Оглавление

ПРЕДИСЛОВИЕ

Решение любой вычислительной задачи предполагает выполнение определенной последовательности шагов. При этом один и тот же результат может быть получен различными способами. Один из способов может быть более эффективным, другой менее эффективным, но более легким в реализации. Алгоритм, будучи той самой последовательностью шагов, можно оценить, как это будет показано дальше. Имея набор алгоритмов, предназначенных для решения какой-либо проблемы, программист в состоянии выбирать удовлетворяющий его нуждам.

В учебном пособии приведены различные алгоритмы решения довольно широкого класса задач. Данные алгоритмы не привязаны к какому-либо языку программирования, хотя предполагается, что программная реализация будет выполняться на языке программирования высокого уровня. Теоретическая часть наглядно иллюстрируется примерами на языках *Pascal*, Си и псевдоязыке, сочетающем высказывания на естественном языке с одним из указанных выше языков программирования.

Приведенные в пособии алгоритмы представляют собой «классику» в области обработки данных. Существенный вклад в развитие современной теории алгоритмов и алгоритмизации решения практических задач внесли такие ученые, как Н. Вирт [2, 3], Д. Кнут [4], Дж. Макконел [6], к трудам которых можно обратиться для более глубокого исследования проблемы построения и анализа алгоритмов. Большое количество информации об алгоритмах можно также найти в сети Интернет, как на англоязычных, так и на русских сайтах.

Невозможно говорить о практической или даже формальной реализации алгоритмов безотносительно структур и типов данных. В пособии структуры данных не выносятся в отдельную главу, а рассматриваются по мере необходимости в контексте тех или иных алгоритмов обработки данных.

В учебном пособии приводятся алгоритмы решения наиболее распространенных задач, фундаментальные алгоритмы и современные методы обработки данных.

Пособие состоит из восьми глав, посвященных алгоритмам решения различных классов задач.

В первой главе даны основные сведения об алгоритмах, их свойствах. Отражена разница между такими понятиями, как «структура данных», «тип данных» и «абстрактный тип данных». Показано, как может измеряться временная сложность алгоритма, которая в дальнейших главах служит основным критерием оценки скорости выполнения алгоритма.

Вторая глава касается поиска образа в строке. Наглядным примером этой задачи может служить поиск слова в тексте в редакторе *Word*.

Вообще, задача поиска является одной из фундаментальных задач теоретического программирования. Приведится алгоритм прямого поиска, а также два алгоритма, позволяющих произвести более эффективный поиск.

Сортировка, или упорядочение, ряда объектов по возрастанию или убыванию является одной из основных задач обработки данных. Выбор алгоритма и даже класса алгоритмов, используя который возможно произвести сортировку, зависит от структуры данных. В связи с этим выделяют два типа сортировки: внутреннюю и внешнюю.

Третья глава пособия посвящена алгоритмам внутренней сортировки, при которой все данные помещаются в оперативную память, где можно получить доступ к ним в любом порядке. Работа алгоритмов данного класса показана на примере упорядочения элементов массива.

Внешняя сортировка применяется в том случае, когда объем упорядочиваемых данных настолько большой, что невозможно поместить все данные в оперативную память. Здесь задействуется механизм обмена большими блоками данных между внешними запоминающими устройствами и оперативной памятью. Тот факт, что физически непрерывные данные необходимо для удобства перемещения организовывать в блочную структуру, вынуждает использовать специальные алгоритмы внешней сортировки, о которых пойдет речь в четвертой главе (предполагается, что все данные хранятся в файлах).

В пятой главе рассматриваются ориентированные графы, решается такая важная задача, как поиск кратчайшего пути между двумя вершинами. Умение решать данную задачу помогает в планировании кратчайшего маршрута, например при передвижении на транспортном средстве или передаче сообщений по компьютерной сети. В этой же главе показано, как можно перебрать все вершины графа методом обхода в глубину, что может потребоваться при поиске одной вершины во всем множестве вершин графа.

В шестой главе приводятся основные определения, связанные с неориентированными графами. Помимо обхода в глубину рассматривается обход в ширину, который, впрочем, также применим и к ориентированным графам. Другой немаловажной задачей является построение минимального остовного дерева графа. Одно из применений минимальных остовных деревьев – организация локальной компьютерной сети с минимальными затратами на объединение всех компьютеров в единую сеть. Здесь же рассматривается связность графа и приводится задача о паросочетании.

Седьмая глава посвящена решению задачи о максимальном потоке в графе. Приведенные здесь алгоритмы применяются на транспортных, коммуникационных и электрических сетях, при моделировании различных процессов физики, в некоторых операциях над матрицами. Ввиду большой практической значимости исследованиями в данной области занимаются во многих крупнейших университетах мира. Как ни странно, в русскоязычной литературе передовые алгоритмы нахождения максимального потока осве-

щены слабо. В данном пособии приведены как классические алгоритмы решения данной задачи, так и их современные модификации.

Восьмая глава содержит краткое изложение некоторых идей, на основе которых разрабатываются современные алгоритмы. Примечательно, что данные алгоритмы используются не только для решения практических задач, но также активно применяются в научных исследованиях. Генетические алгоритмы можно встретить в работах по различным направлениям. Здесь же представлены общие сведения о новом перспективном подходе к решению задач оптимизации на основе муравьиных алгоритмов. Особый интерес представляет тот факт, что в их основе лежит моделирование поведения колонии муравьев.

Несмотря на столь широкий спектр рассмотренных алгоритмов, данное учебное пособие не претендует на полноту охвата всех проблем современной теории алгоритмов и их применения. Тем не менее, информации, представленной в пособии, достаточно для решения реальных задач. Рассмотренные алгоритмы могут быть применены в большом числе предметных областей и практических ситуаций.

1. ОБЩИЕ СВЕДЕНИЯ ОБ АЛГОРИТМАХ

В основе любой компьютерной программы всегда лежит некоторый алгоритм, программа является изложением алгоритма на некотором языке, понятном вычислительной машине.

Первым дошедшим до нас алгоритмом в его интуитивном понимании как конечной последовательности элементарных действий, решающих поставленную задачу, считается предложенный Евклидом в III веке до нашей эры алгоритм нахождения наибольшего общего делителя двух чисел. Отметим, что в течение длительного времени, вплоть до начала XX века, само слово «алгоритм» употреблялось в устойчивом сочетании «алгоритм Евклида». Для описания последовательности пошагового решения других математических задач чаще использовался термин «метод».

Во всех сферах своей деятельности, в частности в сфере обработки информации, человек сталкивается с различными способами или методиками решения разнообразных задач. Они определяют порядок выполнения действий для получения желаемого результата. Можно трактовать это как первоначальное или интуитивное определение алгоритма. Таким образом, можно нестрого определить алгоритм как однозначно трактуемую процедуру решения задачи. Дополнительные требования о выполнении алгоритма за конечное время для любых входных данных приводят к следующему неформальному определению алгоритма:

Алгоритм – это заданное на некотором языке конечное предписание, задающее конечную последовательность выполнимых и точно определенных элементарных операций для решения задачи, общее для класса возможных исходных данных.

Пусть Dz – область (множество) исходных данных задачи Z, а R – множество возможных результатов, тогда можно говорить, что алгоритм осуществляет отображение $Dz \rightarrow R$. Поскольку такое отображение может быть не полным, то вводятся следующие понятия:

Алгоритм называется частичным алгоритмом, если результат может быть получен только для некоторых $d \in Dz$ и полным алгоритмом, если алгоритм получает правильный результат для всех $d \in Dz$.

Несмотря на усилия ученых, сегодня отсутствует одно исчерпывающе строгое определение понятия «алгоритм». Из разнообразных вариантов словесного определения алгоритма наиболее удачные, по мнению автора, принадлежат российским ученым А. Н. Колмогорову и А. А. Маркову:

Определение по Колмогорову: алгоритм – это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Определение по Маркову: Алгоритм – это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату.

Отметим, что различные определения алгоритма, в явной или неявной форме, постулируют следующий ряд общих требований:

- алгоритм должен содержать конечное количество элементарно выполнимых предписаний, т. е. удовлетворять требованию конечности записи;
- алгоритм должен выполнять конечное количество шагов при решении задачи, т. е. удовлетворять требованию конечности действий;
- алгоритм должен быть единым для всех допустимых исходных данных, т. е. удовлетворять требованию универсальности;
- алгоритм должен приводить к правильному по отношению к поставленной задаче решению, т. е. удовлетворять требованию правильности.

Неудобства словесных определений связаны с проблемой однозначной трактовки терминов. В таких определениях должен быть, хотя бы неявно, указан исполнитель действий или предписаний. Алгоритм вычисления производной для полинома фиксированной степени вполне ясен тем, кто знаком с основами математического анализа, но для прочих он может оказаться совершенно непонятным. Это рассуждение заставляет указать так же вычислительные возможности исполнителя, а именно уточнить какие операции для него являются «элементарными». Другие трудности связаны с тем, что алгоритм заведомо существует, но его очень трудно описать в некоторой заранее заданной форме. Классический пример такой ситуации – алгоритм завязывания шнурков на ботинках «в бантик». Вы сможете дать только словесное описание этого алгоритма без использования иллюстраций ?

В связи с этим формально строгие определения понятия алгоритма связаны с введением специальных математических конструкций – формальных алгоритмических систем или моделей вычислений, каковыми являются машина Поста, машина Тьюринга, рекурсивно-вычислимые функции Черча, и постулированием тезиса об эквивалентности такого формализма и понятия «алгоритм». Несмотря на принципиально разные модели вычислений, использующиеся для определения термина «алгоритм», интересным результатом является формулировка гипотез о эквивалентности этих формальных определений в смысле их равномощности.

1.1. Свойства алгоритмов

В качестве содержательных свойств, характеризующих алгоритм, А. А. Марков отмечает следующие:

1. Определенность. Алгоритм должен быть точным, недвусмысленным и понятным исполнителю. Исполнитель, выполняя алгоритм, должен однозначно понимать предписание и знать, что надлежит делать на данном шаге вычислительного процесса и каков будет следующий шаг. Определенность не всегда означает детерминированность, т. е. когда на каждом шаге вычис-

лительного процесса исполнитель должен исполнить единственное предписываемое действие и результат этого действия определен.

Хотя в дальнейшем будем предполагать, что алгоритм детерминирован, однако не стоит забывать, что могут существовать и недетерминированные алгоритмы, особенно при наличии коллектива исполнителей (что характерно для параллельного программирования). При недетерминированности на некоторых шагах вычислительного процесса возможен не определяемый алгоритмом выбор из конечного фиксированного набора действий, но этот набор должен быть точно определен – определенность есть неотъемлемое свойство алгоритма.

2. Массовость. Алгоритм предлагает всегда решение некоторой массовой проблемы, его исходные данные варьируются. Существует некоторое (заведомо не пустое и не единичное) множество наборов исходных данных, определяемое решаемой проблемой, для которых алгоритм обеспечивает получение искомого результата.

Не для любой массовой проблемы существует решающий ее алгоритм – в теории алгоритмов для ряда массовых проблем доказана их неразрешимость, т. е. отсутствие алгоритма, получающего искомый результат для множества наборов исходных данных, возможного для этой проблемы. Близкой для нас неразрешимой массовой проблемой является установление эквивалентности двух произвольных программ – доказано, что не существует алгоритма, который для двух произвольных программ устанавливал бы, всегда ли они для одинакового набора исходных данных получают одинаковый результат. Вместе с тем нас должен утешать тот факт, что большинство реальных проблем в их надлежащей постановке вполне допускают наличие алгоритма их решения.

3. Результативность. Обычно алгоритм должен обеспечивать завершение своего выполнения ожидаемым результатом вычислительного процесса в конечное (и хотелось бы, приемлемое) время, разумеется, при надлежащих допустимых исходных данных. Допустимость исходных данных следует из существа решаемой проблемы: так, если все исходные данные должны быть положительны, то нельзя требовать от алгоритма разумной реакции на то, что одно из данных ошибочно оказалось отрицательным. Вместе с тем обычно стараются обезопасить алгоритм предварительной проверкой исходных данных на допустимость и при их недопустимости явно сообщать пользователю об этом. В пользовательском программировании все реализуемые алгоритмы носят такой завершающийся характер. Однако результатом алгоритма может быть и поддержание некоторого постоянного процесса: процесса управления некоторым устройством, процесса контроля его состояния, наконец, процесса операционной системы, управляющей функционированием компьютера. Результатами таких алгоритмов являются сигналы и сообщения, посылаемые пользователю или устройству, и в этом их результативность. Такие алгоритмы, как правило, находятся за рамками пользовательского программирования.

1.2. Примеры алгоритмов

Классической массовой проблемой, известной с древности, является нахождение наибольшего общего делителя (НОД) двух натуральных чисел. Алгоритм, решающий эту проблему, может быть следующим.

Дана пара произвольных чисел, назовем их m и n. Предписание, представляющее алгоритм, будем описывать на естественном языке в виде последовательности нумеруемых шагов:

Шаг 1. Возьмем в качестве p значение наименьшего из m и n.

Шаг 2. Возьмем в качестве t значение 1.

Шаг 3. Если $p = 1$, то перейдём к шагу 5, иначе к шагу 4.

Шаг 4. Задавая в качестве i последовательно все целые значения от 2 до p (по возрастанию), повторяем Шаг 4.1.

Шаг 4. 1. Если остаток $(m / i) = 0$ и остаток $(n / i) = 0$, то возьмем в качестве t значение i.

Шаг 5. Завершаем алгоритм с результатом НОД = t.

С точки зрения исполнителя-человека это предписание точно и однозначно. Шаги исполняются последовательно, причем шаг 4 является повторением определенного числа раз шага 4. 1 каждый раз со следующим значением i. Будем считать, что мы знаем, как находить остаток от деления, т. е. определенность здесь присутствует.

То, что приведенное предписание обладает массовостью, тоже очевидно. Существует множество пар чисел, для которых предписание дает результат. Ясно, что вычислительный процесс будет нормально исполняться для любых конечных значений m и n, больших нуля. Именно такие пары и дают нам допустимые для решаемой проблемы значения исходных данных.

В данном случае результативность совпадает с завершаемостью. Завершаемость вычислительного процесса следует из того, что m и n конечны, и, следовательно, число повторений шага 4. 1 тоже конечно, остальные же шаги выполняются однократно. Число исполненных шагов зависит прежде всего от числа повторений шага 4. 1, а он будет повторяться $(\min(m, n) – 1)$ раз. То, что результатом будет действительно наибольший общий делитель, очевидно из того, что если одно из значений пары есть 1, то полученным НОД будет 1; если ни одно из чисел от 2 до наименьшего из пары не является одновременно делителем и n, и m, то полученный НОД тоже 1, а если среди этих чисел есть делители и n, и m, то значением t после шага 4 будет наибольшее такое число. Именно оно будет последним и на шаге 4. 1 будет взято в качестве t. Итак, приведенное предписание обладает всеми упомянутыми свойствами алгоритма.

Заметим, что здесь мы применили подход «в лоб» – перебрали все значения, которые в принципе могут быть делителями m и n, и проверили, может ли каждое быть общим делителем, а найдя больший, чем ранее найден-

ный, делитель, заменили им ранее найденный. Такие, так называемые переборные, алгоритмы (перебираются все потенциально возможные значения решений) мы вынуждены применять, если у нас нет никаких соображений, как сократить число перебираемых значений. Зачастую такие соображения у нас могут быть. Так, для данной проблемы давно известно, что

$$НОД\ (m, n) = НОД\ (остаток\ (n / m), m)$$

где m – наименьшее число пары. Таким образом, нахождение НОД для пары чисел можно свести к нахождению НОД для такой пары, в которой прежнее наименьшее становится наибольшим. Перебор уже становится перебором пар, определяемых приведенным выше соотношением, что сокращает его по отношению к уже приведенному алгоритму. Алгоритм (так называемый алгоритм Эвклида) выглядит следующим образом:

Шаг 1. Пока m не равно 0, выполнять шаги 1. 1 – 1. 4, иначе перейти к шагу 2.
Шаг 1. 1. Возьмем значение t как остаток (n / m).
Шаг 1. 2. Возьмем n как значение m.
Шаг 1. 3. Возьмем m как значение t.
Шаг 1. 4. Вернемся к шагу 1.
Шаг 2. Завершаем алгоритм с НОД = n.

Приведенное предписание действительно является алгоритмом, так как оно обладает определенностью (как и предыдущее предписание), массовостью (о допустимых значениях мы поговорим далее) и результативностью (повторения шагов 1. 1 – 1. 4 создают пары с уменьшающимися неотрицательными значениями и, значит, рано или поздно m станет равным нулю). Что касается допустимых значений, то ясно, что m и n должны быть больше нуля (впрочем, они могут быть и равны нулю: если $m = 0$, то за НОД возьмется значение n). На первый взгляд, из предыдущих рассуждений следует, что от должно быть не больше n. Однако если проанализировать исполнение алгоритма, то увидим, что если $m > n$, то при первом же исполнении шагов 1. 1 – 1. 4 значения переставятся: большее станет значением n, а меньшее – значением m, что и нужно для дальнейшего исполнения алгоритма.

Число шагов алгоритма здесь оценить гораздо труднее, чем в предыдущем случае – оно существенно зависит от свойств m и n по отношению друг к другу. Средняя ожидаемая оценка числа повторений шагов 1. 1 – 1. 4 равна приблизительно $(12 * ln\ 2/ \pi^2) * (ln\ m)$. Видно, что это значительно меньше, чем в предыдущем случае, и привлеченные соображения позволили существенно уменьшить перебор.

Важно, что в последнем примере более явно, чем в предыдущем, видна изменяемость данных – исходная пара, идентифицируемая обозначениями m и n, меняется и перестраивается, сохраняя введенные обозначения. Видим, что приходится вводить объекты с фиксированным обозначением и меняющимся значением – то, что в языках программирования называется переменной.

1.3. Типы данных, структуры данных и абстрактные типы данных

Хотя термины тип данных (или просто тип), структура данных и абстрактный тип данных звучат похоже, но имеют они различный смысл. В языках программирования тип данных переменной обозначает множество значений, которые может принимать эта переменная. Например, переменная булевого (логического) типа может принимать только два значения: значение *true* (истина) и значение *false* (ложь) и никакие другие. Набор базовых типов данных отличается в различных языках: в языке *Pascal* это типы целых (*integer*) и действительных (*real*) чисел, булев (*boolean*) тип и символьный (*char*) тип. Правила конструирования составных типов данных на основе базовых типов также различаются в разных языках программирования. В таких языках высокого уровня, как Си и *Pascal* легко и быстро строить составных типы.

Абстрактный тип данных (АТД) – это математическая модель плюс различные операторы, определенные в рамках этой модели. Алгоритм может разрабатываться в терминах АТД, но для реализации алгоритма в конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов, поддерживаемых данным языком программирования. Для представления АТД используются структуры данных, которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом.

Базовым строительным блоком структуры данных является ячейка, которая предназначена для хранения значения определенного базового или составного типа данных. Структуры данных создаются путем задания имен совокупностям (агрегатам) ячеек и (необязательно) интерпретации значения некоторых ячеек как представителей (т. е. указателей) других ячеек.

В качестве простейшего механизма агрегирования ячеек в *Pascal* и большинстве других языков программирования можно применять (одномерный) массив, т. е. последовательность ячеек определенного типа. Массив также можно рассматривать как отображение множества индексов (таких как целые числа 1, 2, …, *n*) во множество ячеек. Ссылка на ячейку обычно состоит из имени массива и значения из множества индексов данного массива. В языке *Pascal* множество индексов может быть нечислового типа, например (север, восток, юг, запад), или интервального типа (как 1..10). Значения всех ячеек массива должны иметь одинаковый тип данных. Объявление

имя: *array*[ТипИндекса] *of* ТипЯчеек;

задает имя для последовательности ячеек, тип для элементов множества индексов и тип содержимого ячеек.

Кстати, *Pascal* необычайно богат на типы индексов. Многие языки программирования позволяют использовать в качестве индексов только множества последовательных целых чисел. Например, чтобы в языке *Fortran*

в качестве индексов массива можно было использовать буквы, надо все равно использовать целые индексы, заменяя «A» на 1, «B» на 2, и т. д.

Другим общим механизмом агрегирования ячеек в языках программирования является структура записи. Запись (*record*) можно рассматривать как ячейку, состоящую из нескольких других ячеек (называемых полями), значения в которых могут быть разных типов. Записи часто группируются в массивы; тип данных определяется совокупностью типов полей записи. Например, в *Pascal* объявление

```
var
    reclist: array[1..4] of record
    data: real;
    next: integer
end
```

задает имя *reclist* (список записей) 4-элементного массива, значениями которого являются записи с двумя полями: *data* (данные) и *next* (следующий).

Третий метод агрегирования ячеек, который можно найти в *Pascal* и некоторых других языках программирования, – это файл. Файл, как и одномерный массив, является последовательностью значений определенного типа. Однако файл не имеет индексов: его элементы доступны только в том порядке, в каком они были записаны в файл. В отличие от файла, массивы и записи являются структурами с «произвольным доступом», подразумевая под этим, что время доступа к компонентам массива или записи не зависит от значения индекса массива или указателя поля записи. Достоинство агрегирования с помощью файла (частично компенсирующее описанный недостаток) заключается в том, что файл не имеет ограничения на количество составляющих его элементов и это количество может изменяться во время выполнения программы.

1.4. Абстрактные типы данных

Перед тем, как рассмотреть абстрактный тип данных, обсудим его роль в процессе разработки программ. Сначала сравнивним абстрактный тип данных с таким знакомым понятием, как процедура.

Процедуру, неотъемлемый инструмент программирования, можно рассматривать как обобщенное понятие оператора. В отличие от ограниченных по своим возможностям встроенных операторов языка программирования (сложения, умножения и т. п.), с помощью процедур программист может создавать собственные операторы и применять их к операндам различных типов, не только базовым. Примером такой процедуры-оператора может служить стандартная подпрограмма перемножения матриц.

Другим преимуществом процедур (кроме способности создавать новые операторы) является возможность использования их для инкапсулирования частей алгоритма путем помещения в отдельный раздел программы всех операторов, отвечающих за определенный аспект функционирования программы. Пример инкапсуляции: использование одной процедуры для чтения входных данных любого типа и проверки их корректности. Преимущество инкапсуляции заключается в том, что мы знаем, какие инкапсулированные операторы необходимо изменить в случае возникновения проблем в функционировании программы. Например, если необходимо организовать проверку, являются ли значения входных данных положительными, следует изменить только несколько строк кода, и мы точно знаем, где эти строки находятся.

Определение абстрактного типа данных

Определим абстрактный тип данных (АТД) как математическую модель с совокупностью операторов, определенных в рамках этой модели. Простым примером АТД могут служить множества целых чисел с операторами объединения, пересечения и разности множеств. В модели АТД операторы могут иметь операндами не только данные, определенные АТД, но и данные других типов: стандартных типов языка программирования или определенных в других АТД. Результат действия оператора также может иметь тип, отличный от определенных в данной модели АТД. Но мы предполагаем, что по крайней мене один операнд или результат любого оператора имеет тип данных, определенный в рассматриваемой модели абстрактных типов данных.

Две характерные особенности процедур – обобщение и инкапсуляция, – о которых говорилось выше, отлично характеризуют абстрактный тип данных. АТД можно рассматривать как обобщение простых типов данных (целых и действительных чисел и т. д.), точно так же, как процедура является обобщением простых операторов (+, – и т. д.). Абстрактный тип данных инкапсулирует типы данных в том смысле, что определение типа и все операторы, выполняемые над данными этого типа, помещаются в один раздел программы. Если необходимо изменить реализацию АТД, мы знаем, где найти и что изменить в одном небольшом разделе программы, и можем быть уверенными, что это не приведет к ошибкам где-либо в программе при работе с этим типом данных. Более того, вне раздела с определением операторов АТД можно рассматривать типы АТД как первичные типы, так как объявление типов формально не связано с их реализацией. Однако в этом случае могут возникнуть сложности, так как некоторые операторы могут инициироваться для более одного АТД и ссылки на эти операторы должны быть в разделах нескольких АТД.

Можно реализовать тип данных любым способом, а программы, использующие объекты этого типа, не зависят от способа реализации типа – за это отвечают процедуры, реализующие операторы для этого типа данных.

Термин реализация АТД подразумевает следующее: перевод в операторы языка программирования объявлений, определяющие переменные этого абстрактного типа данных, плюс процедуры для каждого оператора, выполняемого над объектами АТД. Реализация зависит от структуры данных, представляющих АТД. Каждая структура данных строится на основе базовых типов данных применяемого языка программирования, используя доступные в этом языке средства структурирования данных. Структуры массивов и записей – два важных средства структурирования данных, возможных в языке *Pascal*. Например, одной из возможных реализации переменной *S* типа *SET* может служить массив, содержащий элементы множества S.

Одной из основных причин определения двух различных АТД в рамках одной модели является то, что над объектами этих АТД необходимо выполнять различные действия, т. е. определять операторы разных типов.

В идеале желательно писать программы на языке, базовых типов данных и операторов которого достаточно для реализации АТД. С этой точки зрения язык *Pascal* не очень подходящий язык для реализации различных АТД, но, с другой стороны, трудно найти иной язык программирования, в котором можно было бы так непосредственно декларировать абстрактный тип данных.

1.5. Время выполнения программ

В процессе решения прикладных задач выбор подходящего алгоритма вызывает определенные трудности. В самом деле, на чем основывать свой выбор, если алгоритм должен удовлетворять следующим противоречащим друг другу требованиям:

1. быть простым для понимания, перевода в программный код и отладки;
2. эффективно использовать компьютерные ресурсы и выполняться по возможности быстро.

Если написанная программа должна выполняться только несколько раз, то первое требование наиболее важно. Стоимость рабочего времени программиста обычно значительно превышает стоимость машинного времени выполнения программы, поэтому стоимость программы оптимизируется по стоимости написания (а не выполнения) программы. Если мы имеем дело с задачей, решение которой требует значительных вычислительных затрат, то стоимость выполнения программы может превысить стоимость написания программы, особенно если программа должна выполняться многократно. Поэтому, с финансовой точки зрения, более предпочтительным может стать сложный комплексный алгоритм (в надежде, что результирующая программа будет выполняться существенно быстрее, чем более простая программа). Но и в этой ситуации разумнее сначала реализовать простой алгоритм, чтобы определить, как должна себя вести более сложная программа.

При построении сложной программной системы желательно реализовать ее простой прототип, на котором можно провести необходимые измерения и смоделировать ее поведение в целом, прежде чем приступать к разработке окончательного варианта. Таким образом, программисты должны быть осведомлены не только о методах построения быстрых программ, но и знать, когда их следует применить (желательно с минимальными усилиями).

Измерение времени выполнения программ

На время выполнения программы влияют следующие факторы:

1. Ввод исходной информации в программу.
2. Качество скомпилированного кода исполняемой программы.
3. Машинные инструкции (естественные и ускоряющие), используемые для выполнения программы.
4. Временная сложность алгоритма соответствующей программы[1].

Поскольку время выполнения программы зависит от ввода исходных данных, его можно определить как функцию от исходных данных. Но зачастую время выполнения программы зависит не от самих исходных данных, а от их «размера». В этом отношении хорошим примером являются задачи сортировки, которые мы подробно рассмотрим в главе 3. В задачах сортировки в качестве входных данных выступает список элементов, подлежащих сортировке, а в качестве выходного результата – те же самые элементы, отсортированные в порядке возрастания или убывания. Например, входной список 2, 1, 3, 1, 5, 8 будет преобразован в выходной список 1, 1, 2, 3, 5, 8 (в данном случае список отсортирован в порядке возрастания). Естественной мерой объема входной информации для программы сортировки будет число элементов, подлежащих сортировке, или, другими словами, длина входного списка. В общем случае длина входных данных – подходящая мера объема входной информации, и если не будет оговорено иное, то в качестве меры объема входной информации будем далее понимать именно длину входных данных.

Обычно говорят, что время выполнения программы имеет порядок $T(n)$ от входных данных размера n. Например, некая программа имеет время выполнения $T(n) = cn^2$, где c – константа. Единица измерения $T(n)$ точно не определена, но мы будем понимать $T(n)$ как количество инструкций, выполняемых на идеализированном компьютере.

Для многих программ время выполнения действительно является функцией входных данных, а не их размера. В этой ситуации определяем $T(n)$ как время выполнения в наихудшем случае, т. е. как максимум времени вы-

[1] Под термином «временная сложность алгоритма» понимается «время» выполнения алгоритма, измеряемое в «шагах» (инструкциях алгоритма), которые необходимо выполнить алгоритму для достижения запланированного результата. В качестве синонима для этого термина часто используют выражение «время выполнения алгоритма». В этой связи необходимо различать «время выполнения алгоритма» и «время выполнения программы».

полнения по всем входным данным размера n. Также будем рассматривать $T_{ср}(n)$ как среднее (в статистическом смысле) время выполнения по всем входным данным размера n. Хотя $T_{ср}(n)$ является достаточно объективной мерой времени выполнения, но часто нельзя предполагать (или обосновать) равнозначность всех входных данных. На практике среднее время выполнения найти сложнее, чем наихудшее время выполнения, так как математически это трудноразрешимая задача и, кроме того, зачастую не имеет простого определения понятие «средних» входных данных. Поэтому в основном будем использовать наихудшее время выполнения как меру временной сложности алгоритмов, но не будем забывать и о среднем времени выполнения там, где это возможно.

Теперь сделаем замечание о втором и третьем факторах, влияющих на время выполнения программ: о компиляторе, используемом для компиляции программы, и машине, на которой выполняется программа. Эти факторы влияют на то, что для измерения времени выполнения $T(n)$ нельзя применить стандартные единицы измерения, такие как секунды или миллисекунды. Поэтому можно только делать заключения, подобные «время выполнения такого-то алгоритма пропорционально n^2». Константы пропорциональности также нельзя точно определить, поскольку они зависят от компилятора, компьютера и других факторов.

Асимптотические соотношения

Для описания скорости роста функций используется O-символика. Например, когда мы говорим, что время выполнения $T(n)$ некоторой программы имеет порядок $O(n^2)$ (читается «о большое от n в квадрате» или просто «о от n в квадрате»), то подразумевается, что существуют положительные константы c и n_0 такие, что для всех n, больших или равных n_0, выполняется неравенство $T(n) \le cn^2$.

Например, предположим, что $T(0) = 1$, $T(1) = 4$ и в общем случае $T(n) = (n + 1)^2$. Тогда $T(n)$ имеет порядок $O(n^2)$: если положить $n_0 = 1$ и $c = 4$, то легко показать, что для $n \ge 1$ будет выполняться неравенство $(n + 1)^2 \le 4n^2$. Отметим, что нельзя положить $n_0 = 0$, так как $T(0) = 1$ и, следовательно, это значение при любой константе c больше $c0^2 = 0$.

Подчеркнем: мы предполагаем, что все функции времени выполнения определены на множестве неотрицательных целых чисел и их значения также неотрицательны, но необязательно целые. Будем говорить, что $T(n)$ имеет порядок $O(f(n))$, если существуют константы c и n_0 такие, что для всех $n \ge n_0$ выполняется неравенство $T(n) \le cf(n)$. Для программ, у которых время выполнения имеет порядок $O(f(n))$, говорят, что они имеют порядок (или степень) роста $f(n)$.

Рассмотрим еще один пример. Функция $T(n) = 3n^3 + 2n^2$ имеет степень роста $O(n^3)$. Чтобы это показать, надо положить $n_0 = 0$ и $c = 5$, так как легко видеть, что для всех целых $n \ge 0$ выполняется неравенство $3n^3 + 2n^2 \le 5n^3$.

