4	0	1	–	0
6	7	3	3 ·	0^1	–

Дополнительно в уменьшенной матрице C_4 поставлен запрет в клетке (2, 1), так как выбрано ребро (1, 2) и замыкать преждевременно тур ребром (2, 1) нельзя.

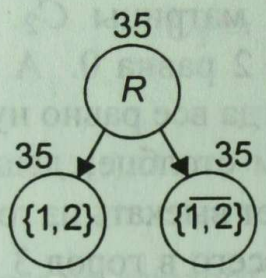


Рис. 12.17. Фрагмент дерева перебора решений

Уменьшенную матрицу можно привести на 1 по первому столбцу, при этом тур, отвечающий ей, будет стоить 35. Результат ветвлений и получения оценок показан на рис. 12.17.

Во фрагменте дерева перебора решений верхняя вершина — множество всех туров; нижняя левая — множество всех туров, включающих ребро (1, 2); нижняя правая — множество всех туров, не включающих ребро (1, 2).

Продолжим ветвление влево-вниз. Для этого оценим нули в матрице C_5:

$C_5 =$

	1	3	4	5	6
2	0^1	1	4	1	0
3	0^3	–	0^1	0	3
4	3	0^1	–	1	3
5	3	0	1	–	0
6	6	3	3	0^1	–

Максимальная оценка в клетке (3, 1) равна 3. Таким образом, оценка для правой нижней вершины на рис. 12.18 будет равна 35 + + 3 = 38. Для оценки левой нижней вершины (рис. 12.18) нужно

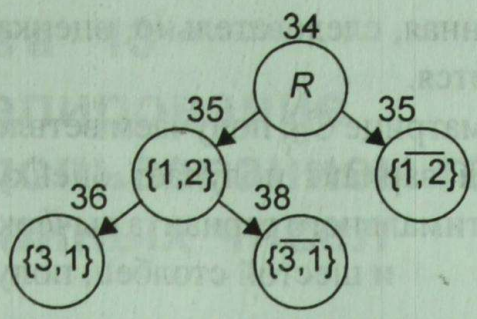


Рис. 12.18. Второй этап построения дерева решений

вычеркнуть из матрицы C_5 строку 3 и столбец 1, получив матрицу C_6:

$C_6 =$

	3	4	5	6
2	–	4	1	0
4	0^1	–	1	3
5	0	1	–	0
6	3	3	0^1	–

В эту матрицу нужно поставить запрет в клетку (2, 3), так как уже построен фрагмент тура из ребер $(1, 2)$ и $(3, 1)$ и нужно запретить преждевременное завершение тура $(2, 3)$. Матрица приводится по столбцу на 1, таким образом, каждый тур соответствующего класса (т.е. тур, содержащий ребра $(1, 2)$ и $(3, 1)$) стоит 36 условных единиц. Далее

$C_7 =$

	3	4	5	6
2	–	3	1	0
4	0^1	–	1	3
5	0	0^2	–	0
6	3	2	0^3	–

Теперь оцениваем нули в приведенной матрице C_7. Нуль с максимальной оценкой 3 находится в клетке $(6,5)$. Отрицательный вариант имеет оценку $38 + 3 = 41$. Для получения оценки положительного варианта удаляем строку 6 и столбец 5, ставим запрет в клетку $(5,6)$:

$C_8 =$

	3	4	6
2	–	3	0^3
4	0^3	–	3
5	0	0^3	–

Матрица C_8 приведенная, следовательно, оценка положительного варианта не увеличивается.