Можно, конечно, сказать, что $T(n)$ имеет порядок $O(n^4)$, но это более слабое утверждение, чем то, что $T(n)$ имеет порядок роста $O(n^3)$.

В качестве следующего примера докажем, что функция $3n$ не может иметь порядок $O(2n)$. Предположим, что существуют константы c и n_0 такие, что для всех $n \geq n_0$ выполняется неравенство $3^n \leq c2^n$. Тогда $c \geq (3 / 2)^n$ для всех $n \geq n_0$. Но $(3 / 2)^n$ принимает любое, как угодно большое, значение при достаточно большом n, поэтому не существует такой константы c, которая могла бы мажорировать $(3 / 2)^n$ для всех n.

Когда мы говорим, что $T(n)$ имеет степень роста $O(f(n))$, то подразумевается, что $f(n)$ является верхней границей скорости роста $T(n)$. Чтобы указать нижнюю границу скорости роста $T(n)$, используется обозначение: $T(n)$ есть $\Omega(g(n))$ (читается «омега-большое от $g(n)$» или просто «омега от $g(n)$»), это подразумевает существование такой константы c, что бесконечно часто (для бесконечного числа значений n) выполняется неравенство $T(n) \geq cg(n)$.[1]

Например, для проверки того, что $T(n) = n^3 + 2n^2$ есть $\Omega(n^3)$, достаточно положить $c = 1$. Тогда $T(n) \geq cn^3$ для $n = 0, 1, \ldots$

Для другого примера положим, что $T(n) = n$ для нечетных $n > 1$ и $T(n) = n / 100$ – для четных $n \geq 0$. Для доказательства того, что $T(n)$ есть $\Omega(n^2)$, достаточно положить $c = 1 / 100$ и рассмотреть множество четных чисел $n = 0, 2, 4, 6, \ldots$

Ограниченность показателя степени роста

Итак, мы предполагаем, что программы можно оценить с помощью функций времени выполнения, пренебрегая при этом константами пропорциональности. С этой точки зрения программа с временем выполнения $O(n^2)$, например, лучше программы с временем выполнения $O(n^3)$. Константы пропорциональности зависят не только от используемых компилятора и компьютера, но и от свойств самой программы. Пусть при определенной комбинации компилятор-компьютер одна программа выполняется за $100n^2$ миллисекунд, а вторая – за $5n^3$ миллисекунд. Может ли вторая программа быть предпочтительнее, чем первая?

Ответ на этот вопрос зависит от размера входных данных программ. При размере входных данных $n < 20$ программа с временем выполнения $5n^3$ завершится быстрее, чем программа с временем выполнения $100n^2$. Поэтому, если программы в основном выполняются с входными данными небольшого размера, предпочтение необходимо отдать программе с временем выполнения $O(n^3)$. Однако при возрастании n отношение времени выполнения $5n^3 / 100\ n^2 = n / 20$ также растет. Поэтому при больших n программа с време-

[1] Отметим асимметрию в определениях O- и Ω-символики. Такая асимметрия бывает полезной, когда алгоритм работает быстро на достаточно большом подмножестве, но не на всем множестве входных данных. Например, есть алгоритмы, которые работают значительно быстрее, если длина входных данных является простым числом, а не (к примеру) четным числом. В этом случае невозможно получить хорошую нижнюю границу времени выполнения, справедливую для всех $n \geq n_0$.

нем выполнения $O(n^2)$ становится предпочтительнее программы с временем выполнения $O(n^3)$. Если даже при сравнительно небольших n, когда время выполнения обеих программ примерно одинаково, выбор лучшей программы представляет определенные затруднения, то естественно для большей надежности сделать выбор в пользу программы с меньшей степенью роста.

Другая причина, заставляющая отдавать предпочтение программам с наименьшей степенью роста времени выполнения, заключается в том, что чем меньше степень роста, тем больше размер задачи, которую можно решить на компьютере. Другими словами, если увеличивается скорость вычислений компьютера, то растет также и размер задач, решаемых на компьютере. Однако незначительное увеличение скорости вычислений компьютера приводит только к небольшому увеличению размера задач, решаемых в течение фиксированного промежутка времени, исключением из этого правила являются программы с низкой степенью роста, как $O(n)$ и $O(n \log n)$.

На рис. 1.1 показаны функции времени выполнения (измеренные в секундах) для четырех программ с различной временной сложностью для одного и того же сочетания компилятор-компьютер.

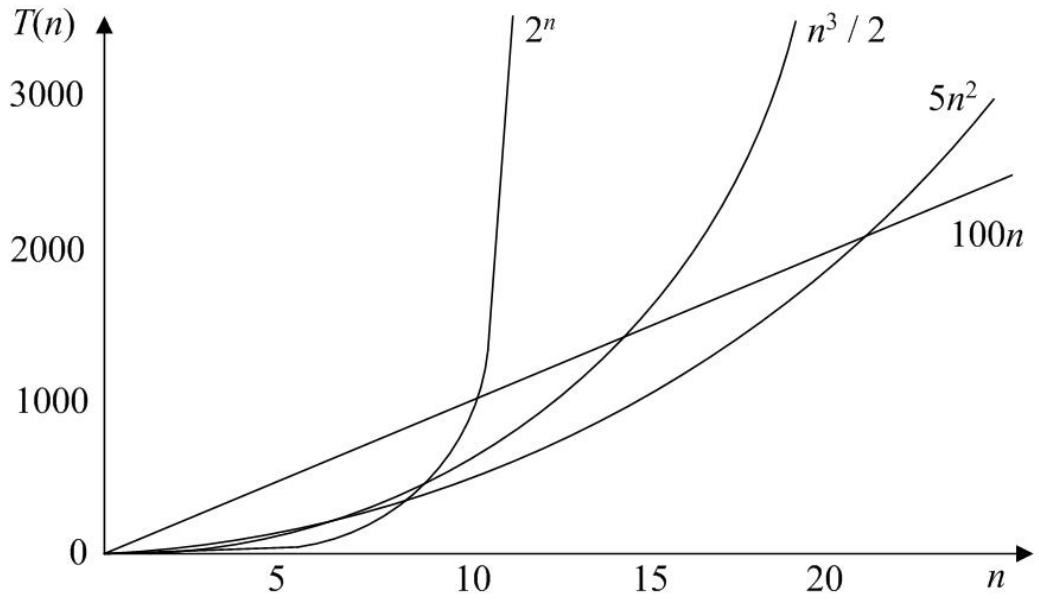


Рис. 1.1. Функции времени выполнения четырех программ

Предположим, что можно использовать 1 000 секунд (примерно 17 минут) машинного времени для решения задачи. Какой максимальный размер задачи, решаемой за это время? За 10 секунд каждый из четырех алгоритмов может решить задачи примерно одинакового размера, как показано во втором столбце табл. 1.1.

Предположим, что получен новый компьютер (без дополнительных финансовых затрат), работающий в десять раз быстрее. Теперь за ту же цену можно использовать 10^4 секунд машинного времени – ранее 10^3 секунд. Максимальный размер задачи, которую может решить за это время каждая из

четырех программ, показан в третьем столбце табл. 1.1. Отношения значений третьего и второго столбцов приведены в четвертом столбце этой таблицы. Здесь мы видим, что увеличение скорости компьютера на 1 000% приводит к увеличению только на 30% размера задачи, решаемой с помощью программы с временем выполнения $O(2^n)$. Таким образом, 10-кратное увеличение производительности компьютера дает в процентном отношении значительно меньший эффект увеличения размера решаемой задачи. В действительности, независимо от быстродействия компьютера, программа с временем выполнения $O(2^n)$ может решать только очень небольшие задачи.

Таблица 1.1

Эффект от 10-кратного увеличения быстродействия компьютера

Время выполнения $T(n)$	Максимальный размер задачи для 10^3 секунд	Максимальный размер задачи для 10^4 секунд	Увеличение максимального размера задачи
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3 / 2$	12	27	2.3
2^n	10	13	1.3

Из третьего столбца табл. 1.1 ясно видно преимущество программ с временем выполнения $O(n)$: 10-кратное увеличение размера решаемой задачи при 10-кратном увеличении производительности компьютера. Программы с временем выполнения $O(n^3)$ и $O(n^2)$ при увеличении быстродействия компьютера на 1 000% дают увеличение размера задачи соответственно на 230% и 320%. Эти соотношения сохранятся и при дальнейшем увеличении производительности компьютера.

Поскольку существует необходимость решения задач все более увеличивающегося размера, приходим к почти парадоксальному выводу. Так как машинное время все время дешевеет, а компьютеры становятся более быстродействующими, мы надеемся, что сможем решать все большие по размеру и более сложные задачи. Но вместе с тем возрастает значимость разработки и использования эффективных алгоритмов именно с низкой степенью роста функции времени выполнения.

Приведем несколько соображений, позволяющих посмотреть на критерий времени выполнения с других точек зрения.

1. Если создаваемая программа будет использована только несколько раз, тогда стоимость написания и отладки программы будет доминировать в общей стоимости программы, т. е. фактическое время выполнения не окажет существенного влияния на общую стоимость. В этом случае следует предпочесть алгоритм, наиболее простой для реализации.

2. Если программа будет работать только с «малыми» входными данными, то степень роста времени выполнения будет иметь меньшее значение, чем константа, присутствующая в формуле времени выполнения. Вместе с тем и понятие «малости» входных данных зависит от точного времени выполнения конкурирующих алгоритмов. Существуют алгоритмы, асимптотически самые эффективные, но которые никогда не используют на практике даже для больших задач, так как их константы пропорциональности значительно превосходят подобные константы других, более простых и менее «эффективных» алгоритмов.

3. Эффективные, но сложные алгоритмы могут быть нежелательными, если готовые программы будут поддерживать лица, не участвующие в написании этих программ. Будем надеяться, что принципиальные моменты технологии создания эффективных алгоритмов широко известны, и достаточно сложные алгоритмы свободно применяются на практике. Однако необходимо предусмотреть возможность того, что эффективные, но «хитрые» алгоритмы не будут востребованы из-за их сложности и трудностей, возникающих при попытке в них разобраться.

4. Известно несколько примеров, когда эффективные алгоритмы требуют таких больших объемов машинной памяти (без возможности использования более медленных внешних средств хранения), что этот фактор сводит на нет преимущество эффективности алгоритма.

5. В численных алгоритмах точность и устойчивость алгоритмов не менее важны, чем их временная эффективность.

Еще раз подчеркнем, что степень роста наихудшего времени выполнения – не единственный, но едва ли не самый важный критерий оценки алгоритмов и программ.

1.6. Вычисление времени выполнения программ

Теоретическое нахождение времени выполнения программ (даже без определения констант пропорциональности) – сложная математическая задача. Однако на практике определение времени выполнения (также без нахождения значения констант) является вполне разрешимой задачей. Для этого нужно знать только несколько базовых принципов. Но прежде чем представить эти принципы, рассмотрим, как выполняются операции сложения и умножения с использованием O-символики.

Пусть $T_1(n)$ и $T_2(n)$ – время выполнения двух программных фрагментов P_1 и P_2, $T_1(n)$ имеет степень роста $O(f(n))$, а $T_2(n) – O(g(n))$. Тогда $T_1(n) + T_2(n)$, т. е. время последовательного выполнения фрагментов P_1 и P_2, имеет степень роста $O(\max(f(n), g(n)))$. Для доказательства этого вспомним, что существуют константы c_1, c_2, n_1 и n_2 такие, что при $n \geq n_1$ выполняется неравенство $T_1(n) \leq c_1 f(n)$, и, аналогично, $T_2(n) \leq c_2 g(n)$, если $n \geq n_2$. Пусть $n_0 = \max(n_1, n_2)$.

Если $n \geq n_0$, то, очевидно, что $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$. Отсюда вытекает, что при $n \geq n_0$ справедливо неравенство $T_1(n) + T_2(n) \leq (c_1 + c_2) \max(f(n), g(n))$. Последнее неравенство и означает, что $T_1(n) + T_2(n)$ имеет порядок роста $O(\max(f(n), g(n)))$.

Правило сумм, данное выше, используется для вычисления времени последовательного выполнения программных фрагментов с циклами и ветвлениями. Предположим, что есть три фрагмента с временами выполнения соответственно $O(n^2)$, $O(n^3)$ и $O(n \log n)$. Тогда время последовательного выполнения первых двух фрагментов имеет порядок $O(\max(n^2, n^3))$, т. е. $O(n^3)$. Время выполнения всех трех фрагментов имеет порядок $O(\max(n^3, n \log n))$, это то же самое, что $O(\mathrm{n}^3)$.

В общем случае время выполнения конечной последовательности программных фрагментов, без учета констант, имеет порядок фрагмента с наибольшим временем выполнения. Иногда возможна ситуация, когда порядки роста времен нескольких фрагментов несоизмеримы (ни один из них не больше, чем другой, но они и не равны). Для примера рассмотрим два фрагмента с временем выполнения $O(f(n))$ и $O(g(n))$, где

$$f(n) = \begin{cases} n^4, \text{ если } n \text{ четное;} \\ n^2, \text{ если } n \text{ нечетное,} \end{cases}$$

$$g(n) = \begin{cases} n^2, \text{ если } n \text{ четное;} \\ n^3, \text{ если } n \text{ нечетное.} \end{cases}$$

В данном случае правило сумм можно применить непосредственно и получить время выполнения $O(\max(f(n), g(n))$, т. е. n^4 при n четном и n^3, если n нечетно.

Из правила сумм также следует, что если $g(n) \leq f(n)$ для всех n, превышающих n_0, то выражение $O(f(n) + g(n))$ эквивалентно $O(f(n))$. Например, $O(n^2 + n)$ то же самое, что $O(n^2)$.

Правило произведений заключается в следующем. Если $T_1(n)$ и $T_2(n)$ имеют степени роста $O(f(n))$ и $O(g(n))$ соответственно, то произведение $T_1(n)$ x $T_2(n)$ имеет степень роста $O(f(n)$ x $g(n))$. Из правила произведений следует, что $O(cf(n))$ эквивалентно $O(f(n))$, если c – положительная константа. Например, $O(n^2 / 2)$ эквивалентно $O(n^2)$.

Прежде чем переходить к общим правилам анализа времени выполнения программ, рассмотрим простой пример, иллюстрирующий процесс определения времени выполнения.

Рассмотрим программу сортировки *bubble* (пузырек), которая упорядочивает массив целых чисел в возрастающем порядке методом «пузырька». За каждый проход внутреннего цикла (операторы (3)–(6)) «пузырек» с наименьшим элементом «всплывает» в начало массива.

```
procedure bubble (var A: array[1..n] of integer);
      {Процедура упорядочивает массив A в возрастающем порядке}
var
      i, j, temp: integer;
begin
(1)   for i:= 1 to n – 1 do
(2)      for j:= n downto i + 1 do
(3)         if A[j – 1] > A[j] then begin
                        {перестановка местами A[j – 1] и A[j]}
(4)                     temp := A[j – 1] ;
(5)                      A[j – 1] := A[j];
(6)                      A[j] := temp;
            end
end;
```

Число элементов n, подлежащих сортировке, может служить мерой объема входных данных. Сначала отметим, что все операторы присваивания имеют некоторое постоянное время выполнения, независящее от размера входных данных. Таким образом, операторы (4) – (6) имеют время выполнения порядка $O(1)$. Запись $O(1)$ означает «равнозначно некой константе». В соответствии с правилом сумм время выполнения этой группы операторов равно $O(\max(1, 1, 1)) = O(1)$.

Теперь необходимо подсчитать время выполнения условных и циклических операторов. Операторы *if* и *for* вложены друг в друга, поэтому мы пойдем от внутренних операторов к внешним, последовательно определяя время выполнения условного оператора и каждой итерации цикла. Для оператора *if* проверка логического выражения занимает время порядка $O(1)$. Мы не знаем, будут ли выполняться операторы в теле условного оператора (строки (4) – (6)), но поскольку мы ищем наихудшее время выполнения, то, естественно, предполагаем, что они выполняются. Таким образом, получаем, что время выполнения группы операторов (3) – (6) имеет порядок $O(1)$.

Далее рассмотрим группу (2) – (6) операторов внутреннего цикла. Общее правило вычисления времени выполнения цикла заключается в суммировании времени выполнения каждой итерации цикла. Для операторов (2) – (6) время выполнения на каждой итерации имеет порядок $O(1)$. Цикл выполняется $n - i$ раз, поэтому по правилу произведений общее время выполнения цикла имеет порядок $O((n – i) \times 1)$, что равно $O(n – i)$.

Теперь перейдем к внешнему циклу, который содержит все исполняемые операторы программы. Оператор (1) выполняется $n - 1$ раз, поэтому суммарное время выполнения программы ограничено сверху выражением

$$\sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = n^2/2 - n/2,$$

которое имеет порядок $O(n^2)$. Таким образом, программа «пузырька» выполняется за время, пропорциональное квадрату числа элементов, подлежащих упорядочиванию. В главе 3 будут рассмотрены программы с временем выполнения порядка $O(n \log n)$, которое существенно меньше $O(n^2)$, поскольку при больших n $\log n$[1] значительно меньше n.

Перед формулировкой общих правил анализа программ позвольте напомнить, что нахождение точной верхней границы времени выполнения программ только в редких случаях так же просто, как в приведенном выше примере, в общем случае эта задача является интеллектуальным вызовом исследователю. Поэтому не существует исчерпывающего множества правил анализа программ. Можно дать только некоторые советы и проиллюстрировать их с разных точек зрения примерами, приведенными в этом пособии.

Дадим несколько правил анализа программ. В общем случае время выполнения оператора или группы операторов можно параметризовать с помощью размера входных данных и/или одной или нескольких переменных. Но для времени выполнения программы в целом допустимым параметром может быть только n, размер входных данных.

1. Время выполнения операторов присваивания, чтения и записи обычно имеет порядок $O(1)$. Есть несколько исключений из этого правила, например в языке *PL / 1*, где можно присваивать большие массивы, или в любых других языках, допускающих вызовы функций в операторах присваивания.

2. Время выполнения последовательности операторов определяется с помощью правила сумм. Поэтому степень роста времени выполнения последовательности операторов без определения констант пропорциональности совпадает с наибольшим временем выполнения оператора в данной последовательности.

3. Время выполнения условных операторов состоит из времени выполнения условно исполняемых операторов и времени вычисления самого логического выражения. Время вычисления логического выражения обычно имеет порядок $O(1)$. Время для всей конструкции *if-then-else* состоит из времени вычисления логического выражения и наибольшего из времени, необходимого для выполнения операторов, исполняемых при значении логического выражения *true* (истина) и при значении *false* (ложь).

4. Время выполнения цикла является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла (обычно последнее имеет порядок $O(1)$). Часто время выполнения цикла вычисляется, пренебрегая определением констант пропорциональности, как произведение количества выполненных итераций цикла на наибольшее возможное время выполнения операторов тела цикла. Время выполнения каждого цикла, если в программе их несколько, должно определяться отдельно.

[1] Если не указано другое, то будем считать, что все логарифмы определены по основанию 2. Однако отметим, что выражение $O(\log n)$ не зависит от основания логарифма, поскольку $\log_a n = c \log_b n$, где $c = \log_a b$.

Вызовы процедур

Для программ, содержащих несколько процедур (среди которых нет рекурсивных), можно подсчитать общее время выполнения программы путем последовательного нахождения времени выполнения процедур, начиная с той, которая не имеет вызовов других процедур. (Вызов процедур определяем по наличию оператора *call*.) Так как мы предположили, что все процедуры нерекурсивные, то должна существовать хотя бы одна процедура, не имеющая вызовов других процедур. Затем можно определить время выполнения процедур, вызывающих эту процедуру, используя уже вычисленное время выполнения вызываемой процедуры. Продолжая этот процесс, найдем время выполнения всех процедур и, наконец, время выполнения всей программы.

Если есть рекурсивные процедуры, то нельзя упорядочить все процедуры таким образом, чтобы каждая процедура вызывала только процедуры, время выполнения которых подсчитано на предыдущем шаге. В этом случае мы должны с каждой рекурсивной процедурой связать временную функцию $T(n)$, где n определяет объем аргументов процедуры. Затем необходимо получить рекуррентное соотношение для $T(n)$, т. е. уравнение (или неравенство) для $T(n)$, где участвуют значения $T(k)$ для различных значений k.

Техника решения рекуррентных соотношений зависит от вида этих соотношений. Проанализируем простую рекурсивную программу. Ниже представлен листинг программы вычисления факториала $n!$, т. е. вычисления произведения целых чисел от 1 до n включительно.

```
function fact (n: integer): integer;
        { fact(n) вычисляет n! }
begin
(1)     if n <= 1 then
(2)             fact := 1
        else
(3)             fact := n * fact(n – 1)
end;
```

Естественной мерой объема входных данных для функции *fact* является значение n. Обозначим через $T(n)$ время выполнения программы. Время выполнения для строк (1) и (2) имеет порядок $O(1)$, а для строки (3) – $O(1) + T(n – 1)$. Таким образом, для некоторых констант c и d имеем

$$T(n) = \begin{cases} c + T(n-1), & \text{если } n > 1; \\ d, & \text{если } n \le 1. \end{cases} \qquad (1.1)$$

Полагая, что $n > 2$, и раскрывая в соответствии с соотношением (1.1) выражение $T(n-1)$ (т. е. подставляя в (1.1) $n-1$ вместо n), получим $T(n) = 2c + T(n-2)$. Аналогично, если $n > 3$, раскрывая $T(n-2)$, получим $T(n) = 3c + T(n-3)$. Продолжая этот процесс, в общем случае для некоторого i, $n > i$, имеем $T(n) = ic + T(n-i)$. Положив в последнем выражении $i = n-1$, окончательно получаем

$$T(n) = c(n-1) + T(1) = c(n-1) + d. \qquad (1.2)$$

Из (1.2) следует, что $T(n)$ имеет порядок $O(n)$. Отметим, что в этом примере анализа программы мы предполагали, что операция перемножения двух целых чисел имеет порядок $O(1)$. На практике, однако, программу, представленную выше, нельзя использовать для вычисления факториала при больших значениях n, так как размер получаемых в процессе вычисления целых чисел может превышать длину машинного слова.

Общий метод решения рекуррентных соотношений, подобных соотношению из рассмотренного примера, состоит в последовательном раскрытии выражений $T(k)$ в правой части уравнения (путем подстановки в исходное соотношение k вместо n) до тех пор, пока не получится формула, у которой в правой части отсутствует T (как в формуле (1.2)). При этом часто приходится находить суммы различных последовательностей. Если значения таких сумм нельзя вычислить точно, то для сумм находятся верхние границы, что позволяет, в свою очередь, получить верхние границы для $T(n)$.

Программы с операторами безусловного перехода

При анализе времени выполнения программ мы неявно предполагали, что все ветвления в ходе выполнения процедур осуществлялись с помощью условных операторов и операторов циклов. Мы останавливаемся на этом факте, так как определяли время выполнения больших групп операторов путем применения правила сумм к этим группам. Однако операторы безусловного перехода (такие как *goto*) могут порождать более сложную логическую групповую структуру. В принципе, операторы безусловного перехода можно исключить из программы. Но, к сожалению, язык *Pascal* не имеет операторов досрочного прекращения циклов, поэтому операторы перехода по необходимости часто используются в подобных ситуациях.

Предлагается следующий подход для работы с операторами безусловного перехода, выполняющих выход из циклов, который гарантирует отслеживание всего цикла. Поскольку досрочный выход из цикла, скорее всего, осуществляется после выполнения какого-нибудь логического условия, то можно предположить, что это условие никогда не выполняется. Но этот консервативный подход никогда не позволит уменьшить время выполнения программы, так как мы предполагаем полное выполнение цикла. Для программ, где досрочный выход из цикла не исключение, а правило, консервативный

подход значительно переоценивает степень роста времени выполнения в наихудшем случае. Если же переход осуществляется к ранее выполненным операторам, то в этом случае вообще нельзя игнорировать оператор безусловного перехода, поскольку такой оператор создает петлю в выполнении программы, что приводит к нарастанию времени выполнения всей программы.

Тем не менее, анализ времени выполнения программ, использующих операторы безусловного перехода, создающих петли, возможен. Если программные петли имеют простую структуру, т. е. о любой паре петель можно сказать, что они или не имеют общих элементов, или одна петля вложена в другую, то в этом случае можно применить методы анализа времени выполнения, описанные в данном разделе.

Анализ программ на псевдоязыке

Если известна степень роста времени выполнения операторов, записанных с помощью неформального «человеческого» языка, то, следовательно, можно проанализировать программу на псевдоязыке (такие программы будем называть псевдопрограммами). Однако часто не возможно в принципе знать время выполнения той части псевдопрограммы, которая еще не полностью реализована в формальных операторах языка программирования. Например, если в псевдопрограмме имеется неформальная часть, оперирующая абстрактными типами данных, то общее время выполнение программы в значительной степени зависит от способа реализации АТД. Это не является недостатком псевдопрограмм, так как одной из причин написания программ в терминах АТД является возможность выбора реализации АТД, в том числе по критерию времени выполнения.

При анализе псевдопрограмм, содержащих операторы языка программирования и вызовы процедур, имеющих неформализованные фрагменты (такие как операторы для работы с АТД), можно рассматривать общее время выполнения программы как функцию пока не определенного времени выполнения таких процедур. Время выполнения этих процедур можно параметризовать с помощью «размера» их аргументов. Часто выбранная математическая модель АТД сама подсказывает логически обоснованный размер аргументов. Например, если АТД строится на основе множеств, то часто количество элементов множества можно принять за параметр функции времени выполнения.

В завершении данной главы опишем в общем виде процесс создания программы на языке программирования, например на языке *Pascal*.

На рис. 1.2 схематически представлен процесс программирования. На первом этапе создается модель исходной задачи, для чего привлекаются соответствующие подходящие математические модели (например, теория графов) [1]. На этом этапе для нахождения решения также строится неформальный алгоритм.

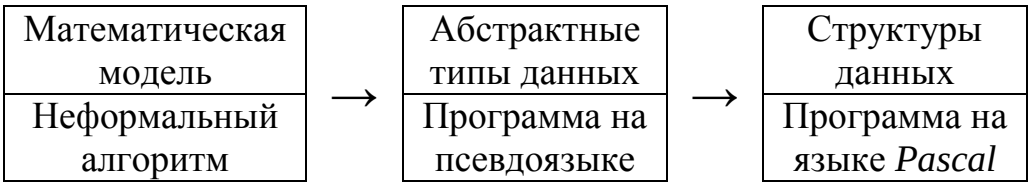


Рис. 1.2. Функции времени выполнения четырех программ

На следующем этапе алгоритм записывается на псевдоязыке – смеси конструкций языка *Pascal* и менее формальных и обобщенных операторов на простом «человеческом» языке. Продолжением этого этапа является замена неформальных операторов последовательностью более детальных и формальных операторов. С этой точки зрения программа на псевдоязыке должна быть достаточно подробной, так как в ней фиксируются (определяются) различные типы данных, над которыми выполняются операторы. Затем создаются абстрактные типы данных для каждого зафиксированного типа данных (за исключением элементарных типов данных, таких как целые и действительные числа или символьные строки) путем задания имен процедурам для каждого оператора, выполняемого над данными абстрактного типа, и замены операторов вызовом соответствующих процедур.

Третий этап процесса программирования обеспечивает реализацию каждого абстрактного типа данных и создание процедур для выполнения различных операторов над данными этих типов. На этом этапе также заменяются все неформальные операторы псевдоязыка на код языка *Pascal*. Результатом этого этапа должна быть выполняемая программа. После ее отладки получается работающую программу. Предполагается что, используя пошаговый подход при разработке программ, схема которого показана на рис. 1.2, процесс отладки конечной программы будет небольшим и безболезненным.

2. ПОИСК ОБРАЗА В СТРОКЕ

Одно из наиболее часто встречающихся в программировании действий – поиск. Он же представляет собой идеальную задачу, на которой можно испытывать различные структуры данных по мере их появления. Существует несколько основных «вариаций этой темы», и для них создано много различных алгоритмов. В данной главе будет рассмотрен специфический поиск, так называемый поиск строки. Его можно определить следующим образом. Пусть задан массив *str* из *N* элементов и массив *img* из *M* элементов, причем $0 \le M < N$. Описаны они так:

char str[*N*];

char img[*M*];

Поиск строки обнаруживает первое вхождение *img* в *str*. Оба массива содержат символы, так что *str* можно считать некоторым текстом или строкой, а *img* – образом, первое вхождение в строку которого необходимо найти. Это действие типично для любых систем обработки текстов, отсюда и очевидная заинтересованность в поиске эффективного алгоритма для этой задачи.

2.1. Прямой поиск строки

Прежде чем обратить внимание на эффективность, разберем некий «прямолинейный» алгоритм поиска, который называется прямым поиском строки. При прямом поиске строки сравнивается первый символ образа и соответствующий ему символ из строки. В начале работы алгоритма этим символом будет первый символ строки. Если сравниваемые символы совпадают, то рассматриваются следующие, вторые, символы образа и строки. Таким образом происходит перемещение по образу от его начала к концу. Если образ закончился, значит, все символы образа совпали с символами рассматриваемой части строки, т. е. поиск образа в строке выполнен успешно.

В том случае, если произошло несовпадение символов образа и строки, образ сдвигается по строке на один символ вправо, и снова происходит сравнение образа, начиная с первого символа. Теперь уже первый символ образа сравнивается со вторым символом строки, второй символ образа – с третьим символом строки и т. д. Поиск происходит до тех пор, пока не будет найдено вхождение образа в строке или же не закончится строка, что значит, что вхождение образа не было найдено. На рис. 2.1 приведен пример прямого поиска строки, сравниваемые символы образа подчеркнуты.

```
Hoola-Hoola girls like Hooligans.
Hooligan
 Hooligan
  Hooligan
       …
                    Hooligan
```

Рис. 2.1. Прямой поиск образа в строке

При программной реализации можно использовать два цикла. Внешний цикл отвечает за продвижение образа по строке, во внутреннем цикле происходит продвижение по образу. Листинг данной программы выглядит следующим образом:

```
i = –1;
do{
  i++;
  j = 0;
  while((j < m) && (str[i + j] == img[j]))
    j++;
}while((j != m) && (i < n – m));
```

В действительности член ($i < n – m$) в условии окончания определяет, не закончилась ли строка и в случае отрицательного ответа свидетельствует, что нигде в строке совпадения с образом не произошло. В то же время невыполнение условия (j != m) соответствует ситуации, когда соответствие образа некоторой части строки найдено.

Эффективность прямого поиска в строке. Данный алгоритм работает достаточно эффективно, если допустить, что несовпадение пары символов происходит, по крайней мере, после всего лишь нескольких сравнений во внутреннем цикле. Можно предполагать, что для текстов, составленных из 128 печатных символов, несовпадение будет обнаруживаться после одной или двух проверок. Тем не менее, в худшем случае производительность будет внушать опасение. Например, для строки, состоящей из $N – 1$ символов A и единственного B, а образа, содержащего $M – 1$ символов A и опять B, чтобы обнаружить совпадение в конце строки, требуется произвести порядка $N * M$ сравнений. Далее рассматриваются два алгоритма, которые намного эффективнее решают задачу поиска образа в строке.

2.2. Алгоритм Кнута, Морриса и Пратта

В 1970 г. Д. Кнут, Д. Моррис и В. Пратт изобрели алгоритм, фактически требующий только N сравнений даже в самом плохом случае. Новый ал-

горитм основывается на том соображении, что, начиная каждый раз сравнение образа с самого начала, после частичного совпадения начальной части образа с соответствующими символами строки пройденная часть строки фактически известна и можно «вычислить» некоторые сведения (на основе самого образа), с помощью которых можно продвинуться по строке дальше, чем при прямом поиске.

Алгоритм Кнута, Морриса и Пратта или КМП-алгоритм использует при сдвиге образа таблицу *d*, которая формируется еще до начала поиска образа в строке. Можно выделить два этапа работы КМП-алгоритма:

1. формирование таблицы *d*, используемой при сдвиге образа по строке;
2. поиск образа в строке.

Таблица *d* формируется на основе образа и содержит значения, которые в дальнейшем будут использованы при вычислении величины сдвига образа. Размер данной таблицы равен длине образа, которую можно определить при помощи функции *int strlen*(*char* *) библиотеки <*string.h*>. Таким образом, таблица *d* фактически является одномерным массивом, состоящим из числа элементов равного количеству символов в образе.

Первый элемент массива *d* всегда равен –1. Все элементы массива *d* соответствующие символам одинаковым первому символу образа также приравниваются –1. Как показано на рис. 2.2, элемент *d*[0], соответствующий первому символу образа – *a* – равен –1, также и четвертый элемент *d*[3], также соответствующий символу *a* равен –1.

образ:	a b c a b c
значения элементов массива *d*:	–1 –1

Рис. 2.2. Образ и значения некоторых элементов массива *d*

Для остальных символов образа значение элементов массива *d* вычисляется следующим образом: значение *d*[*j*], соответствующее *j*-му символу образа, равно максимальному числу символов непосредственно предшествующих данному символу, совпадающих с началом образа. При этом если рассматриваемому символу предшествует *k* символов, то во внимание принимаются только *k*–1 предшествующих символов.

образ:	a b c a b c
значения элементов массива *d*:	–1 0 0 –1 1 2

Рис. 2.3. Образ и значения элементов массива *d*

На рис. 2.3 показаны значения элементов массива *d*. Отметим, что пятому символу образа, символу *b*, предшествует символ *a*, совпадающий с первым символом образа, поэтому *d*[4], соответствующий символу *b*, равен 1. Аналогично, *d*[5] равен 2, поскольку символу *c* предшествуют два символа *a* и *b*, совпадающие с двумя первыми символами образа.

Поскольку $d[j]$ принимает значение равное максимальному числу символов предшествующих j-му символу, то, как показано на рис. 2.4, элемент $d[6]$, соответствующий символу c, равен четырем, а не двум.

образ:		a	b	a	b	a	b	c
значения элементов массива d:							4

Рис. 2.4. Образ и значения элемента $d[6]$

Можно сделать следующий вывод: значения массива d определяется одним лишь образом и не зависит от строки текста. Для определения значения элементов массива d необходимо найти самую длинную последовательность символов образа, непосредственно предшествующих позиции j, которая совпадает полностью с началом образа. Так как эти значения зависят только от образа, то перед началом фактического поиска необходимо вычислить элементы массива d. Эти вычисления будут оправданными, если размер строки или текста значительно превышает размер образа.

Вторым этапом работы КМП-алгоритма является сравнение символов образа и строки и вычисления сдвига образа в случае их несовпадения. Символы образа рассматриваются слева направо, т. е. от начала к концу образа. При несовпадении символов образа и строки образ сдвигается вправо по строке. Величина сдвига вычисляется следующим образом: если при переборе символов образа используется индекс j, то сдвиг образа происходит на $j - d[j]$ символов. На рис. 2.5 приведен ряд примеров иллюстрирующих сдвиг образа при работе КМП-алгоритма.

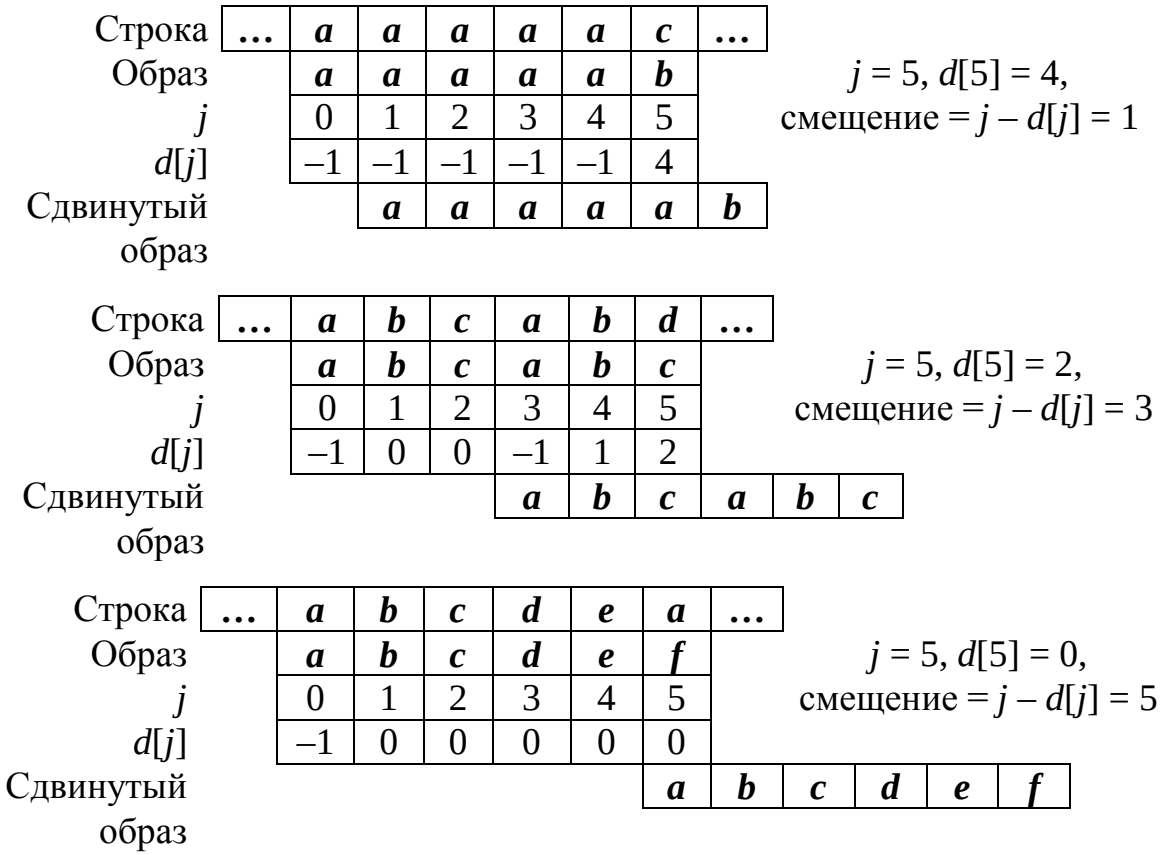


Рис. 2.5. Частичное совпадение и смещение образа

Приведем пример поиска в строке образа «*Hooligan*». На рис. 2.6 приведены значения элементов массива *d*. На рис. 2.7 показан принцип работы КМП-алгоритма, сравниваемые символы подчеркнуты.

образ: `Hooligan`
значения элементов массива *d*: `-10000000`

Рис. 2.6. Образ и значения элементов массива *d*

```
Hoola-Hoola girls like Hooligans.
Hooligan
    Hooligan
     Hooligan
      Hooligan
          Hooligan
            …
                        Hooligan
```

Рис. 2.7. Поиск образа в строке по методу Кнута, Морриса и Пратта

Обратите внимание: при каждом несовпадении символов образ сдвигается на все пройденное расстояние, поскольку меньшие сдвиги не могут привести к полному совпадению.

Эффективность КМП-алгоритма. Точный анализ КМП-поиска, как и сам его алгоритм, весьма сложен. Его разработчики доказывают, что требуется порядка $M + N$ сравнений символов, что значительно лучше, чем $M * N$ сравнений из прямого поиска. Они так же отмечают то приятное свойство, что указатель сканирования строки i никогда не возвращается назад, в то время как при прямом поиске после несовпадения просмотр всегда начинается с первого символа образа и поэтому может включать символы строки, которые уже просматривались ранее. Это может привести к затруднениям, если строка читается из вторичной памяти, ведь в этом случае возврат обходится дорого. Даже при буферизованном вводе может встретиться столь большой образ, что возврат превысит емкость буфера.

2.3. Алгоритм Боуера и Мура

Поиск образа в строке по методу Кнута, Морриса, Пратта дает выигрыш только в том случае, когда несовпадению символов из образа и строки предшествовало некоторое число совпадений. Если при сравнении образа и ана лизируемой части строки сопоставляемые символы различны, то образ сдвигается только на один символ. Продвижение образа более чем на единицу происходит лишь в этом случае, когда происходит совпадение нескольких символов образа и строки. К несчастью, это скорее исключение, чем правило: совпадения встречаются значительно реже, чем несовпадения. Поэтому выигрыш от использования КМП-стратегии в большинстве случаев поиска в обычных текстах весьма незначителен.

В 1975 г. Р. Боуер и Д. Мур предложили метод, который не только улучшает обработку самого плохого случая, но дает выигрыш в промежуточных ситуациях. Скорость в алгоритме Бойера-Мура достигается за счет того, что удается пропускать те части текста, которые заведомо не участвуют в успешном сопоставлении. Данный алгоритм, называемый БМ-поиском, основывается на необычном соображении – сравнение символов начинается с конца образа, а не с начала.

Как и при работе КМП-алгоритма, перед началом поиска образу сопоставляется таблица d, используемая в дальнейшем при смещении образа по строке. При создании матрицы d используется таблица кодов символов ASCII. Любой печатный или служебный символ имеет свой код. Например, код символа n – 110, g – 103, код пробела – 32. Коды ASCII печатных символов приведены в табл. 2.1. Таблица ASCII содержит 256 символов. Поэтому матрицу d можно объявить, как одномерный целочисленный массив, состоящий из 256 элементов: *int d*[256].

Таблица 2.1

Таблица ASCII. Печатные символы

Код	Символ	Код	Символ	Код	Символ	Код	Символ	Код	Символ	Код	Символ	Код	Символ
32	**пробел**	55	7	78	**N**	101	**e**	124	|	208	**Р**	231	**з**
33	**!**	56	**8**	79	**O**	102	**f**	125	**}**	209	**С**	232	**и**
34	"	57	**9**	80	**P**	103	**g**	126	~	210	**Т**	233	**й**
35	#	58	**:**	81	**Q**	104	**h**			211	**У**	234	**к**
36	**$**	59	**;**	82	**R**	105	**i**	168	**Ё**	212	**Ф**	235	**л**
37	**%**	60	<	83	**S**	106	**j**	184	**ё**	213	**Х**	236	**м**
38	**&**	61	=	84	**T**	107	**k**			214	**Ц**	237	**н**
39	'	62	>	85	**U**	108	**l**	192	**А**	215	**Ч**	238	**о**
40	**(**	63	**?**	86	**V**	109	**m**	193	**Б**	216	**Ш**	239	**п**
41	**)**	64	**@**	87	**W**	110	**n**	194	**В**	217	**Щ**	240	**р**
42	*****	65	**A**	88	**X**	111	**o**	195	**Г**	218	**Ъ**	241	**с**
43	+	66	**B**	89	**Y**	112	**p**	196	**Д**	219	**Ы**	242	**т**
44	**,**	67	**C**	90	**Z**	113	**q**	197	**Е**	220	**Ь**	243	**у**
45	-	68	**D**	91	**[**	114	**r**	198	**Ж**	221	**Э**	244	**ф**
46	**.**	69	**E**	92	\	115	**s**	199	**З**	222	**Ю**	245	**х**
47	/	70	**F**	93	**]**	116	**t**	200	**И**	223	**Я**	246	**ц**
48	**0**	71	**G**	94	^	117	**u**	201	**Й**	224	**а**	247	**ч**
49	**1**	72	**H**	95	_	118	**v**	202	**К**	225	**б**	248	**ш**
50	**2**	73	**I**	96	`	119	**w**	203	**Л**	226	**в**	251	**ы**
51	**3**	74	**J**	97	**a**	120	**x**	204	**М**	227	**г**	252	**ь**
52	**4**	75	**K**	98	**b**	121	**y**	205	**Н**	228	**д**	253	**э**
53	**5**	76	**L**	99	**c**	122	**z**	206	**О**	229	**е**	254	**ю**
54	**6**	77	**M**	100	**d**	123	**{**	207	**П**	230	**ж**	255	**я**

Первоначально всем элементам матрицы *d* присваивается значение равное длине образа. Длину образа можно получить, используя функцию *int strlen*(*char* *) из библиотеки <*string.h*>.

Следующим шагом является присвоение каждому элементу таблицы *d*, индекс которого равен коду ASCII текущего рассматриваемого символа образа, значения равного удаленности текущего символа от конца образа.

Рассмотрим формирование таблицы *d* на примере образа «*Hooligan*». Поскольку данный образ содержит 8 символов, то его длина равна 8. Соответственно, на начальном этапе все элементы массива *d* инициализируются числом 8:

$$d[0] = d[1] = \ldots = d[254] = d[255] = 8$$

Далее происходит присваивание элементам массива *d* соответствующих значений, равных расстоянию от рассматриваемого символа до конца образа. При этом индекс элемента массива *d*, который получает новое значение, определяется кодом ASCII рассматриваемого символа. Так, код ASCII

последнего символа образа «*Hooligan*» – *n* – равен 110. Поскольку данный символ является последним, то удаленность от конца образа равна нулю. Таким образом:

d[110] = 0;

также на языке программирования Си можно записать:

d[‘*n*’] = 0;

Код ASCII предпоследнего символа образа «*Hooligan*» – *a* – равен 97. Удаленность данного символа от конца образа равна 1 и следовательно:

d[97] = 1;

или

d[‘*a*’] = 1;

Аналогичным образом изменяются значения элементов таблицы *d*, соответствующие символам образа *g*, *i*, *l*, *o*:

d[‘*g*’] = 2; или *d*[103] = 2;
d[‘*i*’] = 3; или *d*[105] = 3;
d[‘*l*’] = 4; или *d*[108] = 4;
d[‘*o*’] = 5; или *d*[111] = 5;

В том случае, когда образ содержит несколько одинаковых символов, элементу таблицы *d*, соответствующему данному символу, присваивается значение равное удаленности от конца образа самого правого из одинаковых символов. Так, образ «*Hooligan*» содержит два символа *o*, удаленность от конца образа первого из них равно шести, удаленность второго – пять. В этом случае *d*[‘*o*’] будет равно пяти.

В рассматриваемом примере присваивание значений элементам массива *d* происходит при продвижении по образу справа налево. Таким образом, легко определить, был ли уже рассмотрен тот или иной символ, выполнив проверку на содержимое соответствующего элемента массива *d*: если элемент содержит значение равное длине образа, значит данный символ еще не рассматривался. Поэтому, когда будет рассматриваться второй символ образа «*Hooligan*», символ *o*, удаленность которого от конца образа равна шести, проверка содержимого соответствующего элемента массива *d* покажет, что *d*[‘*o*’] не равно длине образа – 8, что свидетельствует о том, что символ *o* уже встречался, и *d*[‘*o*’] изменяться не должно.

Следует отметить также, что элементу массива *d*, соответствующего символу *H*, должно быть присвоено значение семь, т. е. фактическая удаленность символа *H* от конца образа:

d[‘*H*’] = 7; или *d*[72] = 7;

На рис. 2.8 показан образ и значения элементов массива *d*, соответствующие символам данного образа.

образ:	`Hooligan`
значения элементов массива *d*:	`75543210`

Рис. 2.8. Образ и значения элементов массива *d*, соответствующие символам образа

Вторым этапом работы алгоритма БМ-поиска после построения таблицы *d* является собственно сам поиск образа в строке. При сравнении образа со строкой образ продвигается по строке слева направо, однако посимвольные сравнения образа и строки выполняются справа налево.

Сравнение образа со строкой происходит до тех пор, 1) пока не будет рассмотрен весь образ, что говорит о том, что соответствие между образом и некоторой частью строки найдено, 2) пока не закончится строка, что значит, что вхождений, соответствующих образу в строке нет, 3) либо пока не произойдет несовпадения символов образа и строки, что вызывает сдвиг образа на несколько символов вправо и продолжение процесса поиска.

В том случае, если произошло несовпадение символов, смещение образа по строке определяется значением элемента таблицы *d*, причем индексом данного элемента является код ASCII символа *строки*. Подчеркнем, что, несмотря на то, что массив *d* формируется на основе образа, при смещении индексом служит символ из строки. На рис. 2.9. показан пример работы алгоритма БМ-поиска, сравниваемые символы подчеркнуты.

```
Hoola-Hoola girls like Hooligans.
Hooligan
     Hooligan
       Hooligan
              Hooligan
                     Hooligan
```

Рис. 2.9. Поиск образа в строке по методу Боуера и Мура

На первой итерации произошло несовпадение символа образа *n* и символа строки *o*. Значение элемента *d*[‘*o*’] равно пяти. Поэтому образ смещается вправо на пять символов. Если бы произошло совпадение этих двух символов, то далее рассматривался бы предпоследний символ образа и соответствующий ему символ строки и т. д.

При очередном сравнивании образа и строки происходит несовпадение символов *n* и *g*. Опять же используя в качестве индекса элемента массива *d*

код ASCII символа строки *g*, получаем значение два: *d*[‘*g*’] = 2. Образ сдвигается на два символа вправо. Таким образом, образ постепенно сдвигается по строке до тех пор, пока образ в строке не будет найден или не кончится строка.

Эффективность БМ-алгоритма. Замечательным свойством БМ-поиска является то, что почти всегда, кроме специально построенных примеров, он требует значительно меньше *N* сравнений. В самых же благоприятных обстоятельствах, когда последний символ образа всегда попадает на несовпадающий символ строки, число сравнений равно *N* / *M*.

Авторы приводят и несколько соображений по поводу дальнейших усовершенствований алгоритма. Одно из них – объединить приведенную только что стратегию, обеспечивающую большие сдвиги в случае несовпадения, со стратегией Кнута, Морриса и Пратта, допускающей «ощутимые» сдвиги при обнаружении совпадения (частичного). Такой метод требует двух таблиц, получаемых при предтрансляции: d_1 – только что упомянутая таблица, а d_2 – таблица, соответствующая КМП-алгоритму. Из двух сдвигов выбирается больший, причем и тот и другой «говорят», что никакой меньший сдвиг не может привести к совпадению.

3. СОРТИРОВКА МАССИВОВ

В данной главе представлены принципиальные алгоритмы внутренней сортировки. Задача сортировки состоит в упорядочивании элементов в неубывающем (невозрастающем) порядке. Основным условием, предъявляемым к алгоритмам сортировки, является экономное использование доступной памяти. Это предполагает, что перестановки элементов с целью их упорядочивания должны выполняться *на том же месте,* т. е. методы, в которых элементы из массива *a* передаются в результирующий массив *b*, представляют существенно меньший интерес.

Эффективность алгоритмов сортировки массива можно оценить по таким критериям, как число необходимых сравнений элементов (*C*) и число перестановок элементов (*M*).

Эти числа фактически являются функциями от *n* – числа сортируемых элементов. Хотя хорошие алгоритмы сортировки требуют порядка $n * log(n)$ сравнений, рассмотрим несколько простых и очевидных методов, называемых *прямыми,* где требуется порядка n^2 сравнений элементов. Разбор прямых методов целесообразен ввиду следующих причин:

1. Прямые методы особенно удобны для объяснения характерных черт основных принципов большинства сортировок.
2. Программы этих методов легко понимать, и они коротки. Следует помнить, что сами программы также занимают память.
3. Усложненные методы требуют небольшого числа операций, но эти операции обычно сами более сложны, и поэтому для достаточно малых *n* прямые методы оказываются быстрее, хотя при больших *n* их использовать, конечно, не следует.

Методы сортировки *«на том же месте»* можно разбить в соответствии с определяющими их принципами на три основные категории:

- Сортировки с помощью включения (*by insertion*).
- Сортировки с помощью выбора (*by selection*).
- Сортировки с помощью обмена (*by exchange*).

Все рассматриваемые алгоритмы будут оперировать массивом *a*, в котором будут храниться переставляемые на месте элементы. Данный массив объявляется следующим образом:

int a[*n*];

3.1. Сортировка с помощью прямого включения

При сортировке элементов массива с помощью прямого включения массив делят на две части: отсортированную или «готовую» $a_1, \ldots, a_{i-1}$ и неотсортированную или «исходную» $a_i, \ldots, a_n$.

В начале работы алгоритмы в качестве отсортированной части массива принимается только один первый элемент, а в качестве неотсортированной части – все остальные элементы.

На каждом шаге, начиная с $i = 2$ и увеличивая i каждый раз на единицу, из неотсортированной последовательности извлекается i-й элемент и вставляется в отсортированную так, чтобы не нарушить в ней упорядоченности элементов. Каждый шаг алгоритма включает четыре действия:

1. Взять i-й элемент массива, он же является первым элементом в неотсортированной части, и сохранить его в дополнительной переменной.
2. Найти позицию j в отсортированной части массива, куда будет вставлен i-й элемент, значение которого теперь хранится в дополнительной переменной. Вставка этого элемента не должна нарушить упорядоченности элементов отсортированной части массива.
3. Сдвиг элементов массива с $i - 1$ позиции по j-ю на один элемент вправо, чтобы освободить найденную позицию для вставки.
4. Вставка взятого элемента в найденную j-ю позицию.

На рис. 3.1 приведены два первых шага работы алгоритма сортировки с помощью прямого включения. Отсортированная и неотсортированная части отделены вертикальной чертой.

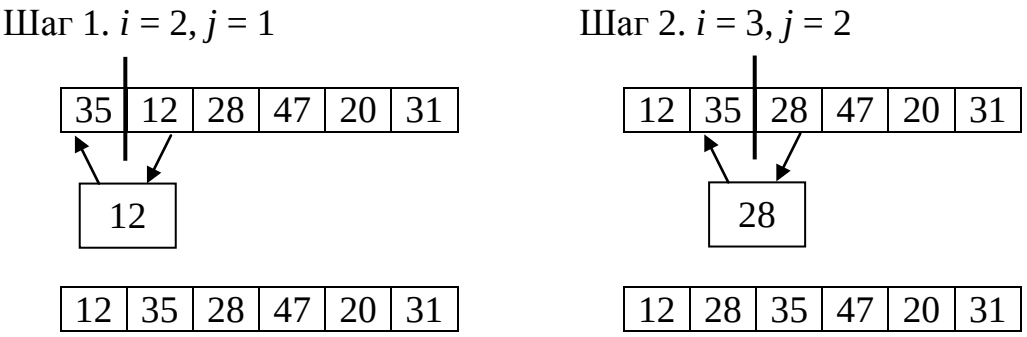


Рис. 3.1. Первые два шага алгоритма сортировки с помощью прямого включения

В табл. 3.1 показан пример работы алгоритма сортировки массива из шести элементов. Отсортированная часть массива подчеркнута.

Таблица 3.1
Пример сортировки с помощью прямого включения

	35	12	28	47	20	31
$i = 2$	12	35	28	47	20	31
$i = 3$	12	28	35	47	20	31
$i = 4$	12	28	35	47	20	31
$i = 5$	12	20	28	35	47	31
$i = 6$	12	20	28	31	35	47

Анализ алгоритма сортировки с прямым включением. Число сравнений ключей (C_i) при i-м проходе самое большее равно $i - 1$, самое меньшее – 1. Если предположить, что все перестановки из n элементов равновероятны, то среднее число сравнений – $i / 2$. Число же перестановок M_i равно $C_i + 2$.

Данный алгоритм показывает наилучшие результаты работы в случае уже упорядоченной исходной части массива, наихудшие – когда элементы первоначально расположены в обратном порядке. Алгоритм с прямым включением можно легко улучшить, если обратить внимание на то, что готовая последовательность, в которую надо вставить новый элемент, сама уже упорядочена. Естественным является остановиться на двоичном поиске, при котором делается попытка сравнения с серединой готовой последовательности, а затем процесс деления пополам идет до тех пор, пока не будет найдена точка включения. Такой модифицированный алгоритм сортировки называется методом с двоичным включением (*binary insertion*).

К несчастью, улучшения, порожденные введением двоичного поиска, касаются лишь числа сравнения, а не числа необходимых перестановок. А поскольку движение элемента, т. е. самого элемента, и связанной с ним информации занимает значительно больше времени, чем сравнение двух ключей, то фактически улучшения не столь уж существенны, ведь важный член M так и продолжает оставаться порядка n^2. И на самом деле, сортировка уже отсортированного массива потребует больше времени, чем в случае последовательной сортировки с прямыми включениями.

Этот пример показывает, что «очевидные улучшения» часто дают не столь уж большой выигрыш, как это кажется на первый взгляд, а в некоторых случаях (случающихся на самом деле) эти «улучшения» могут фактически привести к ухудшениям. После всего сказанного сортировка с помощью включения уже не кажется столь удобным методом для цифровых машин: включение одного элемента с последующим сдвигом на одну позицию целой группы элементов не экономно. Остается впечатление, что лучший результат дадут методы, где передвижка, пусть и на большие расстояния, будет связана лишь с одним-единственным элементом.

3.2. Сортировка с помощью прямого выбора

При сортировке с помощью прямого выбора массив также делится на две части: отсортированную или «готовую» последовательность a_1, …, a_{i-1} и неотсортированную или «исходную» – a_i, …, a_n.

Метод прямого выбора в некотором смысле противоположен методу прямого включения. При прямом включении на каждом шаге рассматриваются только *один* (первый) элемент исходной последовательности и *все* элементы готовой последовательности. При этом отыскивается точка включения этого элемента в «готовую» последовательность так, чтобы при вставке не нарушить ее упорядоченности.

При прямом же выборе происходит поиск *одного* элемента из исходной последовательности, который обладает наименьшим (наибольшим) значением, и уже найденный элемент помещается в конец готовой последовательности.

На момент начала сортировки методом прямого выбора готовая последовательность считается пустой, соответственно, исходная последовательность включает в себя все элементы массива.

Алгоритм сортировки с помощью прямого выбора можно описать следующим образом:

1. Из всего массива выбирается элемент с наименьшим значением.
2. Он меняется местами с первым элементом a_1.
3. Затем этот процесс повторяется с оставшимися n – 1 элементами, n – 2 элементами и т. д. до тех пор, пока не останется один элемент с наибольшим значением.

Рассмотрим несколько первых шагов алгоритма сортировки с помощью прямого выбора на примере упорядочивания по возрастанию следующего массива:

35 12 28 47 20 31

На первом шаге готовая последовательность не содержит ни одного элемента. Выбираем наименьший элемент из исходной последовательности, куда пока что входят все элементы массива. Таким элементом является второй элемент массива, значение которого – 12. Он меняется местами с первым элементом.

Теперь готовая последовательность включает в себя один элемент – 12. На рис. 3.2, иллюстрирующем текущее состояние массива, готовая последовательность подчеркнута сплошной линией. Наименьший элемент исходной последовательности – 20, он должен занять место второго элемента. Оба эти элемента выделены на рис. 3.2 полужирным шрифтом.

<u>12</u> **35** 28 47 **20** 31

Рис. 3.2. Первый шаг алгоритма сортировки с помощью прямого выбора

На втором шаге готовая последовательность состоит уже из двух элементов: 12 и 20. Наименьший элемент исходной последовательности – 28. Он является третьим элементом массива и именно эту позицию он должен

занять. Поэтому на рис. 3.3 полужирным шрифтом выделен только один данный элемент.

12 20 **28** 47 35 31

Рис. 3.3. Второй шаг алгоритма сортировки с помощью прямого выбора

В табл. 3.2 наглядно показаны все шаги алгоритма сортировки с помощью прямого выбора: отмечены готовые последовательности и переставляемые элементы.

Таблица 3.2
Пример сортировки с помощью прямого выбора

	35	**12**	28	47	20	31
$i = 2$	12	**35**	28	47	**20**	31
$i = 3$	12	20	**28**	47	35	31
$i = 4$	12	20	28	**47**	35	**31**
$i = 5$	12	20	28	31	**35**	47
$i = 6$	12	20	28	31	35	47

Анализ алгоритма сортировки с прямым выбором. При работе данного алгоритма число сравнений элементов (C) не зависит от начального порядка элементов. Можно сказать, что в этом смысле поведение данного метода менее естественно, чем поведение прямого включения. Для C имеем

$$C = (n^2 - n) / 2.$$

В случае изначально упорядоченных элементов число перестановок M минимально:

$$M_{min} = 3 * (n - 1).$$

Если же первоначально элементы располагались в обратном порядке, число перестановок будет максимальным:

$$M_{max} = n^2 / 4 + 3 * (n - 1).$$

В общем случае алгоритм с прямым выбором, как правило, предпочтительнее алгоритма прямого включения. Однако если элементы в начале упорядочены или почти упорядочены, алгоритм с прямым включением выполнит сортировку быстрее.

3.3. Сортировка с помощью прямого обмена

Алгоритм прямого обмена основывается на сравнении и перестановке пары соседних элементов и продолжении этого процесса до тех пор, пока не будут упорядочены все элементы. Обмен местами двух элементов представляет собой характерную особенность данного метода, хотя, конечно, в обоих рассматривавшихся до этого методах переставляемые элементы также «обменивались» местами.

3.3.1. Пузырьковая сортировка

Как и в методе прямого выбора при сортировке данным методом выполняется ряд проходов по массиву, сдвигая каждый раз наименьший элемент оставшейся последовательности к началу массива. Однако в отличие от рассмотренных ранее методов при пузырьковой сортировке происходит сравнивание и перестановка только соседних двух элементов.

Если расположить элементы массива вертикально, то за первый проход элемент с наименьшим значением, т. е. самый «легкий» элемент, поднимется на самый верх массива и станет его первым элементом. В результате следующего прохода второй по «легкости» элемент поднимется к началу массива и станет его вторым элементом. Такое продвижение элементов по массиву вызывает ассоциации с пузырьком воздуха, всплывающим в воде. Отсюда и название данного метода – метод «пузырька» или «пузырьковая сортировка».

Описать алгоритм сортировки методом пузырька можно следующим образом. На i-ом проходе алгоритма ($1 \le i \le n$) рассматриваются в обратном порядке элементы массива с n-го по i-й включительно. Сравниваются только соседние элементы. При этом, если производится сортировка по возрастанию и элемент $a[j]$ оказывается меньше предшествующего ему $a[j-1]$, то они меняются местами.

Если говорить терминами «готовой» и «исходной» последовательностей, то элементы до i-го представляют собой готовую последовательность, а с i-го по n-й – исходную. В начале готовая последовательность пуста. По мере работы алгоритма элементы «всплывают» из исходной последовательности и занимают место в конце готовой.

В табл. 3.3 показана работа алгоритма сортировки методом пузырька. Перемещение элементов к началу массива показано стрелками.

Таблица 3.3

Пример пузырьковой сортировки

$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$
21	4	4	4	4	4	4	4
25	21	9	9	9	9	9	9
9	25	21	12	12	12	12	12
17	9	25	21	17	17	17	17
43	17	12	25	21	21	21	21
12	43	17	17	25	25	25	25
4	12	43	32	32	32	32	32
32	32	32	43	43	43	43	43

Улучшения этого алгоритма напрашиваются сами собой. На примере в табл. 3.3 видно, что три последних прохода не влияют на порядок элементов поскольку, они уже отсортированы. Очевидный прием улучшения этого ал-

горитма – запоминать, были или не были перестановки в процессе некоторого прохода. Если в последнем проходе перестановок не было, то работа алгоритма может быть завершена.

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 уже упорядочены. Поэтому просмотр можно заканчивать на этом индексе, а не идти до заранее определенного нижнего предела i.