Оценивая нули в матрице C_8, получаем ветвление по выбору ребра $(2,6)$, отрицательный вариант получает оценку $36+3=39$, а для получения оценки оптимального варианта вычеркиваем вторую строку и шестой столбец, получая матрицу C_9:

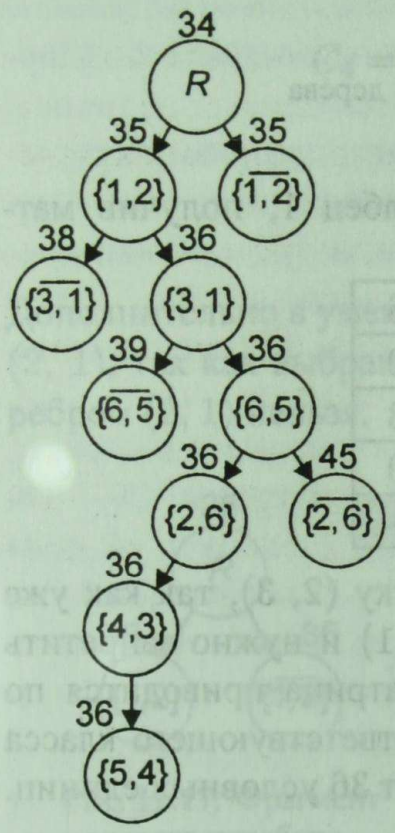


Рис. 12.19. Ограниченное дерево перебора

$$C_9 = \begin{array}{|c|c|c|} \hline & 3 & 4 \\ \hline 4 & 0 & - \\ \hline 5 & - & 0 \\ \hline \end{array}$$

В матрицу надо добавить запрет в клетку $(5,3)$, так как уже построен фрагмент тура $[3,1,2,6,5]$ и надо запретить преждевременное завершение тура $(5,3)$. После преобразований получилась матрица 2×2 с нулевыми элементами по диагонали. Элементы, помеченные нулем (ребра $(4,3)$ и $(5,4)$), автоматически включаются в тур, не меняя его длины.

Таким образом, получен тур: $(1-2-6-5-4-3-1)$ стоимостью в 36 (рис. 12.19). При достижении низа по дереву перебора класс туров сузился до одного тура, а оценка снизу превратилась в точную стоимость.

Контрольные вопросы

1. Какое множество называется рекордом?
2. В чем сущность улучшения рекорда?
3. Укажите главный принцип жадного алгоритма.
4. В чем особенность деревянного алгоритма?
5. Как строится дерево перебора для расшифровки криптограмм?
6. Назовите основные принципы метода ветвей и границ.
7. Сформулируйте условие задачи коммивояжера.
8. Что означает «привести матрицу по строкам»?
9. Что такое функция штрафа?
10. Как исключается досрочное завершение тура?
11. Что такое нижняя граничная оценка?

Глава 13
Моделирование с использованием генераторов случайных чисел

Многие явления в природе, технике, экономике и в других областях носят случайный характер. В этом случае величина, принимающая свои значения в зависимости от исходов некоторого испытания (опыта), называется *случайной величиной.*

Пусть X — дискретная случайная величина, возможными значениями которой являются числа $x_1, x_2, \ldots, x_n$.

Обозначим через $p_i = P(X = x_i)$, $i = 1, 2, \ldots, n$, вероятности этих значений, т.е. p_i — вероятность события, состоящего в том, что X принимает значение x_i.

Закон распределения полностью задает дискретную случайную величину. Однако часто закон распределения случайной величины неизвестен. В таких ситуациях ее описывают числовыми характеристиками.

13.1. Числовые характеристики случайных величин

Случайные величины характеризуются следующими числовыми параметрами:

• ***Математическое ожидание*** $M(X)$ — это статистическое среднее случайной величины:

$$M(X) = \sum_{i=1}^{n} x_i p_i,$$

где x_i — значение случайной величины, p_i — вероятность появления этой величины. При этом

$$\sum_{i=1}^{n} p_i = 1.$$

Для равновероятных событий

$$M(X) = \frac{1}{n} \sum_{i=1}^{n} x_i.$$

• ***Дисперсия*** — это математическое ожидание квадрата отклонения случайной величины от ее математического ожидания:

$$D(X) = \sum_{i=1}^{n} (x_i - M(X))^2 p_i.$$

• Корень квадратный из дисперсии называется ***средним квадратичным отклонением*** $\sigma(X)$ случайной величины, т.е.

$$\sigma(X) = \sqrt{D(X)}.$$

• ***Коэффициент корреляции*** определяется для двух потоков случайных величин:

$$R(X,Y) = \frac{M(XY) - M(X)M(Y)}{\sigma(X)\sigma(Y)}.$$

Коэффициент корреляции определен на отрезке $[-1; 1]$, т.е. $-1 \leq \leq R(X,Y) \leq 1$.