3.3.2. Шейкерная сортировка

Пример пузырьковой сортировки, представленный в табл. 3.3, отражает некоторую своеобразную асимметрию работы алгоритма: легкий пузырек всплывает сразу – за один проход, а тяжелый тонет очень медленно – за один проход на одну позицию. Например, массив

15 29 31 55 70 93 8

с помощью пузырьковой сортировки будет упорядочен за один проход, а для сортировки массива

93 8 15 29 31 55 70

потребуется шесть проходов. Это наводит на мысль о следующем улучшении: чередовать направление просмотра на каждом последующем проходе. Алгоритм, реализующий такой подход, называется «шейкерной сортировкой». Табл. 3.4 иллюстрирует сортировку данным методом тех же (табл. 3.3) восьми элементов. Переменные L и R содержат индексы элементов, до которых должен происходить просмотр при движении влево (или вверх) и вправо (или вниз) соответственно.

Таблица 3.4

Пример шейкерной сортировки

L =	1	2	2	3	3
R =	8	8	7	7	6
направление	↑	↓	↑	↓	↑
	21	4	4	4	4
	25	21	21	9	9
	9	25	9	21	12
	17	9	17	12	17
	43	17	25	17	21
	12	43	12	25	25
	4	12	32	32	32
	32	32	43	43	43

Если ввести переменную, которая будет отвечать за то, были ли перестановки элементов, то алгоритм шейкерной сортировки выполнит сортиров-

ку за пять проходов, в то время как для пузырьковой сортировки без улучшений потребовалось бы семь проходов.

Если же к алгоритму шейкерной сортировки добавить переменную k, содержащую индекс последнего перестановленного элемента, то число проходов сократится до четырех, как показано на примере в табл. 3.4.

Анализ пузырьковой и шейкерной сортировок. Число сравнений в базовом обменном алгоритме – алгоритме пузырьковой сортировке – равно

$$C = (n^2 - n) / 2,$$

а минимальное и максимальное число перестановок элементов равно

$$M_{min} = 0, \quad M_{max} = 3 * (n^2 - n) / 4.$$

Анализ же улучшенных методов, особенно шейкерной сортировки, довольно сложен. Стоит отметить, что все перечисленные выше усовершенствования не влияют на число перемещений, они лишь сокращают число излишних проверок. К несчастью, обмен местами двух элементов – чаще всего более дорогостоящая операция, чем их сравнение. Поэтому очевидные на первый взгляд улучшения дают не такой уж большой выигрыш, как ожидалось.

Шейкерная сортировка с успехом используется лишь в тех случаях, когда известно, что элементы почти упорядочены, что на практике бывает весьма редко. Анализ показывает, что «обменная» сортировка и ее усовершенствования фактически оказываются хуже сортировок с помощью включений и с помощью выбора.

3.4. Сортировка Шелла

Все методы прямой сортировки фактически передвигают каждый элемент на всяком элементарном шаге на одну позицию. Поэтому они требуют порядка n^2 таких шагов. Следовательно, в основу любых улучшений алгоритмов сортировки должен быть положен принцип перемещения на каждом такте элементов на большие расстояния. Можно показать, что среднее расстояние, на которое должен продвигаться каждый из n элементов во время сортировки, равно n / 3 позиций. Эта цифра является целью в поиске улучшений, т. е. в поиске более эффективных методов сортировки.

Рассмотрим улучшенный метод, основанный на методе прямого включения – сортировке с помощью включений с уменьшающимися расстояниями. В 1959 г. Д. Шеллом было предложено усовершенствование алгоритма сортировки с помощью прямого включения. Рассмотрим его работу на примере следующего массива:

35 28 49 79 45 11 89 70 91 67 54 19 13 24

Сначала отдельно группируются элементы, отстоящие друг от друга на расстоянии 4. Таких групп будет четыре, они показаны на рис. 3.4 (а – г). Элементы, принадлежащие одной группе, объединены дугами.

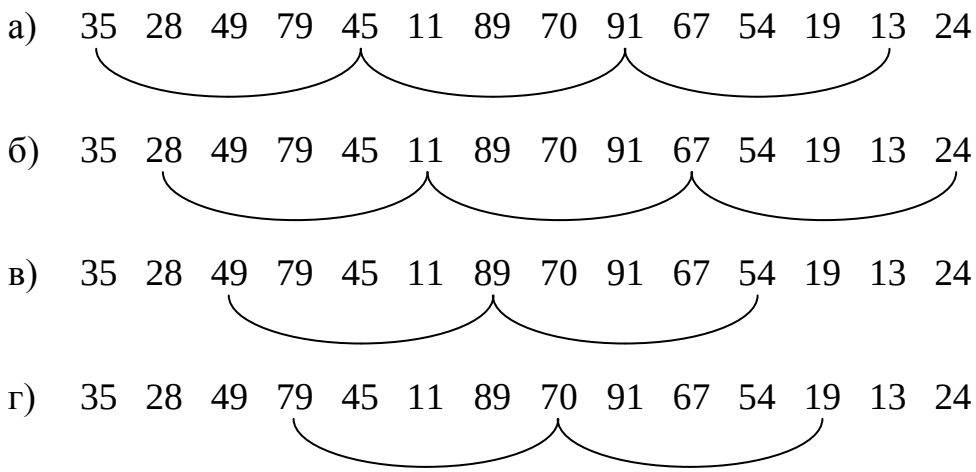


Рис. 3.4. Разбиение массива на группы элементов, отстоящих друг от друга на четыре

Следующим шагом выполняется сортировка внутри каждой из групп методом прямого включения. Сначала сортируются элементы 35, 45, 91, 13, затем 28, 11, 67, 24, следом 49, 89, 54, и, наконец, 79, 70, 19. В результате получаем массив:

13 11 49 19 35 24 54 70 45 28 89 79 91 67

Такой процесс называется четверной сортировкой. Следует отметить, что алгоритм сортировки Шелла также является алгоритмом сортировки «на месте». Поэтому все перестановки происходят в одном и том же массиве.

Следующим проходом элементы группируются так, что теперь в одну группу входят элементы, отстоящие друг от друг на две позиции. Таких групп две, они показаны на рис. 3.5 (а – б).

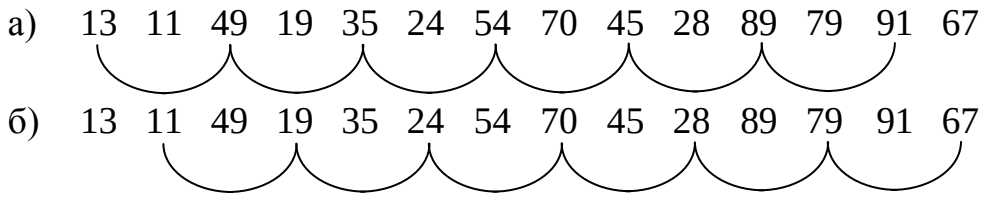


Рис. 3.5. Разбиение массива на группы элементов, отстоящих друг от друга на два

Вновь в каждой группе выполняется сортировка с помощью прямого включения. Это называется двойной сортировкой, ее результатом будет массив:

13 11 35 19 45 24 49 28 54 67 89 70 91 79

И наконец, на третьем проходе идет обычная или одинарная сортировка с помощью прямого включения (рис. 3.6).

Рис. 3.6. Группа элементов, отстоящих друг от друга на один

Результатом работы алгоритма сортировки Шелла является отсортированный массив

11 13 19 24 28 35 45 49 54 67 70 79 89 91

На первый взгляд можно засомневаться: если необходимо несколько процессов сортировки, причем и каждый включаются все элементы, то не добавят ли они больше работы, чем сэкономят? Однако на каждом этапе либо сортируется относительно мало элементов, либо элементы уже довольно хорошо упорядочены и требуется сравнительно немного перестановок.

Очевидно, что такой метод в результате дает упорядоченный массив. Видно также, что каждый проход от предыдущих только выигрывает (т. к. каждая i-сортировка объединяет две группы, уже отсортированные $2i$-сортировкой).

Расстояния в группах можно уменьшать по-разному, лишь бы последнее было единичным, ведь в самом плохом случае последний проход и сделает всю работу. Однако совсем не очевидно, что такой прием «уменьшающихся расстояний» может дать лучшие результаты, если расстояния не будут степенями двойки. При выполнении лабораторных работ рекомендуется начальный шаг взять равным $h_{нач} = n / 3$, где n – количество элементов массива, и уменьшать его на каждом проходе вдвое:

$$h_{i-1} = h_i / 2.$$

Как отмечалось выше $h_{конеч} = 1$.

Анализ сортировки Шелла. Анализ этого алгоритма поставил несколько весьма трудных математических проблем, многие из которых так еще и не решены. В частности, не известно, какие расстояния дают наилучшие результаты. Известно, однако, что выбор расстояний должен быть таким, чтобы взаимодействие различных цепочек проходило как можно чаще. В работе Кнут [3] показал, что имеет смысл использовать следующую последовательность (она записана в обратном порядке): 1, 4, 13, 40, 121, ..., где $h_{k-1} = 3h_k + 1$, h_t (или $h_{конеч}$) = 1 и $t = log_3(n) - 1$. Он рекомендует и другую последовательность: 1, 3, 7, 15, 31, ... где $h_{k-1} = 2h_k + 1$, $h_t = 1$ и $t = log_2(n) - 1$. Математический анализ показывает, что в последнем случае для сортировки n элементов методом Шелла затраты пропорциональны $n^{1.2}$, что значительно лучше n^2, необходимых для методов прямой сортировки.

3.5. Сравнение различных алгоритмов сортировки

Проанализируем эффективность различных алгоритмов сортировки массивов, состоящих из n сортируемых элементов. Анализ будет проводиться

по двум критериям: числу необходимых сравнений элементов C и числу перестановок элементов M. Для всех прямых методов сортировки можно дать точные аналитические формулы. Они приведены в табл. 3.5.

Таблица 3.5

Сравнение прямых методов сортировки

Метод		Минимальное	Среднее	Максимальное
Прямое включение	C =	$n - 1$	$(n^2 + n - 2) / 4$	$(n^2 - n) / 2 - 1$
	M =	$2(n - 1)$	$(n^2 - 9n - 10) / 4$	$(n^2 - 3n - 4) / 2$
Прямой выбор	C =	$(n^2 - n) / 2$	$(n^2 - n) / 2$	$(n^2 - n) / 2$
	M =	$3(n - 1)$	$n * (\ln n + 0.57)$	$n^2 / 4 + 3(n - 1)$
Прямой обмен	C =	$(n^2 - n) / 2$	$(n^2 - n) / 2$	$(n^2 - n) / 2$
	M =	0	$(n^2 - n) * 0.75$	$(n^2 - n) * 1.5$

В табл. 3.6 приведены фактические значения показателей C и M для массива с числом элементов равным тысячи.

Таблица 3.6

Значения показателей C и M при $n = 1000$

Метод		Минимальное	Среднее	Максимальное
Прямое включение	C =	999	250250	499499
	M =	1998	247748	498498
Прямой выбор	C =	499500	499500	499500
	M =	2997	7478	252997
Прямой обмен	C =	499500	499500	499500
	M =	0	749250	1498500

Для сортировки Шелла, как и для других усовершенствованных методов, простых и точных формул не существует. Однако, в случае сортировки Шелла вычислительные затраты составляют в среднем $n^{1.2}$, в то время как для прямых методов затраты составили бы n^2.

Для практических целей полезно иметь некоторые экспериментальные данные об эффективности того или иного алгоритма. В табл. 3.7 показано время работы (в секундах), обсуждавшихся выше методов сортировки, реализованных на персональной ЭВМ *Lilith*. Три столбца содержат времена сортировки уже упорядоченного массива, массива со случайными числами и массива, расположенного в обратном порядке. В начале приводятся цифры для массива, состоящего из 256 элементов, а затем – из 2048 элементов.

Таблица 3.7

Время работы различных методов

Метод	Массив		
	Упорядоченный	Случайный	В обратном порядке
n = 256			
Прямое включение	0.02	0.82	1.64
Двоичное включение	0.12	0.70	1.30
Прямой выбор	0.94	0.96	1.18
Метод пузырька	1.26	2.04	2.80
Метод шейкера	0.02	1.66	2.92
Сортировка Шелла	0.10	0.24	0.28
n = 2048			
Прямое включение	0.22	50.74	103.80
Двоичное включение	1.16	37.66	76.06
Прямой выбор	58.18	58.34	73.46
Метод пузырька	80.18	128.84	178.66
Метод шейкера	0.16	104.44	187.36
Сортировка Шелла	0.80	7.08	12.34

Очевидно преимущество сортировки Шелла, как улучшенного метода, по сравнению с прямыми методами сортировки. Кроме того, можно отметить следующее:

1. Улучшение двоичного включения по сравнению с прямым включением в действительности почти ничего не даст, а в случае упорядоченного массива даже получается отрицательный эффект.

2. Пузырьковая сортировка определенно наихудшая из всех сравниваемых. Ее усовершенствованная версия – шейкерная сортировка – продолжает оставаться плохой по сравнению с прямым включением и прямым выбором (за исключением патологического случая уже упорядоченного массива).

4. СОРТИРОВКА ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Алгоритмы сортировки массивов, рассмотренные во третьей главе, предполагают, что объем данных позволяет разместить и произвести их сортировку, используя исключительно оперативную память компьютера. В том случае, если объем сортируемых данных значительно превышает возможности оперативной памяти, приходится задействовать устройства внешней памяти. Однако характеристики доступа к устройствам внешней памяти отличаются от характеристик доступа к оперативной памяти. Это вынуждает использовать отличные структуры и алгоритмы обработки данных.

Рассматриваемые в данной главе алгоритмы предназначены для сортировки данных, представляющие собой (последовательный) файл, которые будем называть «последовательность». Сортировка данных, организованных в виде файлов, называется внешней сортировкой. Для каждого файла характерно, что в каждый момент непосредственно доступна одна и только одна компонента. Это весьма сильное ограничение, если сравнивать с возможностями, предоставляемыми массивами, и поэтому приходится пользоваться другими методами сортировки.

4.1. Простое слияние

Слияние означает объединение двух (или более) последовательностей в одну-единственную упорядоченную последовательность. Объединение происходит при помощи повторяющегося выбора элемента, удовлетворяющего заданному условию, из ряда доступных в данный момент элементов. Слияние намного проще сортировки, и его используют как вспомогательную операцию в более сложных процессах сортировки последовательностей.

Наиболее простым и в то же время основополагающим методом сортировки данных на основе слияния является сортировка простым слиянием. Она выполняется следующим образом:

1. Последовательность *a* разбивается на две половины: *b* и *c*. Разбиение происходит следующим образом: первый элемент последовательности *a* записывается в последовательность *b*, второй элемент последовательности *a* записывается в последовательность *c*, третий – снова в *b*, четвертый – в *c* и т. д.

2. Обе последовательности *b* и *c* сливаются в *a*. При этом одиночные элементы последовательностей *b* и *c* сравниваются и сливаются в последовательность *a* в порядке возрастания (убывания), образуя упорядоченные пары.

3. Полученная последовательность *a* вновь разбивается на две – *b* и *c*. Данный шаг аналогичен шагу 1, однако разбиение происходит не на единичные элементы, а на упорядоченные пары. То есть первая пара последовательности *a* записывается в последовательность *b*, вторая – в последовательность *c*, третья пара последовательности *a* записывается в последовательность *b*, четвертая – в *c* и т. д.

4. Слияние последовательностей *b* и *c* в последовательность *a*. При этом поочередно сравниваются элементы соответствующих пар последовательностей *b* и *c* и сливаются в последовательность *a* в порядке возрастания (убывания), образуя упорядоченные четверки.

5. Повторяя предыдущие шаги, разбиваем и сливаем четверки в восьмерки и т. д., каждый раз удваивая длину слитых подпоследовательностей до тех пор, пока не будет упорядочена целиком вся последовательность *a*.

Рассмотрим в качестве примера следующую последовательность:
a: 54 35 12 30 16 24 92 19

После разбиения (шаг 1) получаем последовательности:
b: 54 12 16 92
c: 35 30 24 19

Слияние одиночных компонент, т. е. упорядоченных подпоследовательностей длины 1, на шаге 2 даст последовательность *a*, состоящую из упорядоченных пар. Для удобочитаемости пары в последовательности ограничены апострофами:
a: 35 54 ‘ 12 30 ‘ 16 24 ‘ 19 92

На шаге 3 последовательность *a* будет разбита следующим образом:
b: 35 54 ‘ 16 24
c: 12 30 ‘ 19 92

Последовательности *b* и *c* также содержат упорядоченные пары. При слиянии этих последовательностей алгоритму доступен только один (самый левый) элемент каждой последовательности. Сравнивание элементов последовательностей *b* и *c* происходит только в соответствующих парах. Работа алгоритма на шаге 4 выглядит следующим образом:

b: **35** 54 ‘ 16 24
c: **12** 30 ‘ 19 92

сравниваются элементы 12 и 35, так как 12 < 35, то

a: 12

и

b: **35** 54 ‘ 16 24
c: **30** ‘ 19 92

далее сравниваются элементы 30 и 35, так как 30 < 35, то

a: 12 30

и

b: 35 54 ‘ 16 24

c: ‘ 19 92

Поскольку сравнивание происходит только в соответствующих парах, а пара последовательности *c* уже не содержит элементов, то пара последовательности *b* полностью сливается в последовательность *a*:

a: 12 30 35 54 ‘

последовательности *b* и *c* содержат элементы

b: 16 24

c: 19 92

Теперь начинается сравнение элементов вторых пар последовательностей *b* и *c* и слияние их в последовательность *a*. Результатом четвертого шага работы алгоритма будет последовательность *a*, содержащая упорядоченные четверки:

a: 12 30 35 54 ‘ 16 19 24 92

Очередное разбиение последовательности *a* на последовательности *b* и *c* даст:

b: 12 30 35 54

c: 16 19 24 92

Их слияние приводит, наконец, к желаемому результату:

a: 12 16 19 24 30 35 54 92

Исходя из описания алгоритма сортировки и представленного выше примера, можно выделить два основных действия над последовательностями – это разбиение исходной последовательности на две и их последующее слияние. Такие действия называются фазами: фазы разбиения и фазы слияния (рис. 4.1).

Наименьший подпроцесс, повторение которого составляет процесс сортировки, называется проходом или этапом. В приведенном примере сортировка происходит за три прохода, каждый из которых состоит из фазы разбиения и фазы слияния. Для выполнения данной сортировки потребовалось три файла или ленты, поэтому она называется трехленточным слиянием.

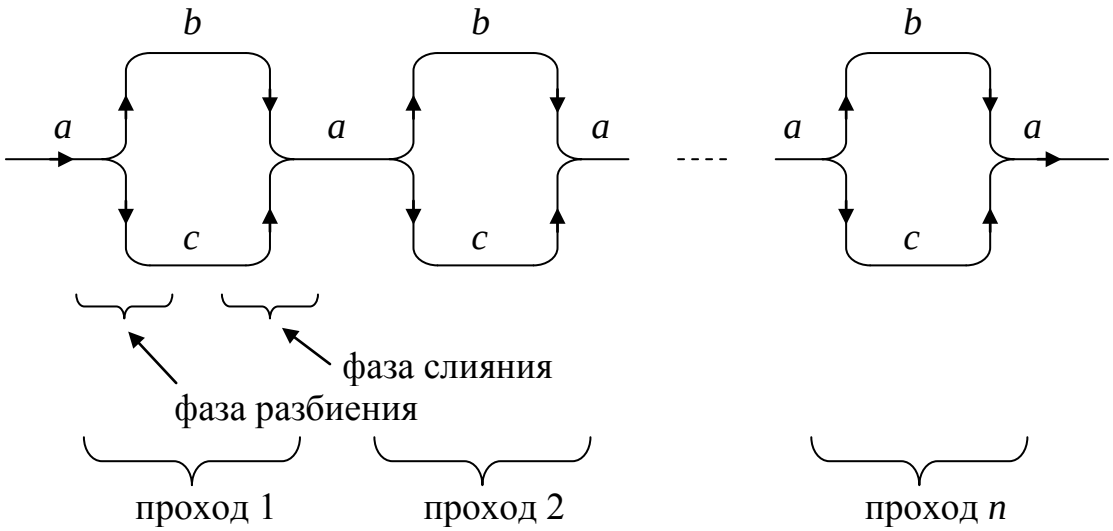


Рис. 4.1. Фазы сортировки и проходы

Фазы разбиения фактически не относятся к сортировке, ведь в них элементы не переставляются. В некотором смысле они непродуктивны, хотя и занимают половину всех операций по переписи. Если объединить разделение со слиянием, то от этих переписей можно вообще избавиться. Вместо слияния в одну последовательность результаты слияния будем сразу распределять по двум лентам, которые станут исходными для последующего прохода. В отличие от упомянутой двухфазной сортировки с помощью слияния будем называть такую сортировку однофазной. Она, очевидно, лучше, поскольку необходима только половина операций по переписи, но для этого приходится задействовать четвертую ленту.

Анализ сортировки с помощью прямого слияния. Поскольку на каждом проходе длина подпоследовательностей увеличивается вдвое, и сортировка заканчивается, когда длина подпоследовательности станет больше или равной длине исходной последовательности, то всего потребуется $log(n)$ проходов. По определению на каждом проходе все n элементов копируются по одному разу, поэтому общее число перестановок

$$M = n * log(n).$$

Число сравнений элементов C даже меньше M, так как при копировании остатков подпоследовательностей сравнения не производятся. Поскольку сортировка слиянием производится на внешних запоминающих устройствах, то затраты на операции пересылки на несколько порядков превышают затраты на сравнения. Поэтому детальный анализ числа сравнений особого практического интереса не представляет.

4.2. Естественное слияние

При сортировке простым слиянием данные разбиваются и сливаются в подпоследовательности длина, которых равна степени двойки (2^k, где $k \geq 0$). Однако данные исходной последовательности могут быть уже частично упорядочены. В этом случае целесообразно просто объединить уже упорядоченные подпоследовательности. Две упорядоченные подпоследовательности длиной m и n, содержащиеся в двух файлах, дадут на фазе слияния одну подпоследовательность из $m + n$ элементов. Сортировка, при которой сливаются упорядоченные подпоследовательности, называется естественным слиянием.

Упорядоченные подпоследовательности называют «сериями». В каждой серии элемент r_i не больше, чем r_{i+1}. Таким образом, в сортировке естественным слиянием объединяются серии, а не последовательности (заранее) фиксированной длины. Если сливаются две последовательности, каждая из n серий, то результирующая также содержит ровно n серий. Следовательно, при каждом проходе общее число серий уменьшается вдвое и общее число пересылок в самом плохом случае равно $n * log(n)$, а в среднем даже меньше. Ожидаемое число сравнений, однако, значительно больше, поскольку кроме сравнений, необходимых для отбора элементов при слиянии, нужны еще дополнительные сравнения между последовательными элементами каждого файла, чтобы определить конец серии.

Рассмотрим естественное слияние на примере двухфазной сортировки с тремя лентами. Предположим, что файл *a* содержит начальную последовательность элементов. Файлы *b* и *c* – вспомогательные. Каждый проход состоит из фазы распределения серий из *a* поочередно в *b* и *c* и фазы слияния, объединяющей серии из *b* и *c* в *a*. Этот процесс соответствует показанному на рис. 4.1.

Пусть исходная последовательность *a* содержит следующие данные

a: 15 24 32 ‘ 18 45 ‘ 40 51 ‘ 21 29 31 43 ‘ 42 60

Серии в последовательности *a* отделены апострофами. При разбиении исходной последовательности *a* в последовательности *b* и *c* серии будут распределены следующим образом:

b: 15 24 32 ‘ 40 51 ‘ 42 60

c: 18 45 ‘ 21 29 31 43

Как уже отмечалось, длина каждой серии не является заранее заданной, а вычисляется в процессе работы алгоритма. В данном примере в последовательность *b* были записаны серии длиной 3, 2 и 2 элемента. Однако эти значения не запоминаются, а конец каждой серии определятся заново на очередной фазе. Последний элемент первой из записанных серий последовательности *b* меньше первого элемента второй серии последовательности *b*. Поэтому на фазе слияния эти две серии будут рассматриваться, как одна серия состоящая из пяти элементов:

b: 15 24 32 40 51 ‘ 42 60

Таким образом, последовательность *a* получается из слияния следующих последовательностей:

b: 15 24 32 40 51 ‘ 42 60
c: 18 45 ‘ 21 29 31 43

Доступ к данным, хранящимся в файлах *b* и *c*, аналогичен доступу в прямом слиянии. То есть в любой момент времени доступен один и только один элемент каждой последовательности. В демонстрируемом примере считываемые элементы находятся слева. Именно, эти элементы сравниваются, и наименьший из них записывается в файл *a*. После естественного слияния последовательностей *b* и *c* получаем:

a: 15 18 24 32 40 45 51 ‘ 21 29 31 42 43 60

На следующем шаге разбиение последовательности *a* даст
b: 15 18 24 32 40 45 51
c: 21 29 31 42 43 60

И в результате слияния этих двух последовательностей получаем отсортированную последовательность

a: 15 18 21 24 29 31 32 40 42 43 45 51 60

Таким образом, для сортировки исходной последовательности естественным слиянием понадобилось два прохода, в то время как для сортировки прямым слиянием понадобилось бы четыре прохода.

Алгоритм сортировки естественным слиянием во многом похож на сортировку прямым слиянием. Единственное различие заключается с том, что длина подпоследовательностей в прямом слиянии кратна двойке: 1, 2, 4, 8, 16 и т. д., а в естественном слиянии длина серии вычисляется в процессе считывания элементов последовательности. И, именно, этим реализация алгоритма естественного слияния оказывается сложнее.

Серьезную трудность представляет собой определение конца серии, поскольку для этого необходимо сравнивать значения двух последовательных элементов. Однако, природа последовательности такова, что непосредственно доступен только один-единственный элемент. Для того чтобы «заглянуть вперед» можно организовать своеобразный буфер, введя переменную, куда будет считываться текущий (самый левый) элемент последовательности. Таким образом, можно будет сравнить только что считанный элемент, который теперь хранится в «буфере», со следующим за ним элементом последовательности. Если элемент, считанный в буфер, меньше следующего за ним, то элементы принадлежат одной серии. Если же элемент, содержащийся в буфере, оказывается больше последующего элемента, это обозначает конец текущей серии и начало новой.

Программный код разбиения исходной последовательности *a* на серии и запись их в последовательности *b* и *c* представлен ниже. Имя переменной-буфера, в которой сохраняется значение прочитанного элемента последовательности, – *buf*. Для файла с исходной последовательностью *a* используется указатель *f*, для последовательностей *b* и *c* – указатели *file*[0] и *file*[1], соответственно. Все три указателя имеют тип *FILE* *.

```
f = fopen(<исходный файл a>, "r");
file[0] = fopen(<файл b>, "w");
file[1] = fopen(<файл c>, "w");

fscanf(f, "%d", &elem);                    // считали один элемент из
                                           // последовательности a
fprintf( file[0], "%d ", elem);            // записали один элемент в b

buf = elem;  // buf – переменная-буфер, содержащая значение
             // текущего элемента последовательности a

j = 0;       // j служит индексом для записи в последовательности b и c,
             // если j = 0 запись происходит в последовательность b,
             // если j = 1 запись происходит в последовательность c

while( !feof(f) )            // перепись одного элемента в файл b или c
{
  fscanf(f, "%d", &elem);
  if( elem >= buf )          // серия не кончилась, запись в тот же файл
  {
        fprintf( file[j], "%d ", elem);
        buf = elem;
  }
  else                       // серия закончилась, начало записи в другой файл
  {
        j = 1 – j;     // индекс j теперь указывает на другой файл
        fprintf( file[j], "%d ", elem);
        buf = elem;
  }
}
```

Если ввести переменную, в которой при слиянии будут подсчитываться серии, то работу алгоритма можно завершить, когда в последовательности *a* будет всего одна серия. Это свидетельствует о том, что исходная последовательность уже отсортирована.

4.3. Многопутевая сортировка

Затраты на сортировку последовательностей пропорциональны числу выполняемых проходов, так как при каждом проходе копируются все данные. Одним из подходов к повышению эффективности сортировок слиянием является вовлечение в процесс дополнительных каналов обмена данными. Распределение серий на более чем две ленты позволит значительно сократить количество перестановок элементов.

Сортировка, при которой используется несколько файлов, называется многопутевым или N-путевым слиянием. Общее число проходов необходимых для сортировки многопутевым слиянием последовательности из n элементов равно $log_N(n)$. Поскольку при каждом проходе выполняется n операций копирования, то в худшем случае общее число перестановок будет равным $M = n * log_N(n)$.

При сортировке N-путевым слиянием используются $2 * N$ последовательностей. Предполагается, что число входных файлов равно числу выходных. На каждом проходе серии входных последовательностей сливаются, объединяясь, в выходные последовательности. После того, как данные во входных последовательностях заканчиваются, последовательности меняются ролями – входные последовательности становятся выходными и, наоборот.

Многопутевое слияние основывается на естественном слиянии с одной-единственной фазой. Так же как и в случае обычного двухфазного трехленточного естественного слияния возможно слияние двух следующих одна за другой серий в одну, если последний элемент какой-либо серии меньше первого элемента серии, следующей за ней.

Рассмотрим в качестве примера сортировку 3-путевым слиянием такой последовательности:

f: 57 24 88 13 19 17 96 37 42 15 21 35 23 10 53 49 33 58 16 72

Для 3-путевого слияния будут использованы 6 файлов: 3 входных и 3 выходных. Первым этапом сортировки многопутевым слиянием является разбиение исходной последовательности. Все данные разбиваются на серии и записываются в файлы f_1, f_2, f_3. Серии в этих файлах отделены апострофами:

f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72
f_2: 24 88 ‘ 37 42 ‘ 10 53 ‘ 49
f_3: 13 19 ‘ 15 21 35 ‘ 33 58

Поскольку в основе многопутевой сортировки лежит естественное слияние, то длина каждой серии заранее неизвестна и определяется непосредственно в процессе чтения данных из файла.

Указанные выше файлы f_1, f_2, f_3 послужат на первом проходе входными файлами, т. е. из них будут считываться серии и сливаться в выходные файлы f_4, f_5, f_6. В начале работы алгоритма сортировки выходные файлы пусты.

На первой итерации первого прохода первые серии входных файлов сливаются в одну серию, записываемую в выходной файл f_4:

f_1: 17 96 ‘ 23 ‘ 16 72 f_4: 13 19 24 57 88
f_2: 37 42 ‘ 10 53 ‘ 49 f_5:
f_3: 15 21 35 ‘ 33 58 f_6:

Результатом второй итерации первого прохода будет слияние вторых серий входных файлов в выходной файл f_5:

f_1: 23 ‘ 16 72 f_4: 13 19 24 57 88
f_2: 10 53 ‘ 49 f_5: 15 17 21 35 37 42 96
f_3: 33 58 f_6:

На третьей итерации первого прохода будет заполнен выходной файл f_6:

f_1: 16 72 f_4: 13 19 24 57 88
f_2: 49 f_5: 15 17 21 35 37 42 96
f_3: f_6: 10 23 33 53 58

Первый проход завершается на четвертой итерации. Входной файл f_3 пуст, поэтому в выходной файл f_4 сливаются серии, находящиеся только в файлах f_1 и f_2, все данные входных файлов заканчиваются:

f_1: f_4: 13 19 24 57 88 ‘ 16 49 72
f_2: f_5: 15 17 21 35 37 42 96
f_3: f_6: 10 23 33 53 58

Если бы во входных файлах еще содержались данные, то серии сливались бы в файл f_5, затем f_6, снова f_4, f_5, f_6 и т. д., пока не закончатся все данные во всех входных файлах.

После того, как входные файлы становятся пустыми, выполняется второй проход. На втором проходе входные файлы f_1, f_2, f_3 становятся выходными и уже в них сливаются серии из файлов f_4, f_5, f_6, которые на втором проходе становятся входными:

f_4: 13 19 24 57 88 ‘ 16 49 72 f_1:
f_5: 15 17 21 35 37 42 96 f_2:
f_6: 10 23 33 53 58 f_3:

Результат первого слияния на втором проходе алгоритма следующий:

f_4: 16 49 72 f_1: 10 13 15 17 19 21 23 24 33 35 37 42 53 57 58 88 96
f_5: f_2:
f_6: f_3:

Второй проход заканчивается на втором слиянии из единственного содержащего данные файла f_4 в выходной файл f_2.

f_4: f_1: 10 13 15 17 19 21 23 24 33 35 37 42 53 57 58 88 96
f_5: f_2: 16 49 72
f_6: f_3:

На третьем проходе входные и выходные файлы опять меняются ролями, теперь f_1, f_2, f_3 – входные файлы, а f_4, f_5, f_6 – выходные.

Алгоритм 3-путевого слияния заканчивает свою работу на третьем проходе. На этом проходе выполняется только одна итерация, на которой серии из файлов f_1 и f_2 сливаются в файл f_4:

f_1: f_4: 10 13 15 16 17 19 21 23 24 33 35 37 42 49 53 57 58 72 88 96
f_2: f_5:
f_3: f_6:

Полученную последовательность можно для удобства записать в исходный файл *f*.

Рассмотренный пример показывает, что алгоритм многопутевого слияния работает эффективнее, чем рассмотренные ранее алгоритмы прямого и естественного слияния. Так при 3-путевом слиянии для сортировки указанной последовательности потребовалось только три прохода, в то время как для естественного и прямого слияния потребовалось бы четыре и пять проходов, соответственно. К тому же последние два алгоритма требуют двойного копирования, поскольку состоят из двух фаз: разбиения и слияния.

В нашем примере используются шесть входных и выходных файлов, но для более эффективной работы их число может быть увеличено. Это требуется для работы с большими последовательностями, содержащими большое количество элементов. В этом случае будет затрачено значительно меньше времени на распределение и последующее слияние.

Теперь рассмотрим несколько вопросов связанных с реализацией алгоритма многопутевого слияния.

Поскольку при сортировке последовательностей многопутевым слиянием используется достаточно большое количество файлов, то для ссылки на файлы целесообразно использовать не отдельные переменные, а массив файлов.

Если объявить имена и указатели на используемые файлы следующим образом:

```
FILE *file[6];
char filename[6][10] = {{"f1.txt"},
                        {"f2.txt"},
                        {"f3.txt"},
                        {"f4.txt"},
                        {"f5.txt"},
                        {"f6.txt"}};
```

то можно легко открыть на чтение и запись соответствующие файлы:

```
for(i = 0; i < 3; i++)        // первые три файла открываются на чтение
  file[i] = fopen(filename[i],"r");
for(i = 3; i < 6; i++)        // остальные три файла открываются на запись
  file[i] = fopen(filename[i],"w");
```

Такой подход обладаем еще одним серьезным преимуществом. Теперь для доступа к какому-либо файлу можно использовать его индекс, что крайне

полезно при переключении группы входных и выходных последовательностей в конце каждого прохода.

Для решения задачи переключения лент можно рекомендовать технику карты индексов лент. Вместо прямой адресации ленты с помощью индекса i она адресуется через карту t, которая объявляется как

```
int t[2 * N];          // 2 * N – общее число входных и выходных файлов
```

В начале работы алгоритма $t_i = i$ для всех i. Переключение же входных и выходных файлов представляет собой обмен местами компонент t_i и t_{i+N} для всех $i = 1, \ldots, N$. В этом случае всегда можем считать, что $f_{t1}, \ldots, f_{tN}$ – входные последовательности, а $f_{t(N+1)}, \ldots, f_{t(2*N)}$ – выходные.

Для описанного выше примера 3-путевой ($N = 3$) сортировки карта индексов лент определяется как

```
int t[6];
for(i = 0; i < 2 * N; i++)
  t[i] = i;
```

Начальные значения элементов данной карты индексов лент представлены в табл. 4.1. Здесь также указаны файлы, на которые ссылается каждый элемент карты индексов лент.

Таблица 4.1

Карта индексов лент

t_i	i	Ссылка на файл
$t[1]$	1	f_1
$t[2]$	2	f_2
$t[3]$	3	f_3
$t[4]$	4	f_4
$t[5]$	5	f_5
$t[6]$	6	f_6

При переключении входных и выходных файлов меняются местами компоненты t_i и t_{i+3}, для всех $i = 1, \ldots, 3$. Результат этого процесса показан в табл. 4.2.

Таблица 4.2

Карта индексов лент после переключения входных и выходных файлов

t_i	i	Ссылка на файл
$t[1]$	4	f_4
$t[2]$	5	f_5
$t[3]$	6	f_6
$t[4]$	1	f_1
$t[5]$	2	f_2
$t[6]$	3	f_3

Таким образом, входные последовательности всегда находятся в верхней половине карты индексов лент и к ним всегда можно получить доступ через *t*[1], *t*[2] и *t*[3]. Выходные последовательности находятся в нижней половине, доступ к ним всегда осуществляется через *t*[4], *t*[5] и *t*[6].

Открыть входные файлы на чтение, а выходные на запись при помощи карты индексов лент можно следующим образом:

```
for(i = 0; i < 3; i++)        // открыть входные файлы на чтение
  file[t[i]] = fopen (filename[t[i]],"r");

for(i = 3; i < 6; i++)        // открыть выходные файлы на запись
  file[t[i]] = fopen (filename[t[i]],"w");
```

Закрыть файлы при помощи карты индексов лент можно так:

```
for(i = 0; i < 6; i++)        // закрыть все файлы
  fclose(file[t[i]]);
```

Преимущество использования карты индексов лент заключается том, что приведенный код остается неизменным независимо от того, являются входными файлами f_1, f_2, f_3 или же f_4, f_5, f_6. Главной задачей становится указание, на какой файл должен ссылаться каждый элемент карты индекса лент t_i.

При новом проходе входные файлы становятся выходными, т. е. их индексы перемещаются во вторую половину карты индексов лент, выходные же файлы становятся входными – их индексы перемещаются в первую половину карты индексов лент. Таким образом происходит переиндексация файлов. Затем входные файлы открываются на чтение, выходные – на запись, и выполняется поочередное слияние серий из входных в файлов в выходные. В завершении каждого прохода все файлы закрываются.

При слиянии серий необходимо точно идентифицировать текущие входные последовательности. Может оказаться, что число входных файлов будет меньше, чем *N*. Данная ситуация характерна для последних проходов. Так, в примере 3-путевого слияния в начале третьего прохода число входных файлов равно двум. Поэтому следует ввести некоторую переменную, скажем *k*1, задающую фактическое число работающих входов. Инициализация *k*l может выглядеть следующим образом:

```
if (L < N)  k1 = L;
else  k1 = N;
```

где *L* – число серий, полученных при последнем слиянии.

Возвращаясь к примеру 3-путевого слияния, можно сказать, что в начале первого прохода *k*1 = 3, в то время, как в начале второго слияния на втором проходе, где входной файл f_4 содержит элементы 16, 49, 72, а остальные два входных файла f_5 и f_6 пусты, *k*1 = 1.

Понятно, что по исчерпании какого-либо входа *k*l должен уменьшаться.

Слияние серий входных файлов включает в себя повторяющийся выбор наименьшего из элементов соответствующих серий и отсылку его в выходной файл. Этот процесс усложняется необходимостью определять конец каждой серии. Конец серии достигается в двух случаях:

очередной элемент меньше текущего элемента;

достигнут конец входного файла.

В последнем случае вход исключается из работы и *k*l уменьшается. В первом же серия «закрывается», элементы соответствующей последовательности в выборе уже не участвуют, но это продолжается лишь до того, как будут считаны все данные соответствующих серий из других входных последовательностей.

Отсюда следует, что нужна еще одна переменная, скажем *k*2, указывающая число входов, действительно используемых при выборе очередного элемента. В начале ее значение устанавливается равным *k*l и уменьшается всякий раз, когда серия заканчивается.

Опять же возвращаясь к ранее рассмотренному примеру, в начале первого прохода имеем:

f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72
f_2: 24 88 ‘ 37 42 ‘ 10 53 ‘ 49
f_3: 13 19 ‘ 15 21 35 ‘ 33 58

Входными файлами являются f_1, f_2, f_3, переменные *k*1 и *k*2 равны трем. При слиянии первых серий этих трех последовательностей в выходной файл f_4 значение *k*2 изменяется следующим образом:

f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72 f_2: 24 88 ‘ 37 42 ‘ 10 53 ‘ 49 f_3: 19 ‘ 15 21 35 ‘ 33 58	f_4: 13 **k2 = 3**
f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72 f_2: 24 88 ‘ 37 42 ‘ 10 53 ‘ 49 f_3: ‘ 15 21 35 ‘ 33 58	f_4: 13 19 **k2 = 2** (закончилась серия в f_3)
f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72 f_2: 88 ‘ 37 42 ‘ 10 53 ‘ 49 f_3: ‘ 15 21 35 ‘ 33 58	f_4: 13 19 24 **k2 = 2**
f_1: ‘ 17 96 ‘ 23 ‘ 16 72 f_2: 88 ‘ 37 42 ‘ 10 53 ‘ 49 f_3: ‘ 15 21 35 ‘ 33 58	f_4: 13 19 24 57 **k2 = 1** (закончилась серия в f_1)
f_1: ‘ 17 96 ‘ 23 ‘ 16 72 f_2: ‘ 37 42 ‘ 10 53 ‘ 49 f_3: ‘ 15 21 35 ‘ 33 58	f_4: 13 19 24 57 88 **k2 = 0** (закончилась серия в f_2)

Когда переменная $k2$ приняла значение ноль, значит, данные первых серий закончились. Переходим к слиянию вторых серий входных последовательностей. Выполняется проверка, нет ли пустых входных файлов. Поскольку все три входных файла содержат данные, то $k1 = 3$, и переменная $k2$, отражающая число активных входов, инициализируется значением $k1$, т. е. $k2 = 3$. Начинается слияние вторых серий в выходной файл f_5.

Можно сказать, что переменная $k1$ носит более глобальный характер, поскольку показывает, сколько всего входных файлов находятся в работе. Переменная $k2$ отвечает за количество активных входных файлов, т. е. таких файлов, чьи серии участвуют в слиянии в данный момент. Значение переменной $k2$ не может превышать значения $k1$.

К сожалению, недостаточно просто ввести переменную $k2$. Необходимо знать не только число активных последовательностей, но и какие, именно, из них используются. Очевидное решение – использовать массив булевых переменных, определяющих доступность последовательностей. Однако существует и другой метод, при котором процедура выбора становится более эффективной, а это, в конечном счете, наиболее часто повторяющаяся часть алгоритма.

Вместо булевого массива вводим вторую карту лент *ta*, где ta_1, …, ta_{k2} – индексы доступных входов. Эта карта будет использоваться вместо карты *t*. Размер карты *ta* в два раза меньше размера карты *t*, так как она используется для индексации только входных файлов. В начале каждого прохода карта индексов активных лент *ta* инициализируется следующим образом:

```
for(i = 0; i < k1; i++)
  ta[i] = t[i];
```

Слияние серий из входных файлов на t_j с помощью карты индексов активных лент *ta* можно записать следующим образом:

```
k2 = k1;
do
{
   выбор минимального элемента,
   ta[mx] – индекс файла, в котором находится минимальный элемент
   перемещение минимального элемента из file[ta[mx]] в file[t[j]]
   если файл file[ta[mx]] пуст – исключить входной файл
   если в файле file[ta[mx]] закончилась серия – закрыть серию
}while(k2 != 0);
```

Операция «исключить файл» предполагает уменьшение $k1$ и $k2$, а также переупорядочение индексов в карте *ta*. Оператор «закрыть серию» уменьшает только $k2$ и переупорядочивает соответствующим образом *ta*.

Рассмотрим переупорядочение индексов в карте *ta* на приведенном ранее примере 3-путевого слияния. На первом проходе входными файлами яв-

ляются f_1, f_2, f_3, первое слияние серий происходит в файл f_4. Начальное распределение серий по входным файлам выглядит следующим образом:

f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72
f_2: 24 88 ‘ 37 42 ‘ 10 53 ‘ 49
f_3: 13 19 ‘ 15 21 35 ‘ 33 58

Помимо значений элементов, содержащихся во входных и выходных файлах, приведем значение переменной $k2$ и карты индексов лент ta.

f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72	f_4: 13	$ta_1 = 1$
f_2: 24 88 ‘ 37 42 ‘ 10 53 ‘ 49		$ta_2 = 2$
f_3: 19 ‘ 15 21 35 ‘ 33 58	**$k2 = 3$**	$ta_3 = 3$

Видно, что первый индекс карты ta указывает на первый файл, второй индекс – на второй файл, третий индекс – на третий. Так как $k2 = 3$, то при слиянии используются все три индекса карты ta, а, следовательно, и все три входных файла, на которые указывают эти индексы.

f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72	f_4: 13 19	$ta_1 = 1$
f_2: 24 88 ‘ 37 42 ‘ 10 53 ‘ 49		$ta_2 = 2$
f_3: ‘ 15 21 35 ‘ 33 58	**$k2 = 2$**	$ta_3 = 3$ (не исп.)

Серия в файле f_3 закончилась, поэтому значение переменной $k2$ уменьшается на единицу. Это в свою очередь значит, что будут использоваться только первые два индекса карты ta: ta_1 и ta_2. Последний, третий, индекс ta_3 не будет использоваться до тех пор, пока не начнется слияние вторых серий входных файлов.

Индекс ta_3, который больше не принимается во внимание, должен указывать на тот файл, в котором только что закончилась серия. В данном случае это файл f_3, поэтому ta_3 должен иметь значение 3. (Конечно, он и раньше был равен трем, но важным здесь является то, что индекс ta_3, который больше не рассматривается, указывает на третий входной файл).

Очередное слияние серий приведет к следующему расположению элементов в файлах:

f_1: 57 ‘ 17 96 ‘ 23 ‘ 16 72	f_4: 13 19 24	$ta_1 = 1$
f_2: 88 ‘ 37 42 ‘ 10 53 ‘ 49		$ta_2 = 2$
f_3: ‘ 15 21 35 ‘ 33 58	**$k2 = 2$**	$ta_3 = 3$ (не исп.)

Значение $k2$ не изменилось, так как серии в файлах f_1 и f_2 не закончились. Поскольку при слиянии использовались только индексы ta_1 и ta_2, то файл f_3, на который указывает индекс ta_3, в слиянии не участвовал.

При очередном слиянии получаем:

f_1: ‘ 17 96 ‘ 23 ‘ 16 72	f_4: 13 19 24 57	$ta_1 = 2$
f_2: 88 ‘ 37 42 ‘ 10 53 ‘ 49		$ta_2 = 1$ (не исп.)
f_3: ‘ 15 21 35 ‘ 33 58	**$k2 = 1$**	$ta_3 = 3$ (не исп.)

Теперь закончилась серия в файле f_1. Значение переменной $k2$ уменьшается на единицу. Это значит, что индекс ta_2 больше не рассматривается. Данный индекс должен получить значение 1, т. е. указывать на файл с закончившейся серией f_1 ($ta_2 = 1$). Однако нельзя терять его бывшее значение – 2. Это значение должно быть присвоено тому индексу, который ранее указывал на файл f_1. Этим индексом является ta_1 и поэтому $ta_1 = 2$. То есть теперь индекс ta_1 указывает на файл f_2, единственный файл, в котором еще осталась незаконченная серия.

Следующее слияние приведет к тому, что все элементы первых серий входных файлов закончатся:

f_1: ‘ 17 96 ‘ 23 ‘ 16 72	f_4: 13 19 24 57 88	$ta_1 = 2$ (не исп.)
f_2: ‘ 37 42 ‘ 10 53 ‘ 49		$ta_2 = 1$ (не исп.)
f_3: ‘ 15 21 35 ‘ 33 58	**$k2 = 0$**	$ta_3 = 3$ (не исп.)

Значение переменной $k2$ станет равным нулю, т. е. индексы карты ta больше не используются. Так заканчивается слияние первых серий входных последовательностей. Можно лишь добавить, что значение переменной $k1$ все это время оставалось равным трем, так как не был достигнут конец ни одного из трех входных файлов.

На следующем цикле выполняется присвоение $k2 = k1$, т. е. $k = 3$. Это значит, что все индексы карты ta снова используются при слиянии. Значения ta_i при этом сохраняются, т. е. индекс ta_1 указывает на файл f_2, ta_2 указывает на f_1, а ta_3 на файл f_3. Порядок входных файлов в карте индексов ta не имеет значения. Существенным является лишь то, что все индексы карты индексов активных лент ta указывают на все существующие входные последовательности.

Как уже отмечалось, значение переменной $k1$ уменьшается на единицу при достижении конца одного из файлов. Увидеть это можно на третьей итерацией первого прохода, где слияние происходит в выходной файл f_6.

f_1: 23 ‘ 16 72		f_4: 13 19 24 57 88	$ta_1 = 1$
f_2: 10 53 ‘ 49		f_5: 15 17 21 35 37 42	$ta_2 = 2$
f_3: 33 58	**k1 = 3** $k2 = 3$	f_6:	$ta_3 = 3$
f_1: 23 ‘ 16 72		f_4: 13 19 24 57 88	$ta_1 = 1$
f_2: 53 ‘ 49		f_5: 15 17 21 35 37 42	$ta_2 = 2$
f_3: 33 58	**k1 = 3** $k2 = 3$	f_6: 10	$ta_3 = 3$
f_1: ‘ 16 72		f_4: 13 19 24 57 88	$ta_1 = 3$
f_2: 53 ‘ 49		f_5: 15 17 21 35 37 42	$ta_2 = 2$
f_3: 33 58	**k1 = 3** $k2 = 2$	f_6: 10 23	$ta_3 = 1$ (не исп.)
f_1: ‘ 16 72		f_4: 13 19 24 57 88	$ta_1 = 3$
f_2: 53 ‘ 49		f_5: 15 17 21 35 37 42	$ta_2 = 2$
f_3: 58	**k1 = 3** $k2 = 2$	f_6: 10 23 33	$ta_3 = 1$ (не исп.)
f_1: ‘ 16 72		f_4: 13 19 24 57 88	$ta_1 = 3$
f_2: ‘ 49		f_5: 15 17 21 35 37 42	$ta_2 = 2$ (не исп.)
f_3: 58	**k1 = 3** $k2 = 1$	f_6: 10 23 33 53	$ta_3 = 1$ (не исп.)
f_1: ‘ 16 72		f_4: 13 19 24 57 88	$ta_1 = 3$ (не исп.)
f_2: ‘ 49		f_5: 15 17 21 35 37 42	$ta_2 = 2$ (не исп.)
f_3:	**k1 = 2** $k2 = 0$	f_6: 10 23 33 53 58	$ta_3 = 1$ (не исп.)

На следующей итерации произойдет слияние входных файлов f_1 и f_2 в файл f_4, и первый проход завершится. При новом проходе входные и выходные файлы должны поменяться ролями:

for(i = 0; $i < N$; i++)
{ *temp* = $t[i]$; $t[i] = t[i + N]$; $t[i + N]$ = *temp*; }

Проходы выполняются до тех пор, пока не останется последовательность, состоящая из одной-единственной серии.

Резюмируя все выше сказанное, можно дать общий «набросок» программы, реализующий алгоритм многопутевого слияния:

инициализация индексов карты лент t_i, для всех i = 1, …, 2 * N;
распределение серий исходной последовательности во входные файлы $t[1]$, …, $t[N]$,
при этом подсчитывается общее количество серий L;
пока $L > 1$

```
{
    если L < N, тогда k1 = L, иначе k1 = N;
    инициализация индексов карты активных входов: ta_i = t_i, для i = 1, …, N;
    j = N + 1, где j – индекс текущего выходного файла;
    пока k1 > 0
    {
        L = L +1;
        k2 = k1;
        пока k2 > 0
        {
            поиск минимального элемента в рассматриваемых сериях,
            ta[mx] – индекс файла, в котором содержится мин. элемент;
            запись минимального элемента из ta[mx] в t[j];
            если конец файла ta[mx], то k1 = k1 – 1, k2 = k2 – 1,
                переупорядочивание индексов карты ta;
            иначе если серия в ta[mx] закончилась, то k2 = k2 – 1,
                переупорядочивание индексов карты ta;
        }
        смена выходного файла: если j < 2 * N, тогда j = j + 1, иначе j = N + 1;
    }
    входные и выходные файлы меняются ролями: t_i меняется с t_{i + N}, для i = 1, …, N;
}
```

4.4. Многофазная сортировка

В основе сортировки последовательности многофазным слиянием лежит распределение начальных серий в соответствии с числами Фибоначчи. Поэтому, перед тем, как перейти непосредственно к алгоритму многофазной сортировки, рассмотрим числа Фибоначчи.

Числа Фибоначчи

В 1202 году итальянский математик Леонардо Фибоначчи поставил в своей книге «Liber abacci» следующую задачу: Человек посадил пару кроликов в загон, окруженный со всех сторон стеной. Сколько пар кроликов будет через *n* месяцев, если известно, что через месяц каждая пара кроликов производит на свет еще одну пару, которая в свою очередь становится способной производить потомство со второго месяца после своего рождения?

В этой же книге Фибоначчи приводит решение этой задачи: количество пар кроликов будет увеличиваться каждый месяц следующим образом:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Иными словами, число пар кроликов создает ряд, каждый член которого равен сумме двух предыдущих. Этот ряд известен как ряд Фибоначчи, а сами числа, как числа Фибоначчи.

На рис. 4.2 показано распределение серий при многофазной сортировке слиянием с тремя последовательностями, две из которых являются входными, одна – выходной. На каждом проходе элементы из двух входных последовательностей сливаются в третью. Как только одна из входных последовательностей исчерпывается, она становится выходной. Числа, указанные на рис. 4.2, показывают количество серий в последовательности и длину серий.

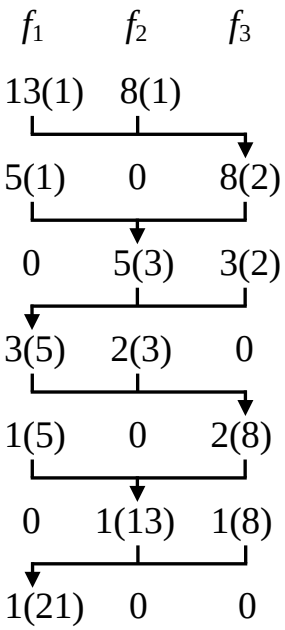


Рис. 4.2. Многофазная сортировка с тремя последовательностями

Слияние начинается с двух последовательностей f_1 и f_2, организованных в виде серий длины 1. Серии из f_1 и f_2 объединяются, образуя серии длины 2, и записываются в третью последовательность f_3. Слияние происходит до полного опустошения последовательности f_2 (как видно из рис. 4.2 в последовательности f_2 серий меньше, чем в f_1). Затем объединяем оставшиеся серии длины 1 из f_1 с таким же количеством серий длины 2 из f_3. В результате получаются серии длины 3, которые помещаются в файл f_2. Затем объединяются серии длины 2 из f_3 с сериями длины 3 из f_2. Эти серии длины 5 помещаются в последовательность f_1, которая была исчерпана во время предыдущего прохода. Процесс продолжается до тех пор, пока в f_1 не окажется отсортированная последовательность.

Оказывается, чтобы сохранить нужный ход процесса сортировки, начальное количество серий в f_1 и f_2 должно представлять собой два последовательных числа Фибоначчи. Так на рис. 4.2 показан процесс сортировки 21 серии, распределенных следующим образом: 13 серий в последовательности f_1 и 8 серий – в f_2.

В табл. 4.3 приведено количество серий во входных файлах $a_1^{(L)}$ и $a_2^{(L)}$ и необходимое для их слияния число проходов L. Понятие прохода при многофазной сортировке менее жесткое, чем в рассмотренных ранее алгоритмах внешней сортировки, и представляет собой слияние серий из входных фай-

лов в выходной, пока один из входных файлов не станет пустым. Как только опустошается один из входных файлов, проход считается завершенным, несмотря на то, что в других входных файлах еще остались серии, не участвовавшие в слиянии. Опустошившийся входной файл на новом проходе становится выходным, все остальные файлы – входными.

Таблица 4.3
Идеальное распределение серий по двум последовательностям

L	$a_1^{(L)}$	$a_2^{(L)}$	$Sum\ a_i^{(L)}$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

В табл. 4.3 также приведена сумма серий в обеих входных последовательностях. Если забыть об условии, что количество серий, хранящихся в последовательных файлах, заранее неизвестно, то можно сказать, что идеальным для многофазной сортировки будет количество серий равное одному из чисел, приведенных в четвертом столбце табл. 4.3. Здесь же приводится и идеальное распределение серий по входным последовательностям.

Так, в рассмотренном примере, исходная последовательность содержит 21 серию, что является идеальным числом серий для алгоритма многофазного слияния. Во входные последовательности будут распределены соответственно 13 и 8 серий. Как видно из табл. 4.3 для сортировки данной последовательности потребуется шесть проходов.

Если бы исходная последовательность содержала 8 серий, то 5 из них были бы распределены в первую входную последовательность, 3 – во вторую. Исходная последовательность была бы отсортирована за четыре прохода.

Из табл. 4.3 при $L > 0$ можно вывести следующие соотношения:
$a_2^{(L+1)} = a_1^{(L)}$,
$a_1^{(L+1)} = a_1^{(L)} + a_2^{(L)}$,
и $a_1^{(0)} = 1$, $a_2^{(0)} = 0$. Принимая $f_{i+1} = a_1^{(i)}$, получаем для $i > 0$:
$f_{i+1} = f_i + f_{i-1}$, $f_1 = 1$, $f_0 = 0$.
Эти рекуррентные отношения задают числа Фибоначчи:
$f = 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...$

Эти числа Фибоначчи называются числами Фибоначчи первого порядка. Существуют числа Фибоначчи и более высоких порядков. В общем виде числа Фибоначчи порядка p определяются следующим образом:

$f^{(p)}_{i+1} = f^{(p)}_i + f^{(p)}_{i-1} + \ldots + f^{(p)}_{i-p}$ для $i \geq p$,
$f^{(p)}_p = 1$,
$f^{(p)}_i = 0$ для $0 \leq i < p$.

Рассмотрим второй пример, где в многофазном слиянии участвуют шесть последовательностей, одна из которых – выходная. Данный пример показан на рис. 4.3.

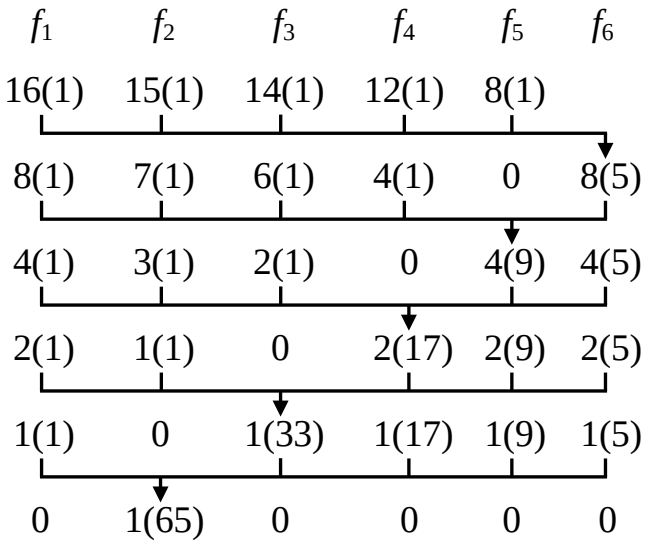


Рис. 4.3. Многофазная сортировка с шестью последовательностями

В данном примере начальное распределение серий следующее: в f_1 находится 16 серий, в f_2 – 15, в f_3 – 14, в f_4 – 12 серий и 8 серий в f_5. На первом проходе сливаются и помещаются в f_6 8 серий. Обратите внимание, что длина этих восьми серий равна пяти, поскольку в слиянии участвовали серии длины 1, хранящиеся в пяти входных файлах. Процесс слияния серий аналогичен слиянию с тремя последовательностями. Заканчивается многофазное слияние с шестью последовательностями через пять проходов, когда f_2 содержит все отсортированное множество элементов.

В примере многофазного слияния с шестью последовательностями выполнялась сортировка 65 элементов (или серий длины 1). Такое количество было выбрано преднамеренно, поскольку идеально подходит для многофазной сортировки с шестью последовательностями, в которой используются числа Фибоначчи четвертого порядка. То есть порядок используемых чисел Фибоначчи зависит от количества последовательностей: при многофазной сортировки с N последовательностями используются числа Фибоначчи $N - 2$ порядка.

В табл. 4.4 приведено распределение серий по пяти входным последовательностям, соответствующих приведенному на рис. 4.3 примеру многофазной сортировки с шестью последовательностями.

Таблица 4.4
Идеальное распределение серий по пяти последовательностям

L	$a_1^{(L)}$	$a_2^{(L)}$	$a_3^{(L)}$	$a_4^{(L)}$	$a_5^{(L)}$	$Sum\ a_i^{(L)}$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Исходя из табл. 4.4, можно вывести правила образования числа серий на каждом проходе:

$$a_5^{(L+1)} = a_1^{(L)},$$
$$a_4^{(L+1)} = a_1^{(L)} + a_5^{(L)} = a_1^{(L)} + a_1^{(L-1)},$$
$$a_3^{(L+1)} = a_1^{(L)} + a_4^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)},$$
$$a_2^{(L+1)} = a_1^{(L)} + a_3^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)} + a_1^{(L-3)},$$
$$a_1^{(L+1)} = a_1^{(L)} + a_2^{(L)} = a_1^{(L)} + a_1^{(L-1)} + a_1^{(L-2)} + a_1^{(L-3)} + a_1^{(L-4)},$$

Подставляя f_i вместо $a_1^{(i)}$, получаем:

$f_{i+1} = f_i + f_{i-1} + f_{i-2} + f_{i-3} + f_{i-4}$, для $i \geq 4$,

$f_4 = 1$,

$f_i = 0$, для $i < 4$,

что полностью соответствует приведенным выше правилам, определяющим числа Фибоначчи порядка p.

Таким образом, любое из чисел Фибоначчи четвертого порядка равно сумме пяти предшествующих ему чисел. Ряд Фибоначчи четвертого порядка выглядит следующим образом:

$f^{(4)} = 0, 0, 0, 0, 1, 1, 2, 4, 8, 16, \ldots$

Эти числа можно увидеть в табл. 4.4 в столбце, соответствующему $a_1^{(L)}$.

В завершение обсуждения чисел Фибоначчи приведем табл. 4.5, в которой содержится количество серий, идеально подходящих для многофазной сортировки с N последовательностями, значение L соответствует числу проходов, необходимых для сортировки всех элементов исходной последовательности.

Таблица 4.5

Количество серий, допускающее идеальное распределение

L	N					
	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	193
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049
11	233	1201	2500	3777	4961	6073
12	377	2209	4819	7425	9841	12097
13	610	4063	9289	14597	19521	24097
14	987	7473	17905	28697	38721	48001

Элементами табл. 4.5 являются суммы серий n последовательностей на определенном проходе. Так, значения $Sum\ a_i^{(L)}$ из табл. 4.3 можно найти в табл. 4.5 с индексом столбца $N = 3$ (в многофазной сортировке используются три последовательности). Значения $Sum\ a_i^{(L)}$ из табл. 4.4 находятся в табл. 4.5 с индексом столбца $N = 6$ (используются шесть последовательностей).