Потоки случайных величин, для которых $R(X,Y) = 0$, называются *некоррелированными* (*независимыми*).

Рассмотрим алгоритмы для детерминированной выборки случайных чисел.

13.2. Метод середины квадрата

Один из ранних генераторов случайных чисел, принадлежащий Джону фон Нейману (1946), известен как *метод середины квадрата*.

Метод используется для генерации k-разрядных псевдослучайных чисел. Должно быть задано k-разрядное начальное число x_0 (для удобства предполагаем, что k четно). Это число возводится в квадрат, и получается число y. Число y должно иметь $2k$ разрядов. Если число разрядов меньше $2k$, то число y дополняется слева нулями. Затем в y выделяют средние k разрядов, которые и дают новое случайное число.

Алгоритм сводится к выполнению следующих шагов.

Шаг 0. Инициализация: $x := x_0$.

Шаг 1. Основной цикл: FOR $j := 1$ TO m DO шаг 2; Stop.

Шаг 2. Генерация нового случайного числа x_j: $y := x^2$; $x_j :=$ (средние k разрядов числа y); $x := x_j$. (Число y будет иметь $2k$ разрядов, а следующее случайное число x_j получается, если удалить по

$k/2$ разрядов с каждого конца y. Десятичная точка помещается перед первым разрядом числа x_j до поступления его на выход случайного генератора.)

Пример 1. Получить три случайных числа методом середины квадрата. Выберем $k = 4$; $x_0 = 3167$. Тогда:

$$\begin{aligned} x_0 &= 3167; \quad y = 10\underline{0298}89; \\ x_1 &= 0298; \quad y = 00\underline{0888}04; \\ x_2 &= 0888; \quad y = 00\underline{7885}44; \\ x_3 &= 7885. \end{aligned}$$

Пример 2. Для $k = 4$ и $x_0 = 2134$ получить первые восемь псевдослучайных чисел, выработанных алгоритмом середины квадрата, в интервале (0; 1):

$$\begin{aligned} x_0^2 &= 04\,553\,956, \quad x_1 = 0{,}5539; \\ x_1^2 &= 30\,680\,521, \quad x_2 = 0{,}6805; \\ x_2^2 &= 46\,308\,025, \quad x_3 = 0{,}3080; \\ x_3^2 &= 09\,486\,400, \quad x_4 = 0{,}4864; \\ x_4^2 &= 23\,658\,496, \quad x_5 = 0{,}6584; \\ x_5^2 &= 43\,349\,056, \quad x_6 = 0{,}3490; \\ x_6^2 &= 12\,180\,100, \quad x_7 = 0{,}1801; \\ x_7^2 &= 03\,243\,601, \quad x_8 = 0{,}2436. \end{aligned}$$

Несмотря на видимую случайность чисел, генерируемых алгоритмом, ему свойственны недостатки. В самом деле, если в последовательности когда-нибудь появится число 0,0000, то все следующие за ним числа будут также равны 0,0000. Таким образом, многое зависит от начального выбора k и x_0.

13.3. Линейный конгруэнтный метод

Метод используется для генерации последовательности $x_1, x_2, \ldots, x_m$ из m псевдослучайных чисел. Должны быть заданы следующие входные значения:

- b — целочисленный множитель, $b \geq 1$;
- x_0 — начальное случайное целое число $x_0 \geq 1$;

• k — шаг, $k \geq 0$ целое;

• m — целочисленный модуль, $m \geq x_0, b, k$.