Начальное распределение серий для алгоритма многофазной сортировки

В рассмотренных примерах слияния предполагалось, что количество серий в исходной последовательности имеет значение, соответствующее одному из чисел Фибоначчи, что позволит произвести начальное распределение серий идеальным образом. На практике же это условие выполняется редко. В этом случае предлагается ввести гипотетические пустые серии, которые в сумме с реально существующими сериями дадут число, подходящее для идеального распределения серий. Новые введенные серии называются «пустыми сериями».

Пустые серии необходимо распределять по входным последовательностям как можно более равномерно, поскольку эффективность алгоритма многофазной сортировки зависит от слияния реальных серий. Если серия последовательности f_i пустая, то в слиянии она не участвует, и слияние происходит не из $N - 1$ входных последовательностей, а из меньшего их числа. А при многофазной сортировке должно использоваться как можно большее число последовательностей. Если во всех $N - 1$ последовательностях содержатся пустые серии, то никакого реального слияния не происходит, а выходную последовательность записывается пустая серия.

Рассмотрим задачу распределения серий исходной последовательности по $N - 1$ последовательностям. Поскольку мы имеем дело с внешней сортировкой, то число серий заранее неизвестно. В процессе распределения вычисляются числа Фибоначчи порядка $N - 2$, определяющие желательное число серий в каждой из входных последовательностей.

Предположим, что общее число последовательностей $N = 6$, из которых при сортировке пять будут входными. Следовательно, необходимо распределить серии по этим пяти последовательностям.

Так как при распределении серии записываются в пять последовательностей, воспользуемся табл. 4.4. Сначала серии считываются из исходной последовательности и распределяются, как указано в строке с индексом $L = 1$ (1, 1, 1, 1, 1), если в исходной последовательности еще остались серии, то переходим ко второй строке (2, 2, 2, 2, 1), если серии все еще остались, то переходим к третьей строке (4, 4, 4, 3, 2) и т. д. Будем называть индекс строки «уровнем». Очевидно, чем больше число серий, тем выше уровень чисел Фибоначчи, который в данном случае равен количеству проходов, необходимых для сортировки исходной последовательности.

Проиллюстрируем распределение серий на примере с пятью последовательностями f_1, f_2, f_3, f_4, f_5. Для оптимального распределения серий воспользуемся алгоритмом горизонтального распределения.

Сначала записываем в последовательности по одной серии (рис. 4.4). Числа на рис. 4.4 означают порядок серий, т. е. первая серия исходной последовательности записывается в f_1, вторая серия – в f_2 и т. д.

1	2	3	4	5
f_1	f_2	f_3	f_4	f_5

Рис. 4.4. Горизонтальное распределение серий, $L = 1$

Если исходная последовательность еще содержит серии, то переходим на второй уровень, где число серий в последовательностях f_1, f_2, f_3, f_4, f_5. должно быть равно, соответственно, 2, 2, 2, 2, 1. На рис. 4.5 показаны серии последовательностей, записанные на первом уровне, а также ячейки, в которые будут теперь записываться серии из исходной последовательности.

1	2	3	4	5
f_1	f_2	f_3	f_4	f_5

Рис. 4.5. Горизонтальное распределение серий, $L = 2$

Фактически пустые ячейки представляют собой пустые серии, которые замещаются реальными сериями при считывании их из исходной последовательности.

Введем переменные a_i и d_i, отражающие идеальное число серий и число «пустых» серий для последовательности i. В начале работы алгоритма горизонтального распределения эти переменные инициализируются следующим образом:

$a_i = 1$, $d_i = 1$, для $i = 1, \ldots, N - 1$.

Эти значения соответствуют рис. 4.4.

Значения a_i приведены в табл. 4.4 и изменяются в соответствии с уровнем L. Значения d_i определяются как разность a_i текущего и предыдущего уровней: $d_i = a_i^{(L)} - a_i^{(L-1)}$. На рис. 4.6 приведены значения переменных a_i и d_i при переходе на второй уровень.

d_i	1	1	1	1	0
a_i	2	2	2	2	1

1	2	3	4	5
f_1	f_2	f_3	f_4	f_5

Рис. 4.6. Горизонтальное распределение серий при переходе на второй уровень

По мере того, как считываются реальные серии из исходной последовательности, замещая пустые серии входных последовательностей, меняются значения d_i. (рис. 4.6, 4.7).

d_i	0	1	1	1	0
a_i	2	2	2	2	1

6				
1	2	3	4	5
f_1	f_2	f_3	f_4	f_5

Рис. 4.6. Горизонтальное распределение при считывании шестой серии

d_i	0	0	0	0	0
a_i	2	2	2	2	1
	6	7	8	9	
	1	2	3	4	5
	f_1	f_2	f_3	f_4	f_5

Рис. 4.7. Горизонтальное распределение при считывании девятой серии

Переход на третий уровень (L = 3) приведет к изменению значений переменных a_i и d_i, показанному на рис. 4.8.

d_i	2	2	2	1	1
a_i	4	4	4	3	2
	6	7	8	9	
	1	2	3	4	5
	f_1	f_2	f_3	f_4	f_5

Рис. 4.8. Горизонтальное распределение серий при переходе на третий уровень

Поскольку, пустые серии должны быть распределены по последовательностям равномерно, то десятая серия исходной последовательности будет записана не в f_5 (что привело бы к d_5 = 0), а в f_1 (d_1 станет равным 1). Запись десятой серии показана на рис. 4.10.

d_i	1	2	2	1	1
a_i	4	4	4	3	2
	10				
	6	7	8	9	
	1	2	3	4	5
	f_1	f_2	f_3	f_4	f_5

Рис. 4.10. Горизонтальное распределение при считывании десятой серии

Аналогично, одиннадцатая и двенадцатая серии будут записаны в f_2 и f_3. После этого запись серий будет происходить поочередно, начиная с f_1 по f_5 (рис. 4.11, 4.12).

d_i	0	1	1	1	1
a_i	4	4	4	3	2

13				
10	11	12		
6	7	8	9	
1	2	3	4	5
f_1	f_2	f_3	f_4	f_5

Рис. 4.11. Горизонтальное распределение при считывании тринадцатой серии

d_i	0	0	0	0	0
a_i	4	4	4	3	2

13	14	15		
10	11	12	16	
6	7	8	9	17
1	2	3	4	5
f_1	f_2	f_3	f_4	f_5

Рис. 4.12. Горизонтальное распределение при считывании семнадцатой серии

Состояние последовательностей и переменных a_i и d_i при переходе на четвертый уровень показано на рис. 4.13.

d_i	4	4	3	3	2
a_i	8	8	7	6	4

13	14	15		
10	11	12	16	
6	7	8	9	17
1	2	3	4	5
f_1	f_2	f_3	f_4	f_5

Рис. 4.13. Горизонтальное распределение серий при переходе на четвертый уровень

Замещение пустых серий реальными происходит также с учетом равномерного распределения пустых серий по всем последовательностям. Порядок записи серий на четвертом уровне ($L = 4$) показан на рис. 4.14.

d_i	0	0	0	0	0
a_i	8	8	7	6	4
	29	30			
	24	25	31		
	20	21	26	32	
	18	19	22	27	
	13	14	15	23	33
	10	11	12	16	28
	6	7	8	9	17
	1	2	3	4	5
	f_1	f_2	f_3	f_4	f_5

Рис. 4.14. Горизонтальное распределение серий в конце четвертого уровня

Таким образом происходит заполнение входных последовательностей, пока в исходной последовательности не закончатся все серии. В конце распределения во входных последовательностях будут содержаться реальные и пустые серии, сумма которых представляет собой одно из чисел Фибоначчи порядка $N - 2$, что идеально подходит для многофазной сортировки.

При повышении уровня при горизонтальном распределении серий можно представить d_i как цель, которую можно достичь, если направить d_i серий в последовательность i.

Можно считать, что в последовательность i сразу помещается d_i пустых серий. Последующее распределение рассматривается как замена пустых серий на реальные, отмечая каждую замену вычитанием 1 из переменной d_i. Таким образом, по исчерпании исходной последовательности d_i будет указывать число пустых серий в последовательности i.

В общем виде алгоритм горизонтального распределения серий можно записать следующим образом:

1. Цель – получить при распределении по одной серии в каждой последовательности f_i (при распределении используются числа Фибоначчи порядка $N - 2$).
2. Проводим распределение, стремясь достичь цели.
3. Если цели невозможно достичь из-за того, что серии исходной последовательности исчерпаны, заканчиваем распределение.
4. Если цель же достигнута, вычисляем следующий уровень чисел Фибоначчи. Разность между этими числами и числами на предыдущем уровне ($d_i = a_i^{(L)} - a_i^{(L-1)}$) становится новой целью распределения. Возвращаемся к шагу 2.

Следует отметить, что для выходной последовательности $a_N = 0$, $d_N = 0$.

После того, как был рассмотрен пример и сформулирован алгоритм распределения серий, можно написать код функции *select*(), обращение к которой происходит всякий раз, когда необходимо выбрать, в какую последовательность записывать новую серию. Функция *select*() должна также вычислять новую строку табл. 4.4, т. е. значения $a_1^{(L)}$, …, $a_{N-1}^{(L)}$, и новые цели $d_i = a_i^{(L)} - a_i^{(L-1)}$. Листинг тела функции *select*() приведен ниже:

```
int i, a0;

if ((d[j]) < (d[j + 1]))  j++;
else
{
  if (d[j] == 0)
  {
    L++;
    a0 = a[0];
    for(i = 0; i < (N – 1); i++)
    {
      d[i] = a0 + a[i + 1] – a[i];
      a[i] = a0 + a[i + 1];
    }
  }
  j = 0;
}
d[j] = d[j] – 1;
```

Переменная j представляет собой индекс входной последовательности. Ее первоначальное значение равно нулю.

Если исходной последовательностью является f_0, тогда начальное распределение серий может быть сформулировано следующим образом:

```
пока (не конец последовательности f0)
{
  вызвав функцию select(), определить, куда записать серию;
  записать серию из f0 в fj;
}
```

Однако при распределении серий во входные последовательности следует учесть, что, как и при записи серий в ранее разбиравшейся сортировке естественным слиянием, возможно объединение двух последовательно поступающие в одну последовательность серий. А это приведет к нарушению ожидаемого числа серий. В многофазной сортировке особенно важно точное

число серий в каждой последовательности. Поэтому во избежание такого случайного слияния приходится усложнять алгоритм распределения.

Нужно для всех последовательностей хранить значение последнего элемента последней серии. Тогда алгоритм распределения может выглядеть так:

```
пока (не конец последовательности f0)
{
  вызвав функцию select(), определить, куда записать серию;
  если (последний элемент fj больше первого элемента f0),
  тогда
    записать серию из f0 в fj;
  иначе
    записать серию из f0 в fj;     // она объединится с последней серией fj.
    если (последовательность f0 не закончилась),
    тогда
      записать еще одну серию из f0 в fj;
                                   // и это будет действительно новая серия fj.
    иначе d[j] = d[j] + 1;         // d[j] необходимо увеличить на 1, так как в
                                   // последней строке функции select() он
                                   // был уменьшен на 1, и теперь это
                                   // нужно компенсировать.
}
```

Поскольку в начале работы алгоритма распределения серий входные последовательности пусты, то и значения их последних элементов не определены. Поэтому следует сначала скопировать по одной серии в каждую последовательность, и затем уже применять только что рассмотренный алгоритм.

Алгоритм многофазной сортировки

При обсуждении чисел Фибоначчи мы уже коснулись самого алгоритма многофазной сортировки. Так, мы знаем, что на каждом проходе серии из $n - 1$ входных файлов сливаются в один выходной файл. При этом нет необходимости использовать все серии каждого входного файла. Выходной файл заполняется сериями, количество которых равно минимальному количеству серий среди всех входных файлах. Когда серии какого-либо входного файла заканчиваются, он становится выходным файлом, и начинается новый проход. При этом все остальные файлы становятся входными.

Основная структура алгоритма многофазной сортировки подобна главной части программы многопутевого слияния. Она состоит из трех вложенных циклов: внешний цикл сливает серии, пока не будут исчерпаны входы, внутренний сливает одиночные серии из каждой входной последовательности, а самый внутренний цикл выбирает элемент с минимальным значением и пересылает его в выходной файл.

Таблица 4.6

Отличия многопутевой и многофазной сортировок

Параметры	Многопутевая сортировка	Многофазная сортировка
Число входных последовательностей	$N/2$	$N - 1$
Число выходных последовательностей	$N/2$	1
Переключение входных и выходных последовательностей в конце прохода	Входные и выходные последовательности меняются ролями	Опустошившаяся последовательность становится выходной, остальные – входными
Наличие пустых серий	Отсутствуют. Алгоритмом непредусмотрены	Присутствуют. Начальное распределение и сортировка без пустых серий невозможны

В табл. 4.6 приведены основные отличия многопутевой и многофазной сортировок. При реализации многофазной сортировки для индексации входных и выходных последовательностей, так же как и при многопутевой сортировке, используется карта индексов лент t_i. Слияние происходит из входных последовательностей $t_1, \ldots, t_{N-1}$ в t_N. Как только одна из входных последовательностей становится пустой, индексу t_N присваивается индекс этой последовательности. Таким образом, индекс t_N всегда указывает на выходную последовательность. Разумеется, предыдущее значение индекса t_N не теряется, поскольку, последовательность, на которую он указывал, теперь становится входной и участвует в слиянии наряду с остальными последовательностями.

Для индексации активных входных последовательностей используются карты индексов активных входов *ta*. Эта карта содержит индексы последовательностей, которые участвуют в слиянии. Считается, что последовательность участвует в слиянии, если она не содержит пустых серий, и конец текущей рассматриваемой серии еще не наступил. Число входных последовательностей, участвующих в слиянии, или число активных входов, обозначается через k ($k \le N - 1$).

Число активных входных последовательностей меняется от серии к серии. В начале каждой серии оно определяется по числу пустых серий d_i: если для всех, то делаем вид, что сливаем $N - 1$ пустых серий и создаем пустую серию на выходе, т. е. просто увеличиваем d_N для выходной последовательности.

В противном случае сливается по одной серии из всех входных последовательностей, у которых $d_i = 0$, а d_i остальных последовательностей уменьшается на единицу. Это означает, что из них взято по одной пустой серии.

При многофазной сортировке уровень, вычисленный при начальном распределении, уменьшается. При этом перевычисляется и необходимое число серий a_i последовательности i. Как только уровень станет равным нулю – сортировка закончена, исходная последовательность отсортирована.

В общем виде алгоритм многофазной сортировки можно записать так:

```
do                          // слияние из t1, …, tN–1 в tN
{
  z = a[N – l]; d[N] = 0;
  открываем на запись последовательность tN;
  do                        // слияние одной серии
  {
    k = 0;
    for(i = 0; i < (N – 1); i++)        // определение числа активных входов
      if (d[i] > 0)  d[i] = d[i] – l;
      else { k++;  ta[k] = t[i]; }

    if (k == 0) d[N] = d[N] + l;
    else  слияние одной реальной серии из t1, …, tN–1 в tN
    z—;
  }while (z > 0);
  закрываем последовательность tN; и открываем ее на чтение;
  поворот карты индексов лент t;
  вычисление a[i] для следующего уровня;
  открываем на запись последовательность tN;
  L—;                                   // уменьшаем уровень на единицу
 } while (L > 0);
отсортированный результат содержится в t[0];
```

Предполагается, что в начале все $N – 1$ входных последовательностей с исходными сериями находятся в состоянии чтения, а компоненты карты индексов лент $t_i = i$.

Операция слияния реальной серии почти идентична той же операции в сортировке с помощью многопутевого слияния, разница только в том, что здесь алгоритм исключения последовательности несколько проще:

```
do
{
  i = 0;  mx = 0;  min = первый элемент последовательности с индексом ta[0];
  while(i < k)
  {
    i++;  x = первый элемент последовательности с индексом ta[i];
    if (x < min) {min = x;  mx = i;}
  }
  скопировать элемент из последовательности с индексом ta[i] в
```

```
                                выходную последовательность, индекс которой t[N];
  if (конец серии в последовательности с индексом ta[mx])
    { ta[mx] = ta[k];  k—; }        // исключаем из списка активных входов
}while (k > 0);
```

Поворот карт индексов последовательностей, соответствующих счетчиков d_i, а также перевычисление коэффициентов a_i при переходе на низший уровень происходит следующим образом:

```
tn = t[N];  dn = d[N];  z = a[N – 1];
for(i = N; i > 1 ; i— )
  { t[i] = t[i – 1];  d[i] = d[i – 1];  a[i] = a[i – 1] – z; }
t[0] = tn;  d[0] = dn;  a[0] = z;
```

В заключение приведем листинг программы, реализующей алгоритм многофазной сортировки, в общем виде:

```
int j, a[N], d[N], L;

select()
{
int i, a0;

if ((d[j]) < (d[j + 1]))  j++;
else
{
  if (d[j] == 0)
  {
    L++;
    a0 = a[0];
    for(i = 0; i < (N – 1); i++)
    {
      d[i] = a0 + a[i + 1] – a[i];
      a[i] = a0 + a[i + 1];
    }
  }
  j = 0;
}
d[j] = d[j] – 1;
}


main()
{
```

```
int i, z, k, x, mx, min;
int t[N], ta[N –1];
FILE *f[N], f0;

for(i = 0; i < (N – 1); i++)
  открыть входные последовательности fi на запись;
for(i = 0; i < (N – 1); i++)
  { a[i] = 1;  d[i] = 1; }
L = 1;  j = 0;  a[N] = 0;  d[N] = 0;
записать по одной серии в каждый из входных файлов;

пока (не конец последовательности f0)          // выполнить начальное
{                                              // распределение
  вызвав функцию select(), определить, куда записать серию;
  если (последний элемент fj больше первого элемента f0),
  тогда
    записать серию из f0 в fj;
  иначе
    записать серию из f0 в fj;
    если (последовательность f0 не закончилась),
    тогда
      записать еще одну серию из f0 в fj;
    иначе d[j] = d[j] + 1;
}

for(i = 0; i < N; i++)
  t[i] = i;
открыть входные последовательности fi на чтение;

do                                             // сортировка
{
  z = a[N – l]; d[N] = 0;
  открываем на запись последовательность tN;
  do                    // слияние одной серии
  {
    k = 0;
    for(i = 0; i < (N – 1); i++)
      if (d[i] > 0)  d[i] = d[i] – l;
      else { k++;  ta[k] = t[i]; }

    if (k == 0) d[N] = d[N] + l;
    else
    do                  // слияние одной реальной серии из t1, …, tN –1 в tN
    {
```

```
        i = 0;  mx = 0;  min = первый элемент файла с индексом ta[0];
        while(i < k)
        {
          i++;  x = первый элемент последовательности с индексом ta[i];
          if (x < min) {min = x;  mx = i;}
        }
        скопировать элемент из последовательности с индексом ta[i] в
                              выходную последовательность, индекс которой t[N];
        if (конец серии в последовательности с индексом ta[mx])
          { ta[mx] = ta[k];  k—; }
      }while (k > 0);
      z—;
    }while (z > 0);
    закрываем последовательность t_N; и открываем ее на чтение;
    tn = t[N];  dn = d[N];  z = a[N – 1];
    for(i = N; i > 1 ; i— )                    // поворот карты индексов лент t;
      { t[i] = t[i – 1];  d[i] = d[i – 1];
        a[i] = a[i – 1] – z; }          // вычисление a[i] для следующего уровня;
    t[0] = tn;  d[0] = dn;  a[0] = z;
    открываем на запись последовательность t_N;
    L—;                                        // уменьшаем уровень на единицу
  } while (L > 0);

  вывод на экран или сохранение в файл отсортированной
                        последовательности, которая теперь содержится в t[0];
}
```

Алгоритм многофазной сортировки гораздо эффективнее рассмотренных ранее алгоритмов многопутевого, естественного и простого слияния. Его применение при сортировке последовательности позволяет значительно сократить время, необходимое для обработки данных.

Однако, очевидно, что реализация многофазной сортировки требует значительных трудозатрат на кодирование и отладку программного кода. Поэтому применение данной сортировки целесообразно при большом объеме данных и необходимости минимизации времени, затрачиваемого на сортировку.

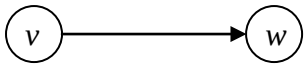
В том случае, если объем данных небольшой, а при современных вычислительных мощностях это может быть до нескольких сот тысяч элементов, рекомендуется использовать более простые методы внешней сортировки.

5. ОРИЕНТИРОВАННЫЕ ГРАФЫ

Во многих задачах, встречающихся в компьютерных науках, математике, технических дисциплинах часто возникает необходимость наглядного представления отношений между какими-либо объектами. Ориентированные и неориентированные графы – естественная модель для таких отношений [1]. В этой главе рассмотрены основные структуры данных, которые применяются для представления ориентированных графов, а также описаны некоторые основные алгоритмы определения связности ориентированных графов и нахождения кратчайших путей.

5.1. Основные определения

Ориентированный граф (или сокращенно орграф) $G = (V, E)$ состоит из множества вершин V и множества дуг E. Вершины также называют узлами, а дуги – ориентированными ребрами. Дуга представима в виде упорядоченной пары вершин (v, w), где вершина v называется началом, а w – концом дуги. Дугу (v, w) часто записывают как $v \rightarrow w$ и изображают в виде



Говорят также, что дуга $v \rightarrow w$ ведет от вершины v к вершине w, а вершина w смежная с вершиной v.

На рис. 5.1 показан орграф с четырьмя вершинами и пятью дугами.

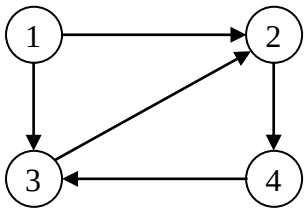


Рис. 5.1. Ориентированный граф

Вершины орграфа можно использовать для представления объектов, а дуги – для отношений между объектами. Например, вершины орграфа могут представлять города, а дуги – маршруты рейсовых полетов самолетов из одного города в другой. В другом случае, в виде орграфа может быть представлена блок-схема потока данных в компьютерной программе. В последнем примере вершины соответствуют блокам операторов программы, а дугам – направленное перемещение потоков данных.

Путем в орграфе называется последовательность вершин v_1, v_2, …, v_n для которой существуют дуги $v_1 \to v_2$, $v_2 \to v_3$, …, $v_{n-1} \to v_n$. Этот путь начинается в вершине v_1 и, проходя через вершины v_2, v_3, …, v_{n-1}, заканчивается в вершине v_n. Длина пути – это количество дуг, составляющих путь, в данном случае длина пути равна $n - 1$. Как особый случай пути рассмотрим одну вершину v как путь длины 0 от вершины v к этой же вершине v. На рис. 5.1 последовательность вершин 1, 2, 4 образуют путь длины 2 от вершины 1 к вершине 4.

Путь называется простым, если все вершины на нем, за исключением, может быть, первой и последней, различны. Цикл – это простой путь длины не менее 1, который начинается и заканчивается в одной и той же вершине. На рис. 5.1 вершины 3, 2, 4, 3 образуют цикл длины 3.

Во многих приложениях удобно к вершинам и дугам орграфа присоединить какую-либо информацию. Для этих целей используется помеченный орграф, т. е. орграф, у которого каждая дуга и/или каждая вершина имеет соответствующие метки. Меткой может быть имя, вес или стоимость (дуги), или значение данных какого-либо заданного типа.

На рис. 5.2 показан помеченный орграф, в котором каждая дуга имеет буквенную метку. Этот орграф имеет интересное свойство: любой цикл, начинающийся в вершине 1, порождает последовательность букв a и b, в которой всегда четное количество букв a и b.

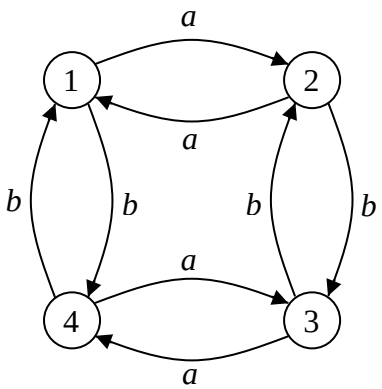


Рис. 5.2. Помеченный орграф

В помеченном орграфе вершина может иметь как имя, так и метку. Часто метки вершин используются в качестве имен вершин. Так, на рис. 5.2 числа, помещенные в вершины, можно интерпретировать как имена вершин и как метки вершин.

5.2. Представления ориентированных графов

Для представления ориентированных графов можно использовать различные структуры данных. Выбор структуры данных зависит от операторов, которые будут применяться к вершинам и дугам орграфа. Одним из наиболее общих представлений орграфа $G = (V, E)$ является матрица смежности. Предположим, что множество вершин $V = \{1, 2, \ldots, n\}$. Матрица смежности для орграфа G – это матрица A размера n x n со значениями булевого типа, где $A[i, j] = true$ тогда и только тогда, когда существует дуга из вершины i в вершину j. Часто в матрицах смежности значение *true* заменяется на 1, а значение *false* – на 0. Время доступа к элементам матрицы смежности зависит от размеров множества вершин и множества дуг. Представление орграфа в виде матрицы смежности удобно применять в тех алгоритмах, в которых надо часто проверять существование данной дуги.

Обобщением описанного представления орграфа с помощью матрицы смежности можно считать представление помеченного орграфа также посредством матрицы смежности, но у которой элемент $A[i, j]$ равен метке дуги $i \rightarrow j$. Если дуги от вершины i к вершине j не существует, то значение $A[i, j]$ не может быть значением какой-либо допустимой метки, а может рассматриваться как «пустая» ячейка.

На рис. 5.3 показана матрица смежности для помеченного орграфа, приведенного на рис. 5.2. Здесь дуги представлены меткой символьного типа, а пробел используется при отсутствии дуг.

Основной недостаток матриц смежности заключается в том, что она требует $O(n^2)$ объема памяти, даже если дуг значительно меньше, чем n^2. Поэтому для чтения матрицы или нахождения в ней необходимого элемента требуется время порядка $O(n^2)$, что не позволяет создавать алгоритмы с временем $O(n)$ для работы с орграфами, имеющими порядка $O(n)$ дуг.

	1	2	3	4
1		*a*		*b*
2	*a*		*b*	
3		*b*		*a*
4	*b*		*a*	

Рис. 5.3. Матрица смежности для помеченного орграфа из рис. 5.2

Поэтому вместо матриц смежности часто используется другое представление для орграфа $G = (V, E)$, называемое представлением посредством списков смежности. Списком смежности для вершины i называется список всех вершин, смежных с вершиной i, причем определенным образом упорядоченный. Таким образом, орграф G можно представить посредством массива *HEAD* (заголовок), чей элемент $HEAD[i]$ является указателем на список

смежности вершины i. Представление орграфа с помощью списков смежности требует для хранения объем памяти, пропорциональный сумме количества вершин и количества дуг. Если количество дуг имеет порядок $O(n)$, то и общий объем необходимой памяти имеет такой же порядок. Но и для списков смежности время поиска определенной дуги может иметь порядок $O(n)$, так как такой же порядок может иметь количество дуг у определенной вершины.

На рис. 5.4 показана структура данных, представляющая орграф из рис. 5.1 посредством связанных списков смежности. Если дуги имеют метки, то их можно хранить в ячейках связанных списков.

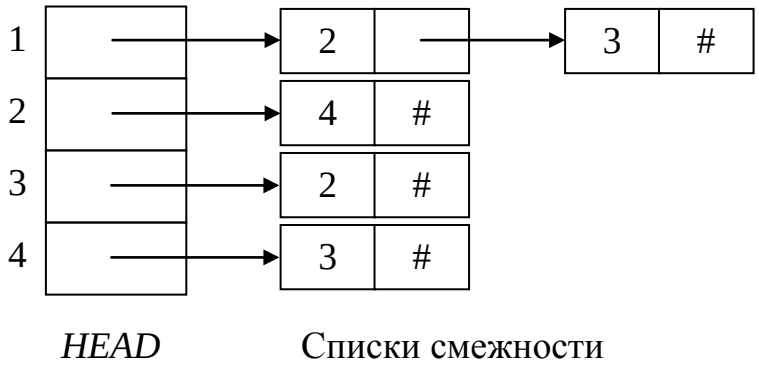


Рис. 5.4. Структура списков смежности для орграфа из рис. 5.1

Для вставки и удаления элементов в списках смежности необходимо иметь массив *HEAD*, содержащий указатель на ячейки заголовков списков смежности, но не сами смежные вершины. Но если известно, что граф не будет подвергаться изменениям (или они будут незначительны), то предпочтительно сделать массив *HEAD* массивом курсоров на массив *ADJ* (от *adjacency* – смежность), где ячейки *ADJ*[*HEAD*[i]], *ADJ*[*HEAD*[i] + 1] и т. д. содержат вершины, смежные с вершиной i, и эта последовательность смежных вершин заканчивается первым встреченным нулем в массиве *ADJ*. Пример такого представления для графа из рис. 5.1 показан на рис. 5.5.

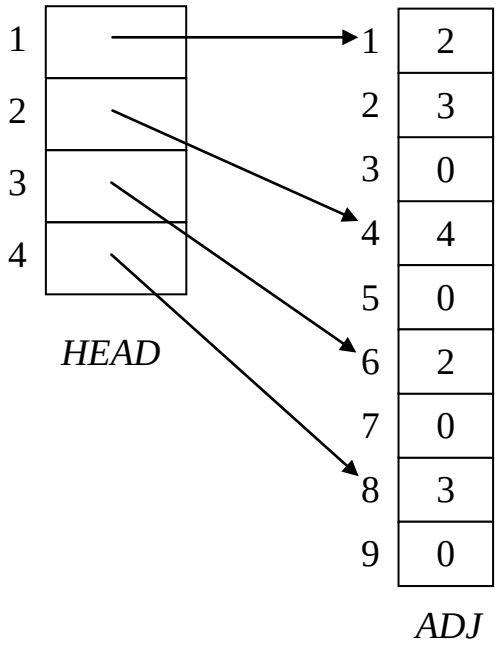


Рис. 5.5. Структура списков смежности для орграфа из рис. 5.1

5.3. Задача нахождения кратчайшего пути

В этом параграфе рассматриваются задачи нахождения путей в ориентированном графе. Пусть есть ориентированный граф $G = (V, E)$, у которого все дуги имеют неотрицательные метки (стоимости дуг), а одна вершина определена как источник. Задача состоит в нахождении стоимости кратчайших путей от источника ко всем другим вершинам графа G (здесь длина пути определяется как сумма стоимостей дуг, составляющих путь). Эта задача часто называется задачей нахождения кратчайшего пути с одним источником[1]. Отметим, что мы будем говорить о длине пути даже тогда, когда она измеряется в других, не линейных, единицах измерения, например во временных единицах.

Можно представить орграф G в виде карты маршрутов рейсовых полетов из одного города в другой, где каждая вершина соответствует городу, а дуга $v \rightarrow w$ – рейсовому маршруту из города v в город w. Метка дуги $v \rightarrow w$ – это время полета из города v в город w[2]. В этом случае решение задачи нахо-

[1] Может показаться, что более естественной задачей будет нахождение кратчайшего пути от источника к определенной вершине назначения. Но эта задача в общем случае имеет такой же уровень сложности, что и задача нахождения кратчайшего пути для всех вершин графа (за исключением того «счастливого» случая, когда путь к вершине назначения будет найден ранее, чем просмотрены пути ко всем вершинам графа).