Случайные числа генерируются по рекуррентной формуле:

$$x_j = (bx_{j-1} + k) \bmod m.$$

Алгоритм сводится к выполнению следующих шагов.

Шаг 1. Основной цикл:

FOR $j := 1$ TO m DO шаг 2; шаг 3; Stop.

Шаг 2. Генерация нового необработанного случайного числа:

$$x_j := (bx_{j-1} + k) \bmod m.$$

Число x_j должно лежать в полуинтервале $0 \leq x_j < m$. По определению для целых a и m ($a \bmod m$) есть остаток от целочисленного деления a на m. Например, $5 \bmod 3 = 2$, $7 \bmod 3 = 1$, $9 \bmod 3 = 0$.

Шаг 3. Генерация следующего случайного числа: $y_j := x_j/m$ (y_j будет лежать в полуинтервале $0 \leq y_j < 1$ и обладать требуемым распределением).

Пример 1. Получить три случайных числа линейным конгруэнтным методом. Выбираем $x_0 = 27$; $b = 5$; $k = 10$; $m = 40$. В результате:

$$x_1 = (5 \cdot 27 + 10) \bmod 40 = 145 \bmod 40 = 25;$$

$$x_2 = (5 \cdot 25 + 10) \bmod 40 = 15;$$

$$x_3 = (5 \cdot 15 + 10) \bmod 40 = 5.$$

Пример 2. Для $k = 0$, $m = 2^{10}$, $b = 101$, $x_0 = 432$ получить первые восемь псевдослучайных чисел, выработанных линейным конгруэнтным методом, при этом

$$x_1 = 624, \quad y_1 = \frac{624}{1024} = 0,610;$$

$$x_2 = 560, \quad y_2 = \frac{560}{1024} = 0,546;$$

$$x_3 = 240, \quad y_3 = \frac{240}{1024} = 0,234;$$

$$x_4 = 688, \quad y_4 = \frac{688}{1023} = 0,673;$$

$$x_5 = 880, \quad y_5 = \frac{880}{1024} = 0,859;$$

$$x_6 = 816, \quad y_6 = \frac{816}{1024} = 0,796;$$

$$x_7 = 496, \quad y_7 = \frac{496}{1024} = 0,486;$$

$$x_8 = 944, \quad y_8 = \frac{944}{1024} = 0,923.$$

Существует такой выбор параметров k, b, m, x_0, при котором алгоритм будет генерировать числа на отрезке [0; 1], представляющиеся непредсказуемыми и удовлетворяющие определенным статистическим критериям. Для всех практических целей эти числа оказываются последовательностью наблюдений равномерно распределенной случайной переменной.

13.4. Полярный метод генерации случайных чисел

Метод используется для генерации двух независимых случайных чисел с нормальным распределением $N(0,1)$ из двух независимых равномерно распределенных случайных чисел. Распределение $N(0,1)$ преобразуется в $N(\mu, \sigma^2)$ с использованием центральной предельной теоремы.

Алгоритм сводится к выполнению следующих шагов.

Шаг 1. Генерация двух равномерно распределенных случайных чисел. Генерируются два независимых случайных числа u_1 и u_2 с распределением $U(0,1)$; $\nu_1 := 2u_1 - 1$; $\nu_2 := 2u_2 - 1$; (ν_1 и ν_2 имеют распределение $U(-1,1)$).

Шаг 2. Вычисление и проверка s: $s := \nu_1^2 + \nu_2^2$; IF $s \geq 1$ THEN GOTO шаг 1.

Шаг 3. Вычисление n_1 и n_2: $n_1 := \nu_1 \sqrt{-\frac{2\ln s}{s}}$; $n_2 := \nu_2 \sqrt{-\frac{2\ln s}{s}}$.

Шаг 4. Stop. (n_1 и n_2 распределены нормально.)