[2] Можно предположить, что в данном случае в качестве модели больше подходит неориентированный граф, поскольку метки дуг $v \rightarrow w$ и $w \rightarrow v$ могут совпадать. Но фактически в большинстве случаев время полета в противоположных направлениях между двумя городами различно. Кроме того, предположение о совпадении меток дуг $v \rightarrow w$ и $w \rightarrow v$ не влияет (и не помогает) на решение задачи нахождения кратчайшего пути.

ждения кратчайшего пути с одним источником для ориентированного графа трактуется как минимальное время перелетов между различными городами.

Для решения поставленной задачи будем использовать «жадный» алгоритм, который часто называют алгоритмом Дейкстры (*Dijkstra*). Алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от источника меньше, чем для других оставшихся вершин. Если стоимости всех дуг неотрицательны, то можно быть уверенным, что кратчайший путь от источника к конкретной вершине проходит только через вершины множества S. Назовем такой путь особым. На каждом шаге алгоритма используется также массив D, в который записываются длины кратчайших особых путей для каждой вершины. Когда множество S будет содержать все вершины орграфа, т. е. для всех вершин будут найдены «особые» пути, тогда массив D будет содержать длины кратчайших путей от источника к каждой вершине.

Ниже представлен листинг программы, реализующей алгоритм Дейкстры. Здесь предполагается, что в орграфе G вершины поименованы целыми числами, т. е. множество вершин $V = \{1, 2, \ldots, n\}$, причем вершина 1 является источником. Массив C – это двумерный массив стоимостей, где элемент $C[i, j]$ равен стоимости дуги $i \to j$. Если дуги $i \to j$ не существует, то $C[i, j]$ принимается равным ∞, т. е. большим любой фактической стоимости дуг. На каждом шаге $D[i]$ содержит длину текущего кратчайшего особого пути к вершине i.

```
procedure Dijkstra;
begin
(1)        S := {1};
(2)        for i := 2 to n do
(3)            D[i] := C[1, i]; { инициализация D }
(4)        for i := 1 to n – 1 do begin
(5)            выбор из множества V \ S такой вершины w,
                       что значение D[w] минимально;
(6)            добавить w к множеству S;
(7)            for каждая вершина v из множества V \ S do
(8)                D[v] := min(D[v], D[w] + C[w, v])
           end
end;
```

Применим алгоритм Дейкстры для ориентированного графа, показанного на рис. 5.6.

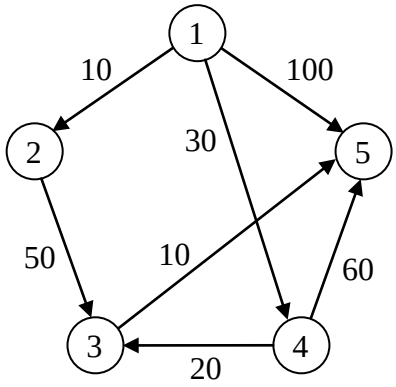


Рис. 5.6. Орграф с помеченными дугами

Вначале $S = \{1\}$, $D[2] = 10$, $D[3] = \infty$, $D[4] = 30$ и $D[5] = 100$. На первом шаге цикла (строки (4) – (8) листинга) $w = 2$, т. е. вершина 2 имеет минимальное значение в массиве D. Затем вычисляем $D[3] = \min(\infty, 10 + 50) = 60$. $D[4]$ и $D[5]$ не изменяются, так как не существует дуг, исходящих из вершины 2 и ведущих к вершинам 4 и 5. Последовательность значений элементов массива D после каждой итерации цикла показаны в табл. 5.1.

Несложно внести изменения в алгоритм так, чтобы можно было определить сам кратчайший путь (т. е. последовательность вершин) для любой вершины. Для этого надо ввести еще один массив P вершин, где $P[v]$ содержит вершину, непосредственно предшествующую вершине v в кратчайшем пути. Вначале положим $P[v] = 1$ для всех $v \neq 1$. В приведенном выше листинге после строки (8) надо записать условный оператор с условием $D[w] + C[w, v] < D[v]$, при выполнении которого элементу $P[v]$ присваивается значение w. После выполнения алгоритма кратчайший путь к каждой вершине можно найти с помощью обратного прохождения по предшествующим вершинам массива P.

Таблица 5.1

Вычисления по алгоритму Дейкстры для орграфа из рис. 5.6

Итерация	S	w	$D[2]$	$D[3]$	$D[4]$	$D[5]$
Начало	{1}	–	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3, 5}	5	10	50	30	60

Определим кратчайший путь на примере орграфа, показанного на рис. 5.6. Массив P имеет следующие значения: $P[2] = 1$, $P[3] = 4$, $P[4] = 1$, $P[5] = 3$. Для определения кратчайшего пути, например, от вершины 1 к вершине 5, надо отследить в обратном порядке предшествующие вершины, начиная с вершины 5. На основании значений массива P определяем, что вершине 5 предшествует вершина 3, вершине 3 – вершина 4, а ей, в свою оче-

редь, – вершина 1. Таким образом, кратчайший путь из вершины 1 в вершину 5 составляет следующая последовательность вершин: 1, 4, 3, 5.

Обоснование алгоритма Дейкстры

Алгоритм Дейкстры – пример алгоритма, где «жадность» окупается в том смысле, что если что-то «хорошо» локально, то оно будет «хорошо» и глобально. В данном случае что-то локально «хорошее» – это вычисленное расстояние от источника к вершине *w*, которая пока не входит в множество *S*, но имеет кратчайший особый путь. (Напомним, что особым называется путь, который проходит только через вершины множества *S*.) Чтобы понять, почему не может быть другого кратчайшего, но не особого, пути, рассмотрим рис. 5.7. Здесь показан гипотетический кратчайший путь к вершине *w*, который сначала проходит до вершины *x* через вершины множества *S*, затем после вершины *x* путь, возможно, несколько раз входит в множество *S* и выходит из него, пока не достигнет вершины *w*.

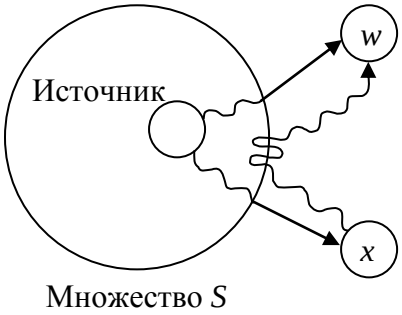


Рис. 5.7. Гипотетический кратчайший путь к вершине *w*

Но если этот путь короче кратчайшего особого пути к вершине *w*, то и начальный сегмент пути от источника к вершине *x* (который тоже является особым путем) также короче, чем особый путь к *w*. Но в таком случае в строке листинга (5) при выборе вершины *w* необходимо было выбрать не эту вершину, а вершину *x*, поскольку *D*[*x*] меньше *D*[*w*]. Таким образом, приходим к противоречию, следовательно, не может быть другого кратчайшего пути к вершине *w*, кроме особого. (Отметим здесь определяющую роль того факта, что все стоимости дуг неотрицательны, без этого свойства помеченного орграфа алгоритм Дейкстры не будет работать правильно.)

Для завершения обоснования алгоритма Дейкстры надо еще доказать, что *D*[*v*] действительно показывает кратчайшее расстояние до вершины *v*. Рассмотрим добавление новой вершины *w* к множеству *S* (строка листинга (6)), после чего происходит пересчет элементов массива *D* в строках (7), (8) этого листинга; при этом может получиться более короткий особый путь к некоторой вершине *v*, проходящий через вершину *w*. Если часть этого пути до вершины *w* проходит через вершины предыдущего множества *S* и затем непосредственно к вершине *v*, то стоимость этого пути, *D*[*w*] + *C*[*w*, *v*], в

строке листинга (8) сравнивается с $D[v]$ и, если новый путь короче, изменяется значение $D[v]$.

Существует еще одна гипотетическая возможность кратчайшего особого пути к вершине v, которая показана на рис. 5.8.

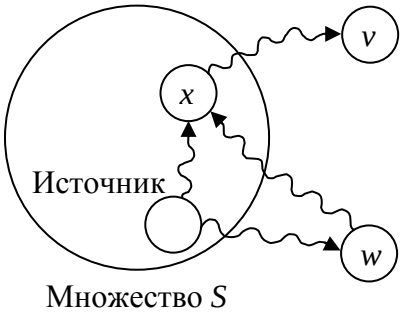


Рис. 5.8. Реально невозможный кратчайший особый путь

Здесь этот путь сначала идет к вершине w, затем возвращается к вершине x, принадлежащей предыдущему множеству S, затем следует к вершине v. Но реально такого кратчайшего пути не может быть. Поскольку вершина x помещена в множество S раньше вершины w, то все кратчайшие пути от источника к вершине x проходят исключительно через вершины предыдущего множества S. Поэтому показанный на рис. 5.8 путь к вершине x, проходящий через вершину w, не короче, чем путь к вершине x, проходящий через вершины множества S. В результате и весь путь к вершине v, проходящий через вершины x и w, не короче, чем путь от источника к вершине x, проходящий через вершины множества S, и далее непосредственно к вершине v. Таким образом, доказано, что оператор в строке листинга (8) действительно вычисляет длину кратчайшего пути.

Время выполнения алгоритма Дейкстры

Предположим, что алгоритм Дейкстры оперирует с орграфом, имеющим n вершин и e дуг. Если для представления орграфа используется матрица смежности, то для выполнения внутреннего цикла строк (7) и (8) потребуется время $O(n)$, а для выполнения всех $n - 1$ итераций цикла строки (4) потребуется время порядка $O(n^2)$. Время, необходимое для выполнения оставшейся части алгоритма, как легко видеть, не превышает этот же порядок.

Если количество дуг e значительно меньше n^2, то лучшим выбором для представления орграфа будут списки смежности, а для множества вершин $V \setminus S$ – очередь с приоритетами, реализованная в виде частично упорядоченного дерева. Тогда время выбора очередной вершины из множества $V \setminus S$ и пересчет стоимости путей для одной дуги составит $O(\log n)$, а общее время выполнения цикла строк (7) и (8) – $O(e \log n)$, а не $O(n^2)$.

Строки (1) – (3) выполняются за время порядка $O(n)$. При использовании очередей с приоритетом для представления множества $V \setminus S$ строка (5) реализуется посредством оператора *DELETEMIN*, а каждая из $n - 1$ итераций цикла (4) – (6) требует времени порядка $O(\log n)$.

В результате получаем, что общее время выполнения алгоритма Дейкстры ограничено величиной порядка $O(e \log n)$. Это время выполнения значительно меньше, чем $O(n^2)$, когда e существенно меньше n^2.

5.4. Нахождение кратчайших путей между парами вершин

Предположим, что имеется помеченный орграф, который содержит время полета по маршрутам, связывающим определенные города. Необходимо построить таблицу, где приводилось бы минимальное время перелета из одного (произвольного) города в любой другой. В этом случае мы сталкиваемся с общей задачей нахождения кратчайших путей, т. е. нахождения кратчайших путей между всеми парами вершин орграфа. Более строгая формулировка этой задачи следующая: есть ориентированный граф $G = (V, E)$, каждой дуге $v \rightarrow w$ этого графа сопоставлена неотрицательная стоимость $C[v, w]$. Общая задача нахождения кратчайших путей заключается в нахождении для каждой упорядоченной пары вершин (v, w) любого пути от вершины v в вершины w, длина которого минимальна среди всех возможных путей от v к w.

Можно решить эту задачу, последовательно применяя алгоритм Дейкстры для каждой вершины, объявляемой в качестве источника. Но существует прямой способ решения данной задачи, использующий алгоритм Флойда (*R. W. Floyd*). Для определенности положим, что вершины графа последовательно пронумерованы от 1 до n. Алгоритм Флойда использует матрицу A размера n x n, в которой вычисляются длины кратчайших путей. Вначале $A[i, j] = C[i, j]$ для всех $i \neq j$. Если дуга $i \rightarrow j$ отсутствует, то $C[i, j] = \infty$. Каждый диагональный элемент матрицы A равен 0.

Над матрицей A выполняется n итераций. После k-й итерации $A[i, j]$ содержит значение наименьшей длины путей из вершины i в вершину j, которые не проходят через вершины с номером, большим k. Другими словами, между концевыми вершинами пути i и j могут находиться только вершины, номера которых меньше или равны k.

На k-й итерации для вычисления матрицы A применяется следующая формула:

$$A_k[i, j] = \min(A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j]).$$

Нижний индекс k обозначает значение матрицы A после k-й итерации. Но это не означает, что существует n различных матриц, этот индекс используется для сокращения записи. Графическая интерпретация этой формулы показана на рис. 5.9.

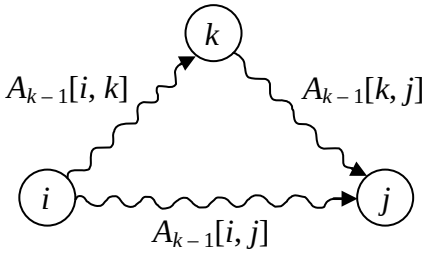


Рис. 5.9. Включение вершины k в путь от вершины i к вершине j

Для вычисления $A_k[i, j]$ проводится сравнение величины $A_{k-1}[i, j]$ (т. е. стоимость пути от вершины i к вершине j без участия вершины k или другой вершины с более высоким номером) с величиной $A_{k-1}[i, k] + A_{k-1}[k, j]$ (стоимость пути от вершины i до вершины k плюс стоимость пути от вершины k до вершины j). Если путь через вершину k дешевле, чем $A_{k-1}[i, j]$, то величина $A_k[i, j]$ изменяется.

На рис. 5.10 показан помеченный орграф, а на рис. 5.11 – значения матрицы A после трех итераций.

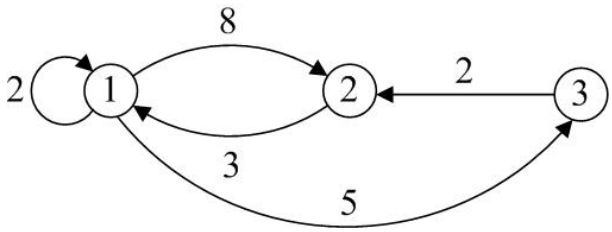


Рис. 5.10. Помеченный ориентированный граф

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_0[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_1[i, j]$

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_2[i, j]$

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

$A_3[i, j]$

Рис. 5.11. Последовательные значения матрицы A

Равенства $A_k[i, k] = A_{k-1}[i, k]$ и $A_k[k, j] = A_{k-1}[k, j]$ означают, что на k-й итерации элементы матрицы A, стоящие в k-й строке и k-м столбце, не изменяются. Более того, все вычисления можно выполнить с применением только одной копии матрицы A..

Время выполнения программы, реализующая алгоритм Флойда, имеет порядок $O(n^3)$. Доказательство «правильности» работы этого алгоритма выполняется с помощью математической индукции по k, показывая, что на k-й итерации вершина k включается в путь только тогда, когда новый путь короче старого.

Сравнение алгоритмов Флойда и Дейкстры

Поскольку версия алгоритма Дейкстры с использованием матрицы смежности находит кратчайшие пути от одной вершины за время порядка $O(n^2)$, то в этом случае применение алгоритма Дейкстры для нахождения всех кратчайших путей потребует времени порядка $O(n^3)$, т. е. получим такой же временной порядок, как и в алгоритме Флойда. Константы пропорциональности в порядках времени выполнения для обоих алгоритмов зависят от применяемых компилятора и вычислительной машины, а также от особенностей реализации алгоритмов. Вычислительный эксперимент и измерение времени выполнения – самый простой путь подобрать лучший алгоритм для конкретного приложения.

Если e, количество дуг в орграфе, значительно меньше, чем n^2, тогда, несмотря на относительно малую константу в выражении порядка $O(n^3)$ для алгоритма Флойда, рекомендуется применять версию алгоритма Дейкстры со списками смежности. В этом случае время решения общей задачи нахождения кратчайших путей имеет порядок $O(ne \log n)$, что значительно лучше алгоритма Флойда, по крайней мере, для больших разреженных графов.

Вывод на печать кратчайших путей

Во многих ситуациях требуется распечатать самый дешевый путь от одной вершины к другой. Чтобы восстановить при необходимости кратчайшие пути, можно в алгоритме Флойда ввести еще одну матрицу P, в которой элемент $P[i, j]$ содержит вершину k, полученную при нахождении наименьшего значения $A[i, j]$. Если $P[i, j] = 0$, то кратчайший путь из вершины i в вершину j состоит из одной дуги $i \rightarrow j$.

Для вывода на печать последовательности вершин, составляющих кратчайший путь от вершины i до вершины j, вызывается процедура *path*(i, j):

```
procedure path (i, j: integer);
var
    k: integer;
begin
    k := P[i, j];
```

```
    if k = 0 then
            return;
    path (i, k);
    writeln(k);
    path (k, j);
end;
```

На рис. 5.12 показана результирующая матрица P для орграфа из рис. 5.10.

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

P

Рис. 5.12. Матрица P для орграфа из рис. 5.10

Транзитивное замыкание

Во многих задачах интерес представляет только сам факт существования пути, длиной не меньше единицы, от вершины i до вершины j. Алгоритм Флойда можно приспособить для решения таких задач. Но полученный в результате алгоритм еще до Флойда разработал Уоршелл (*S. Warshall*), поэтому будем называть его алгоритмом Уоршелла.

Предположим, что матрица стоимостей C совпадает с матрицей смежности для данного орграфа G, т.е. $C[i, j] = 1$ только в том случае, если есть дуга $i \rightarrow j$, и $C[i, j] = 0$, если такой дуги не существует. Необходимо вычислить матрицу A такую, что $A[i, j] = 1$ тогда и только тогда, когда существует путь от вершины i до вершины j длиной не менее 1 и $A[i, j] = 0$ – в противном случае. Такую матрицу A часто называют транзитивным замыканием матрицы смежности.

На рис. 5.13 показано транзитивное замыкание матрицы смежности орграфа из рис. 5.10.

	1	2	3
1	1	1	1
2	1	1	1
3	1	1	1

Рис. 5.13. Матрица P для орграфа из рис. 5.10

Транзитивное замыкание можно вычислить, применяя на k-м шаге следующую формулу к булевой матрице A:

$$A_k[i, j] = A_{k-1}[i, j] \textbf{ or } (A_{k-1}[i, k] \textbf{ and } A_{k-1}[k, j]).$$

Эта формула устанавливает, что существует путь от вершины i до вершины j, проходящий через вершины с номерами, не превышающими k, только в следующих случаях:

1. Уже существует путь от вершины i до вершины j, который проходит через вершины с номерами, не превышающими $k - 1$.

2. Существует путь от вершины i до вершины k, проходящий через вершины с номерами, не превышающими $k - 1$, и путь от вершины k до вершины j, который также проходит через вершины с номерами, не превышающими $k - 1$.

Здесь $A_k[i, k] = A_{k-1}[i, k]$ и $A_k[k, j] = A_{k-1}[k, j]$, и вычисления можно выполнять в одной копии матрицы A. Программа *Warshall* вычисления транзитивного замыкания показана в следующем листинге:

```
procedure Warshall (var A: array[1..n, 1..n] of boolean;
                                C: array[1..n, 1..n] of boolean);
var
     i, j, k: integer;
begin
    for i := 1 to n do
        for j := 1 to n do
            A[i, j] := C[i, j];
   for k: = 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                if A[i, j] = false then
                    A[i, j] := A[i, k] and A[k, j]
end;
```

Нахождение центра ориентированного графа

Предположим, что необходимо найти центральную вершину в орграфе. Эту задачу также можно решить с помощью алгоритма Флойда. Однако сначала уточним термин центральная вершина. Пусть v – произвольная вершина орграфа $G = (V, E)$. Эксцентриситет или максимальное удаление вершины v определяется как

$$\max_{w \in V} \{\text{минимальная длина пути от вершины } w \text{ до вершины } v\}$$

Центром орграфа G называется вершина с минимальным эксцентриситетом. Другими словами, центром орграфа является вершина, для которой максимальное расстояние (длина пути) до других вершин минимально.

Рассмотрим помеченный орграф, показанный на рис. 5.14. В этом графе вершины имеют следующие эксцентриситеты:

Вершина	Эксцентриситет
a	∞
b	6
c	8
d	5
e	7

Найти центр орграфа сравнительно просто. Пусть C – матрица стоимостей для орграфа G.

1. Сначала применим алгоритм Флойда к матрице C для вычисления матрицы A, содержащей все кратчайшие пути орграфа G.

2. Находим максимальное значение в каждом столбце i матрицы A. Это значение равно эксцентриситету вершины i.

3. Находим вершину с минимальным эксцентриситетом. Она и будет центром графа G.

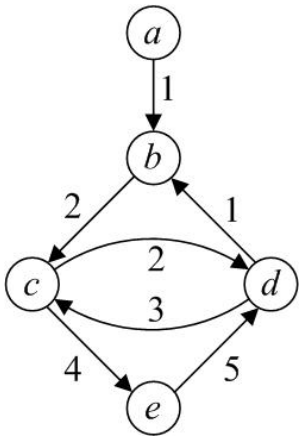


Рис. 5.14. Помеченный граф

Время выполнения этого процесса определяется первым шагом, для которого время имеет порядок $O(n^3)$. Второй шаг требует времени порядка $O(n^2)$, а третий – $O(n)$.

Матрица всех кратчайших путей для орграфа из рис. 5.14 представлена на рис. 5.15.

	a	*b*	*c*	*d*	*e*
a	0	1	3	5	7
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	6	8	5	0
max	∞	6	8	5	7

Рис. 5.15. Матрица кратчайших путей

Максимальные значения в каждом столбце приведены под матрицей.

5.5. Обход ориентированных графов

При решении многих задач, касающихся ориентированных графов, необходим эффективный метод систематического обхода вершин и дуг орграфов. Таким методом является так называемый поиск в глубину. Метод поиска в глубину составляет основу многих других эффективных алгоритмов работы с графами. В последних двух разделах этой главы представлены различные алгоритмы, основанные на методе поиска в глубину.

Предположим, что есть ориентированный граф *G*, в котором первоначально все вершины помечены меткой *unvisited* (не посещалась). Поиск в глубину начинается с выбора начальной вершины *v* графа *G*, для этой вершины метка *unvisited* меняется на метку *visited* (посещалась). Затем для каждой вершины, смежной с вершиной *v* и которая не посещалась ранее, рекурсивно применяется поиск в глубину. Когда все вершины, которые можно достичь из вершины *v*, будут посещены, поиск заканчивается. Если некоторые вершины остались не посещенными, то выбирается одна из них и поиск повторяется. Этот процесс продолжается до тех пор, пока обходом не будут охвачены все вершины орграфа *G*.

Этот метод обхода вершин орграфа называется поиском в глубину, поскольку поиск не посещенных вершин идет в направлении вперед (вглубь) до тех пор, пока это возможно. Например, пусть *x* – последняя посещенная вершина. Для продолжения процесса выбирается какая-либо нерассмотренная дуга $x \rightarrow y$, выходящая из вершины *x*. Если вершина *y* уже посещалась, то ищется другая вершина, смежная с вершиной *x*. Если вершина *y* ранее не посещалась, то она помечается меткой *visited* и поиск начинается заново от вершины *y*. Пройдя все пути, которые начинаются в вершине *y*, возвращаемся в вершину *x*, т. е. в ту вершину, из которой впервые была достигнута вершина *y*. Затем продолжается выбор нерассмотренных дуг, исходящих из вершины *x*, и так до тех пор, пока не будут исчерпаны все эти дуги.

Для представления вершин, смежных с вершиной *v*, можно использовать список смежности *L*[*v*], а для определения вершин, которые ранее посещались, – массив *mark* (метка), чьи элементы будут принимать только два значения: *visited* и *unvisited*. Эскиз рекурсивной процедуры *dfs* (от *depth-first search* – поиск в глубину), реализующей поиск в глубину, представлен в следующем листинге:

```
procedure dfs (v: вершина);
var
        w: вершина;
```

```
begin
(1)      mark[v] := visited;
(2)      for каждая вершина w из списка L[v] do
(3)          if mark[w] = unvisited then
(4)              dfs(w)
end;
```

Чтобы применить эту процедуру к графу, состоящему из n вершин, надо сначала присвоить всем элементам массива *mark* значение *unvisited*, затем начать поиск в глубину для каждой вершины, помеченной как *unvisited*. Описанное можно реализовать с помощью следующего кода:

```
for v := 1 to n do
    mark[v] := unvisited;
for v := 1 to n do
    if mark[v] = unvisited then
       dfs(v)
```

Отметим, что листинг процедуры поиска в глубину является только эскизом, который еще следует детализировать. Заметим также, что эта процедура изменяет только значения массива *mark*.

Анализ процедуры поиска в глубину

Все вызовы процедуры *dfs* для полного обхода графа с n вершинами и e дугами, если $e \geq n$, требуют общего времени порядка $O(e)$. Чтобы показать это, заметим, что нет вершины, для которой процедура *dfs* вызывалась бы больше одного раза, поскольку рассматриваемая вершина помечается как *visited* в строке листинга (1) еще до следующего вызова процедуры *dfs* и никогда не вызывается для вершин, помеченных этой меткой. Поэтому общее время выполнения строк (2) и (3) для просмотра всех списков смежности пропорционально сумме длин этих списков, т. е. имеет порядок $O(e)$. Таким образом, предполагая, что $e \geq n$, общее время обхода по всем вершинам орграфа имеет порядок $O(e)$, необходимое для «просмотра» всех дуг графа.

Рассмотрим следующий пример. Пусть процедура *dfs* применяется к ориентированному графу, представленному на рис. 5.16, начиная с вершины A.

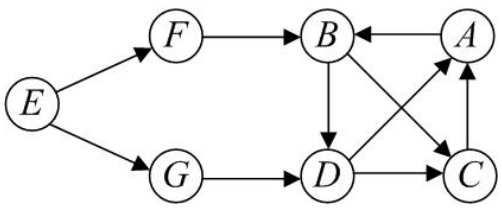


Рис. 5.16. Ориентированный граф

Алгоритм помечает эту вершину как *visited* и выбирает вершину *B* из списка смежности вершины *A*. Поскольку вершина *B* помечена как *unvisited*, обход графа продолжается вызовом процедуры *dfs*(*B*). Теперь процедура помечает вершину *B* как *visited* и выбирает первую вершину из списка смежности вершины *B*. В зависимости от порядка представления вершин в списке смежности, следующей рассматриваемой вершиной может быть или вершина *C*, или вершина *D*.

Предположим, что в списке смежности вершина *C* предшествует вершине *D*. Тогда осуществляется вызов *dfs*(*C*). В списке смежности вершины *C* присутствует только вершина *A*, но она уже посещалась ранее. Поскольку все вершины в списке смежности вершины *C* исчерпаны, то поиск возвращается в вершину *B*, откуда процесс поиска продолжается вызовом процедуры *dfs*(*D*). Вершины *A* и *C* из списка смежности вершины *D* уже посещались ранее, поэтому поиск возвращается сначала в вершину *B*, а затем в вершину *A*.

На этом первоначальный вызов *dfs*(*A*) завершен. Но орграф имеет вершины, которые еще не посещались: *E*, *F* и G. Для продолжения обхода вершин графа выполняется вызов *dfs*(*E*).

Глубинный остовный лес

В процессе обхода ориентированного графа методом поиска в глубину только определенные дуги ведут к вершинам, которые ранее не посещались. Такие дуги, ведущие к новым вершинам, называются дугами дерева и формируют для данного графа остовный лес, построенный методом поиска в глубину, или, сокращенно, глубинный остовный лес. На рис. 5.17 показан глубинный остовный лес для графа из рис. 5.16. Здесь сплошными линиями обозначены дуги дерева. Отметим, что дуги дерева формируют именно лес, т. е. совокупность деревьев, поскольку методом поиска в глубину к любой ранее не посещавшейся вершине можно придти только по одной дуге, а не по двум различным дугам.

В добавление к дугам дерева существуют еще три типа дуг, определяемых в процессе обхода орграфа методом поиска в глубину. Это обратные, прямые и поперечные дуги.

Обратные дуги (как дуга $C \rightarrow A$ на рис. 5.17) – это такие дуги, которые в остовном лесе идут от потомков к предкам. Отметим, что дуга из вершины в саму себя также является обратной дугой. Прямыми дугами называются дуги, идущие от предков к истинным потомкам, но которые не являются дугами дерева. На рис. 5.17 прямые дуги отсутствуют.

Дуги, такие как $D \rightarrow C$ и $G \rightarrow D$ на рис. 5.17, соединяющие вершины, не являющиеся ни предками, ни потомками друг друга, называются поперечными дугами. Если при построении остовного дерева сыновья одной вершины располагаются слева направо в порядке их посещения и если новые деревья добавляются в лес также слева направо, то все поперечные дуги идут справа налево, что видно на рис. 5.17. Такое взаимное расположение вершин

и деревьев выбрано не случайно, так как это помогает формировать остовный лес.

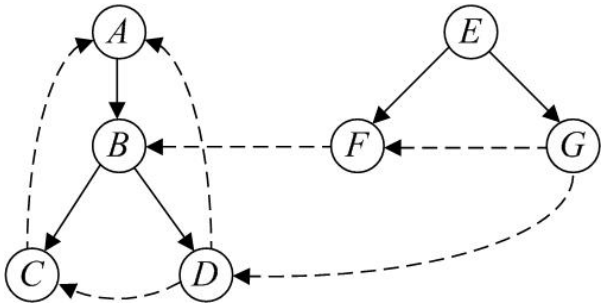


Рис. 5.17. Глубинный остовной лес для орграф из рис 5.16

Как можно различить эти четыре типа дуг? Легко определить дуги дерева, так как они получаются в процессе обхода графа как дуги, ведущие к тем вершинам, которые ранее не посещались. Предположим, что в процессе обхода орграфа его вершины нумеруются в порядке их посещения. Для этого в листинге процедуры поиска в глубину после строки (1) надо добавить следующие строки:

```
dfnumber[v] := count;
    count := count + 1;
```

Назовем такую нумерацию глубинной нумерацией ориентированного графа. Всем потомкам вершины v присваиваются глубинные номера, не меньшие, чем номер, присвоенный вершине v. Фактически вершина w будет потомком вершины v тогда и только тогда, когда выполняются неравенства $dfnumber(v) \le dfnumber(w) \le dfnumber(v)$ + количество потомков вершины v[1]. Очевидно, что прямые дуги идут от вершин с низкими номерами к вершинам с более высокими номерами, а обратные дуги – от вершин с высокими номерами к вершинам с более низкими номерами.

Все поперечные дуги также идут от вершин с высокими номерами к вершинам с более низкими номерами. Чтобы показать это, предположим, что есть дуга $x \to y$ и выполняется неравенство $dfnumber(x) \le dfnumber(y)$. Отсюда следует, что вершина x пройдена (посещена) раньше вершины y. Каждая вершина, пройденная в промежуток времени от вызова $dfs(x)$ и до завершения $dfs(y)$, становится потомком вершины x в глубинном остовном лесу. Если при рассмотрении дуги $x \to y$ вершина y еще не посещалась, то эта дуга становится дугой дерева. В противном случае дуга $x \to y$ будет прямой дугой. Таким образом, если для дуги $x \to y$ выполняется неравенство $dfnumber(x) \le dfnumber(y)$, то она не может быть поперечной дугой.

[1] Для истинных потомков вершины v первое из приведенных неравенств должно быть строгим.

Далее будет рассмотрено применение метода поиска в глубину для решения различных задач на графах.

5.6. Ориентированные ациклические графы

Ориентированный ациклический граф – это орграф, не имеющий циклов. Можно сказать, что ациклический орграф более общая структура, чем дерево, но менее общая по сравнению с обычным ориентированным графом. На рис. 5.18 представлены дерево, ациклический орграф и ориентированный граф с циклом.

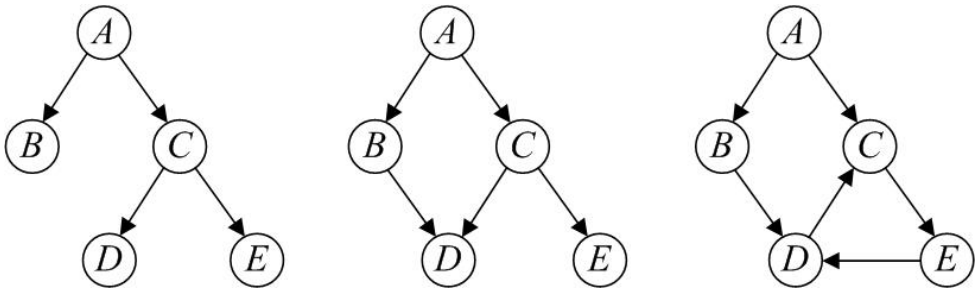


Рис. 5.18. Три ориентированных графа

Ациклические ориентированные графы удобны для представления синтаксических структур арифметических выражений, имеющих повторяющиеся подвыражения. Например, на рис. 5.19 показан ациклический орграф для выражения

$$((a + b) * c + ((a + b) + e) * (e + f)) * ((a + b) * c).$$

Подвыражения $a + b$ и $(a + b) * c$ встречаются в выражении несколько раз, поэтому они представлены вершинами, в которые входят несколько дуг.

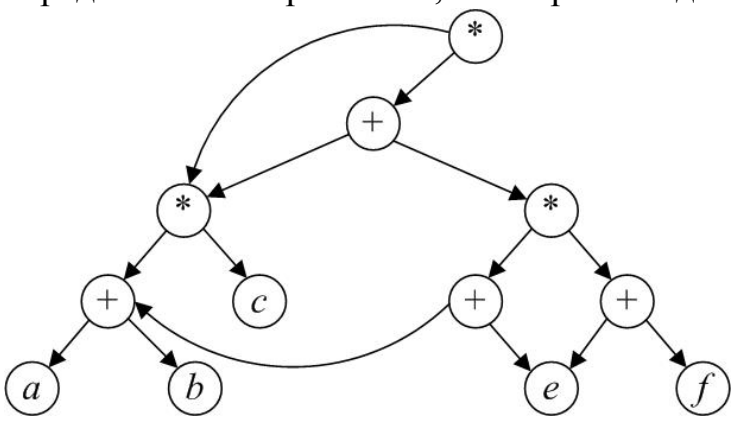


Рис. 5.19. Ориентированный ациклический граф для арифметического выражения

Ациклические орграфы также полезны для представления отношений частичных порядков. Отношением частичного порядка R, определенным на множестве S, называется бинарное отношение, для которого выполняются следующие условия:

1. Ни для какого элемента a из множества S не выполняется aRa (свойство антирефлексивности).

2. Для любых a, b, c из S, таких, что aRb и bRc, выполняется aRc (свойство транзитивности).

Двумя естественными примерами отношений частичного порядка могут служить отношение «меньше чем» (обозначается знаком «<»), заданное на множестве целых чисел, и отношение строгого включения ($\subset$) множеств.

Например, пусть $S = \{1, 2, 3\}$ и $P(S)$ – множество всех подмножеств множества S, т. е. $P(S) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Отношение строгого включения ($\subset$) является отношением частичного порядка на множестве $P(S)$. Очевидно, включение $A \subset A$ не выполняется для любого множества A из P (антирефлексивность), и при выполнении $A \subset B$ и $B \subset C$ выполняется $A \subset C$ (транзитивность).

Ациклические орграфы можно использовать для графического изображения отношения частичного порядка между какими-либо объектами. Для начала можно представить отношение R как одноименное множество пар (дуг) таких, что пара (a, b) принадлежит этому множеству только тогда, когда выполняется aRb. Если отношение R определено на множестве элементов S, то ориентированный граф $G = (S, R)$ будет ациклическим. И наоборот, пусть $G = (S, R)$ является ациклическим орграфом, а R^+ – отношение, определенное следующим образом: aR^+b выполняется тогда и только тогда, когда существует путь (длиной не менее 1) от вершины a к вершине b. Тогда R^+ – отношение частичного порядка на S. (Отношение R^+ является транзитивным замыканием отношения R.)

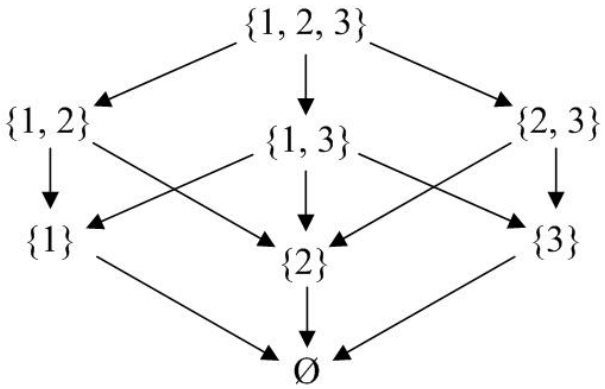


Рис. 5.20. Ациклический граф, представляющий отношение строгого включения

В качестве примера на рис. 5.20 показан ациклический орграф $(P(S), R)$, где $S = \{1, 2, 3\}$. Здесь R^+ – отношение строгого включения на множестве $P(S)$.

Проверка ацикличности орграфа

Предположим, что есть ориентированный граф $G = (V, E)$. Необходимо определить, является ли он ациклическим, т. е. имеет ли он циклы. Чтобы ответить на этот вопрос, можно использовать метод поиска в глубину. Если при обходе орграфа G методом поиска в глубину встретится обратная дуга, то ясно, что граф имеет цикл. С другой стороны, если в орграфе есть цикл, тогда обратная дуга обязательно встретится при обходе этого орграфа методом поиска в глубину.

Чтобы доказать это, предположим, что орграф G имеет цикл (рис. 5.21).

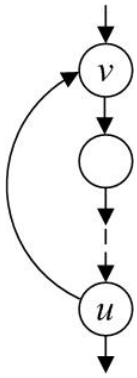


Рис. 5.21. Ациклический орграф, представляющий структуру предшествований

Пусть при обходе данного орграфа методом поиска в глубину вершина v имеет наименьшее глубинное число среди всех вершин, составляющих цикл. Рассмотрим дугу $u \rightarrow v$, принадлежащую этому циклу. Поскольку вершина u входит в цикл, то она должна быть потомком вершины v в глубинном остовном лесу. Поэтому дуга $u \rightarrow v$ не может быть поперечной дугой. Так как глубинный номер вершины u больше глубинного номера вершины v, то отсюда следует, что эта дуга не может быть дугой дерева и прямой дугой. Следовательно, дуга $u \rightarrow v$ является обратной дугой, как показано на рис. 5.21.

Топологическая сортировка

Большие проекты часто разбиваются на совокупность меньших задач, которые выполняются в определенном порядке и обеспечивают полное завершение целого проекта. Например, план учебного заведения требует определенной последовательности в чтении учебных курсов. Ациклические орграфы могут служить естественной моделью для таких ситуаций. Например, когда чтение учебного курса C предшествует чтению учебного курса D, создается дуга от курса C к курсу D.

Чтобы проиллюстрировать этот пример приведем ациклический орграф (рис. 5.22), представляющий структуру предшествований пяти учебных курсов.

Чтение учебного курса *C*3, например, требует предварительного чтения курсов *C*1 и *C*2.

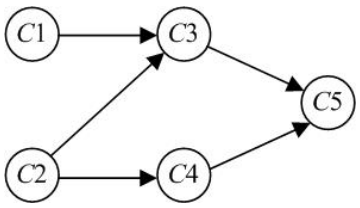


Рис. 5.22. Ациклический орграф, представляющий структуру предшествований

Топологическая сортировка – это процесс линейного упорядочивания вершин ациклического орграфа таким образом, что если существует дуга от вершины *i* к вершине *j*, то в упорядоченном списке вершин орграфа вершина *i* предшествует вершине *j*. Например, для орграфа из рис. 5.22 вершины после топологической сортировки расположатся в следующем порядке: *C*1, *C*2, *C*3, *C*4, *C*5.

Топологическую сортировку легко выполнить с помощью модифицированной процедуры поиска в глубину, если в ней после строки (4) добавить оператор вывода на печать:

```
procedure topsort (v: вершина);
                        {печать в обратном топологическом порядке вершин,
var                                  достижимых из вершины v}
    w: вершина;
begin
    mark[v] := visited;
    for каждая вершина w из списка L[v] do
        if mark[w] = unvisited then
            topsort (w);
    writeln (v)
end;
```

Когда процедура *topsort* заканчивает поиск всех вершин, являющихся потомками вершины *v*, печатается сама вершина *v*. Отсюда следует, что процедура *topsort* распечатывает все вершины в обратном топологическом порядке.

Эта процедура работает вследствие того, что в ациклическом орграфе нет обратных дуг. Рассмотрим, что происходит, когда поиск в глубину достигает конечной вершины *x*. Из любой вершины могут исходить только дуги дерева, прямые и поперечные дуги. Но все эти дуги направлены в вершины, которые уже пройдены (посещались до достижения вершины *x*), и поэтому предшествуют вершине *x*.

5.7. Сильная связность

Сильно связной компонентой ориентированного графа называется максимальное множество вершин, в котором существуют пути из любой вершины в любую другую вершину. Метод поиска в глубину можно эффективно использовать для нахождения сильно связных компонентов ориентированного графа.

Дадим точную формулировку понятия сильной связности. Пусть $G = (V, E)$ – ориентированный граф. Множество вершин V разбивается на классы эквивалентности V_i, $1 \le i \le r$, так, что вершины v и w будут эквивалентны тогда и только тогда, когда существуют пути из вершины v в вершину w и из вершины w в вершину v. Пусть E_i, $1 \le i \le r$, – множество дуг, которые начинаются и заканчиваются в множестве вершин V_i. Тогда графы $G_i = (V_i, E_i)$ называются сильно связными компонентами графа G. Ориентированный граф, состоящий только из одной сильно связной компоненты, называется сильно связным.

На рис. 5.23 представлен ориентированный граф с двумя сильно связными компонентами. Эти сильно связные компоненты показаны на рис. 5.24.

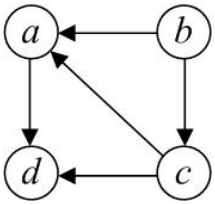


Рис. 5.23. Ориентированный граф

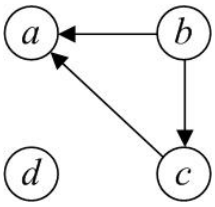


Рис. 5.24. Сильно связные компоненты орграфа из рис. 5.23

Отметим, что каждая вершина ориентированного графа G принадлежит какой-либо сильно связной компоненте, но некоторые дуги могут не принадлежать никакой сильно связной компоненте. Такие дуги идут от вершины одной компоненты к вершине, принадлежащей другой компоненте. Можно представить связи между компонентами путем создания редуцированного (приведенного) графа для графа G. Вершинами приведенного графа являются сильно связные компоненты графа G. В приведенном графе дуга от вершины C к вершине D существует только тогда, когда в графе G есть дуга от какой-

либо вершины, принадлежащей компоненте C, к какой-либо вершине, принадлежащей компоненте D. Редуцированный граф всегда является ациклическим орграфом, поскольку если бы существовал цикл, то все компоненты, входящие в этот цикл, в действительности были бы одной связной компонентой. На рис. 5.25 показан редуцированный граф для графа из рис. 5.23.

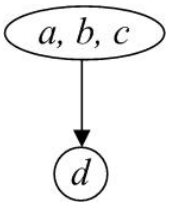


Рис. 5.25. Редуцированный граф

Теперь рассмотрим алгоритм нахождения сильно связных компонент для заданного ориентированного графа G.

1. Сначала выполняется поиск в глубину на графе G. Вершины нумеруются в порядке завершения рекурсивно вызванной процедуры *dfs*, т. е. присвоение номеров вершинам происходит после строки (4) в листинге процедуры поиска в глубину.

2. Конструируется новый ориентированный граф G_r путем обращения направления всех дуг графа G.

3. Выполняется поиск в глубину на графе G_r, начиная с вершины с наибольшим номером, присвоенным на шаге 1. Если проведенный поиск не охватывает всех вершин, то начинается новый поиск с вершины, имеющей наибольший номер среди оставшихся вершин.

4. Каждое дерево в полученном остовном лесу является сильно связной компонентой графа G.

Применим описанный алгоритм к ориентированному графу, представленному на рис. 5.23. Прохождение вершин графа начнем с вершины a и затем перейдем на вершину b. Присвоенные после завершения шага 1 номера вершин показаны на рис. 5.26. Полученный путем обращения дуг граф G_r представлен на рис. 5.27.

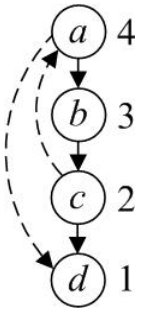


Рис. 5.26. Номера вершин после выполнения шага 1 алгоритма

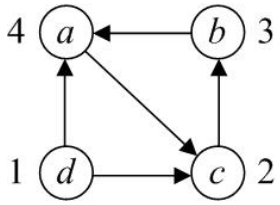


Рис. 5.27. Граф G_r

Выполнив поиск в глубину на графе G_r, получим глубинный остовный лес, показанный на рис. 5.28. Обход остовного леса начинается с вершины *a*, принимаемой в качестве корня дерева, так как она имеет самый высокий номер. Из корня *a* можно достигнуть только вершины *c* и *b*. Следующее дерево имеет только корень *d*, поскольку вершина *d* имеет самый высокий номер среди оставшихся (но она и единственная оставшаяся вершина). Каждое из этих деревьев формирует отдельную сильно связанную компоненту исходного графа *G*.

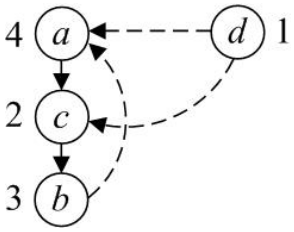


Рис. 5.28. Глубинный остовный лес

Выше утверждалось, что вершины строго связной компоненты в точности соответствуют вершинам дерева в остовном лесу, полученном при применении поиска в глубину к графу G_r. Докажем это.

Сначала покажем, что если вершины *v* и *w* принадлежат одной связной компоненте, то они принадлежат и одному остовному дереву. Если вершины *v* и *w* принадлежат одной связной компоненте, то в графе *G* существует путь от вершины *v* к вершине *w* и путь от вершины *w* к вершине *v*. Допустим, что поиск в глубину на графе G_r начат от некоторого корня *x* и достиг или вершины *v*, или вершины *w*. Поскольку существуют пути (в обе стороны) между вершинами *v* и *w*, то обе они принадлежат остовному дереву с корнем *x*.

Теперь предположим, что вершины *v* и *w* принадлежат одному и тому же остовному дереву в глубинном остовном лесу графа *G*. Покажем, что они принадлежат одной и той же сильно связной компоненте. Пусть *x* – корень остовного дерева, которому принадлежат вершины *v* и *w*. Поскольку вершина *v* является потомком вершины *x*, то в графе G_r существует путь от вершины *x* к вершине *v*, а в графе *G* – путь от вершины *v* к вершине *x*.

В соответствии с прохождением вершин графа *Gr* методом поиска в глубину вершина *v* посещается позднее, чем вершина *x*, т. е. вершина *x* имеет более высокий номер, чем вершина *v*. Поэтому при обходе графа *G* рекурсивный вызов процедуры *dfs* для вершины *v* заканчивается раньше, чем вызов *dfs* для вершины *x*. Но если при обходе графа *G* вызывается процедура *dfs* для вершины *v*, то из-за наличия пути от вершины *v* к вершине *x* процедура *dfs* для вершины *x* начнется и закончится до окончания процедуры *dfs*(*v*), и, следовательно, вершина *x* получит меньший номер, чем вершина *v*. Получаем противоречие.

Отсюда заключаем, что при обходе графа *G* вершина *v* посещается при выполнении *dfs*(*x*) и поэтому вершина *v* является потомком вершины *x*. Следовательно, в графе *G* существует путь от вершины *x к* вершине *v*. Более того, так как существует и путь от вершины *v* к вершине *x*, то вершины *x* и *v* принадлежат одной сильно связной компоненте. Аналогично доказывается, что вершины *x* и *w* принадлежат одной сильно связной компоненте. Отсюда вытекает, что и вершины *v* и *w* принадлежат той же сильно связной компоненте, так как существует путь от вершины *v* к вершине *w* через вершину *x* и путь от вершины *w* к вершине *v* через вершину *x*.

6. НЕОРИЕНТИРОВАННЫЕ ГРАФЫ

Неориентированный граф $G = (V, E)$ состоит из конечного множества вершин V и множества ребер E. В отличие от ориентированного графа, здесь каждое ребро (v, w) соответствует *неупорядоченной* паре вершин[1]: если (v, w) – неориентированное ребро, то $(v, w) = (w, v)$. Далее неориентированный граф будем называть просто графом.

Графы широко используются в различных областях науки и техники для моделирования симметричных отношений между объектами. Объекты соответствуют вершинам графа, а ребра – отношениям между объектами [1]. В этой главе будут описаны различные структуры данных, которые применимы для представления графов. Также будут рассмотрены алгоритмы решения трех типовых задач, возникающих при работе с графами: построения минимальных остовных деревьев, определения двусвязных компонент и нахождения максимального паросочетания графа.

6.1. Основные определения

Многое из терминологии ориентированных графов применимо к неориентированным графам. Например, вершины v и w называются смежными, если существует ребро (v, w). Будем также говорить, что ребро (v, w) инцидентно вершинам v и w.

Путем называется такая последовательность вершин $v_1, v_2, \ldots, v_n$, что для всех i, $1 \le i < n$, существуют ребра (v_i, и, v_{i+1}). Путь называется простым, если все вершины пути различны, за исключением, возможно, вершин v_i и v_n. Длина пути равна количеству ребер, составляющих путь, т. е. длина равна $n - 1$ для пути из n вершин. Если для вершин v_i и v_n существует путь $v_1, v_2, \ldots, v_n$, то эти вершины называются связанными. Граф называется связным, если в нем любая пара вершин связанная.

Пусть есть граф G = (V, E) с множеством вершин V и множеством ребер E. Граф $G' = (V', E')$ называется подграфом графа G, если

1. множество V' является подмножеством множества V;
2. множество E' состоит из ребер (v, w) множества E таких, что обе вершины v и w принадлежат V'.

Если множество E' состоит из всех ребер (v, w) множества E таких, что обе вершины v и w принадлежат V', то в этом случае граф G' называется индуцированным подграфом графа G.

На рис. 6.1, *а* показан граф $G = (V, E)$ с множеством вершин $V = \{a, b, c, d\}$ и множеством дуг $E = \{(a, b), (a, d), (b, c), (b, d), (c, d)\}$. На рис. 6.1, *б*

[1]Пока не будет сказано другое, будем считать, что ребро всегда соответствует паре различных вершин.

представлен один из его индуцированных подграфов, заданный множеством вершин $V = \{a, b, c\}$ и содержащий все ребра, не инцидентные вершине d.

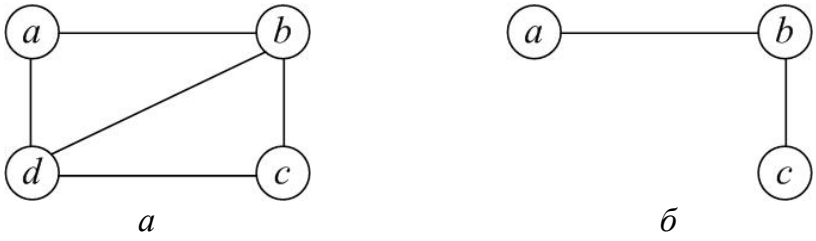


Рис. 6.1. Граф и один из его подграфов

Связной компонентой графа G называется максимальный связный индуцированный подграф графа G.

Граф, показанный на рис. 6.1, *а*, является связным графом и имеет только одну связную компоненту, а именно – самого себя. На рис. 6.2 представлен несвязный граф, состоящий из двух связных компонент.

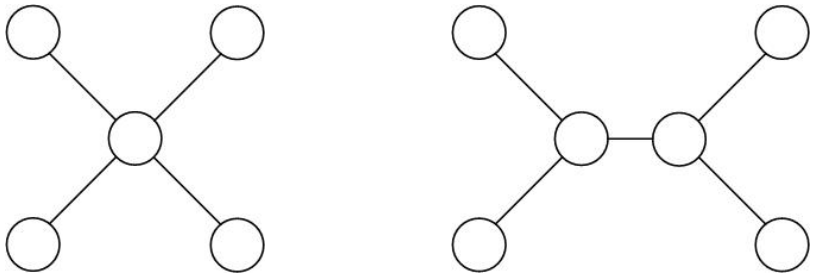


Рис. 6.2. Несвязный граф

Циклом (простым) называется путь (простой) длины не менее 3 от какой-либо вершины до нее самой. Мы не считаем циклами пути длиной 0, длиной 1 (петля от вершины v к ней самой) и длиной 2 (путь вида v, w, v). Граф называется циклическим, если имеет хотя бы один цикл. Связный ациклический граф, представляющий собой «дерево без корня», называют свободным деревом. На рис. 6.2 показан граф, состоящий из двух связных компонент, каждая из которых является свободным деревом. Свободное дерево можно сделать «обычным» деревом, если какую-либо вершину назначить корнем, а все ребра сориентировать в направление от этого корня.

Свободные деревья имеют два важных свойства, которые будут использоваться в следующих разделах:

1. Каждое свободное дерево с числом вершин n, $n \geq 1$, имеет в точности $n - 1$ ребер.

2. Если в свободное дерево добавить новое ребро, то обязательно получится цикл.

Докажем первое утверждение методом индукции по n. Очевидно, что утверждение справедливо для $n = 1$, поскольку в этом случае имеем только одну вершину и не имеем ребер. Пусть утверждение 1 справедливо для свободного дерева с $n - 1$ вершинами. Рассмотрим дерево G с n вершинами.

Сначала покажем, что в свободном дереве существуют вершины, имеющие одно инцидентное ребро. Заметим, что G не содержит изолированных вершин (т. е. вершин, не имеющих инцидентных ребер), иначе граф G не был бы связным. Теперь создадим путь от некоторой вершины v_1, следуя по произвольному ребру, инцидентному вершине v_1. На каждом шаге построения этого пути, достигнув какой-либо вершины, выбираем инцидентное ей ребро, которое ведет к вершине, еще не участвовавшей в формировании пути. Пусть таким способом построен путь $v_1, v_2, ..., v_k$. Вершина v_k будет смежной либо с одной вершиной v_{k-1}, либо еще с какой-нибудь, не входящей в построенный путь (иначе получится цикл). В первом случае получаем, что вершина v_k имеет только одно инцидентное ей ребро, и значит, наше утверждение о том, что в свободном дереве существуют вершины, имеющие одно инцидентное ребро, будет доказано. Во втором случае обозначим через v_{k+1} вершину, смежную с вершиной v_k и строим путь $v_1, v_2, ... , v_k, v_{k+1}$. В этом пути все вершины различны (иначе опять получится цикл). Так как количество вершин конечно, то этот процесс закончится за конечное число шагом и мы найдем вершину, имеющую одно инцидентное ребро. Обозначим такую вершину через v, а инцидентное ей ребро через (v, w).

Теперь рассмотрим граф G', полученный в результате удаления из графа G вершины v и ребра (v, w). Граф G' имеет $n - 1$ вершину, для него выполняется утверждение 1 и поэтому он имеет $n - 2$ ребра. Но граф G имеет на одну вершину и на одно ребро больше, чем граф G', поэтому для него также выполняется утверждение 1. Следовательно, утверждение 1 доказано.

Теперь докажем утверждение 2 о том, что добавление ребра в свободное дерево формирует цикл. Применим доказательство от противного, т. е. предположим, что добавление ребра в свободное дерево не формирует цикл. Итак, после добавления нового ребра получаем граф с n вершинами и n ребрами. Этот граф остался связным и, по нашему предположению, ацикличным. Следовательно, этот граф – свободное дерево. Но в таком случае получаем противоречие с утверждением 1. Отсюда вытекает справедливость утверждения 2.

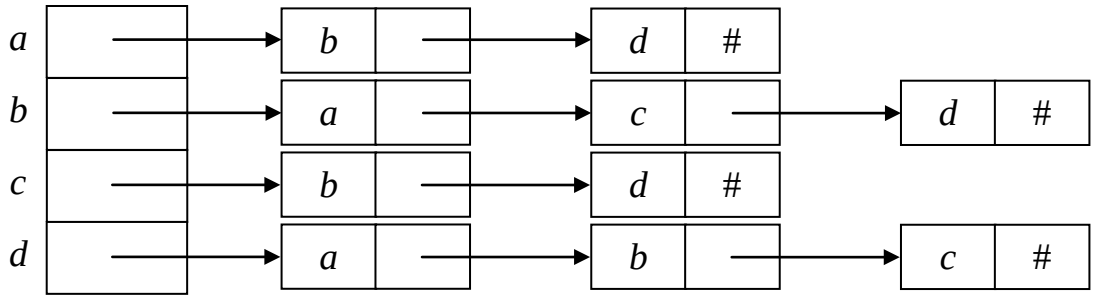
Представление неориентированных графов

Для представления неориентированных графов можно применять те же методы, что и для представления ориентированных графов, если неориентированное ребро между вершинами v и w рассматривать как две ориентированных дуги от вершины v к вершине w и от вершины w к вершине v.

На рис. 6.3 показаны матрица смежности и списки смежности, представляющие граф, приведенный на рис. 6.1, а.

	a	*b*	*c*	*d*
a	0	1	0	1
b	1	0	1	1
c	0	1	0	1
d	1	1	1	0

а



б

Рис. 6.3. Представления неориентированного графа:
а – матрица смежности, *б* – списки смежности

Очевидно, что матрица смежности для неориентированного графа симметрична. Отметим, что в случае представления графа посредством списков смежности для существующего ребра (*i*, *j*) в списке смежности вершины *i* присутствует вершина *j*, а в списке смежности вершины *j* – вершина *i*.

6.2. Остовные деревья минимальной стоимости

Пусть G = (*V*, *E*) – связный граф, в котором каждое ребро (*v*, *w*) помечено числом *c*(*v*, *w*), которое называется стоимостью ребра. Остовным деревом графа *G* называется свободное дерево, содержащее все вершины *V* графа *G*. Стоимость остовного дерева вычисляется как сумма стоимостей всех ребер, входящих в это дерево. Далее будут рассмотрены методы нахождения остовных деревьев минимальной стоимости. На рис. 6.4 показаны граф с помеченными ребрами и его остовное дерево минимальной стоимости.

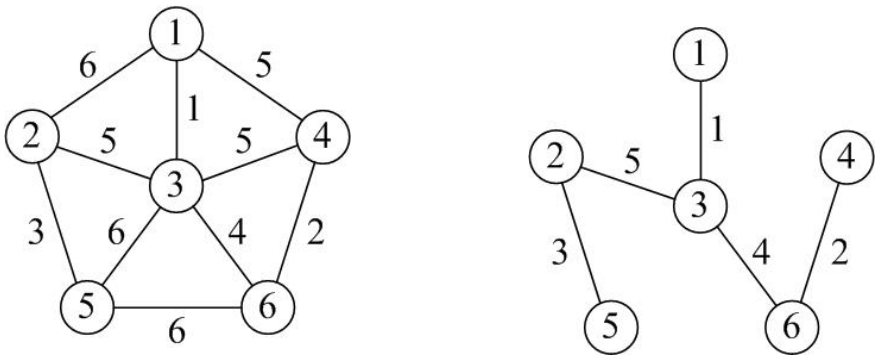


Рис. 6.4. Граф и его остовное дерево минимальной стоимости

Типичное применение остовных деревьев минимальной стоимости можно найти при разработке коммуникационных сетей. Здесь вершины графа представляют города, ребра – возможные коммуникационные, линии между городами, а стоимость ребер соответствует стоимости коммуникационных линий. В этом случае остовное дерево минимальной стоимости представляет коммуникационную сеть, объединяющую все города коммуникационными линиями минимальной стоимости.

Свойство остовных деревьев минимальной стоимости

Существуют различные методы построения остовных деревьев минимальной стоимости. Многие из них основываются на следующем свойстве остовных деревьев минимальной стоимости, которое для краткости будем называть свойством ОДМС. Пусть $G = (V, E)$ – связный граф с заданной функцией стоимости, определенной на множестве ребер. Обозначим через U подмножество множества вершин V. Если (u, v) – такое ребро наименьшей стоимости, что $u \in U$ и $v \in V \setminus U$, тогда для графа G существует остовное дерево минимальной стоимости, содержащее ребро (u, v).

Доказать это свойство нетрудно. Допустим противное: существует остовное дерево для графа G, обозначим его $T = (V, E')$, содержащее множество U и не содержащее ребро (u, v), стоимость которого меньше любого остовного дерева для G, содержащего ребро (u, v).

Поскольку дерево T – свободное дерево, то из второго свойства свободных деревьев следует, что добавление ребра (u, v) к этому дереву приведет к образованию цикла. Этот цикл содержит ребро (u, v) и будет содержать другое ребро (u', v') такое, что $u' \in U$ и $v' \in V \setminus U$, как показано на рис. 6.5.

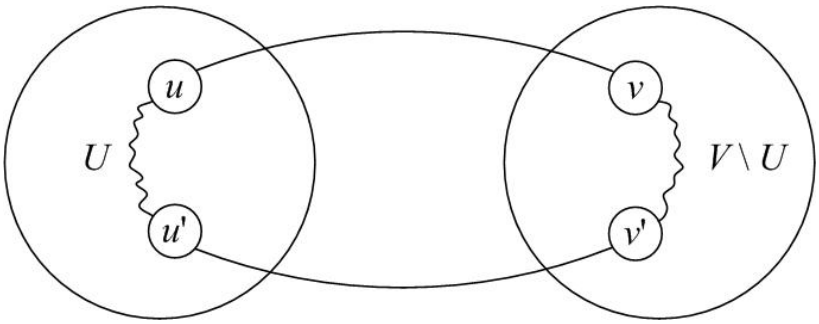


Рис. 6.5. Цикл в остовном дереве

Удаление ребра (u', v') приведет к разрыву цикла и образованию остовного дерева, чья стоимость будет не выше стоимости дерева T, поскольку $c(u, v) \le c(u', v')$. Приходим к противоречию с предположением, что остовное дерево T – это дерево минимальной стоимости.

Алгоритм Прима

Существуют два популярных метода построения остовного дерева минимальной стоимости для помеченного графа $G = (V, E)$, основанные на свойстве ОДМС. Один такой метод известен как алгоритм Прима (*Prim*). В этом алгоритме строится множество вершин U, из которого «вырастает» остовное дерево. Пусть $V = \{1, 2, \ldots, n\}$. Сначала $U = \{1\}$. На каждом шаге алгоритма находится ребро наименьшей стоимости (u, v) такое, что $u \in U$ и $v \in V \setminus U$, затем вершина v переносится из множества $V \setminus U$ в множество U. Этот процесс продолжается до тех пор, пока множество U не станет равным множеству V. Процесс построения остовного дерева для графа из рис. 6.4, *а* показан на рис. 6.6, эскиз же алгоритма следующий:

```
procedure Prim (G: граф; var T: множество ребер) ;
var U: множество вершин;
    u, v: вершина;
begin
    T := 0;   U := {1};
    while U ≠ V do begin
        нахождение ребра (u, v) наименьшей стоимости и такого,
                                что u ∈ U и v ∈ V \ U
        T := T ∪ {(u, v)};
        U := U ∪ {v}
    end
end;
```

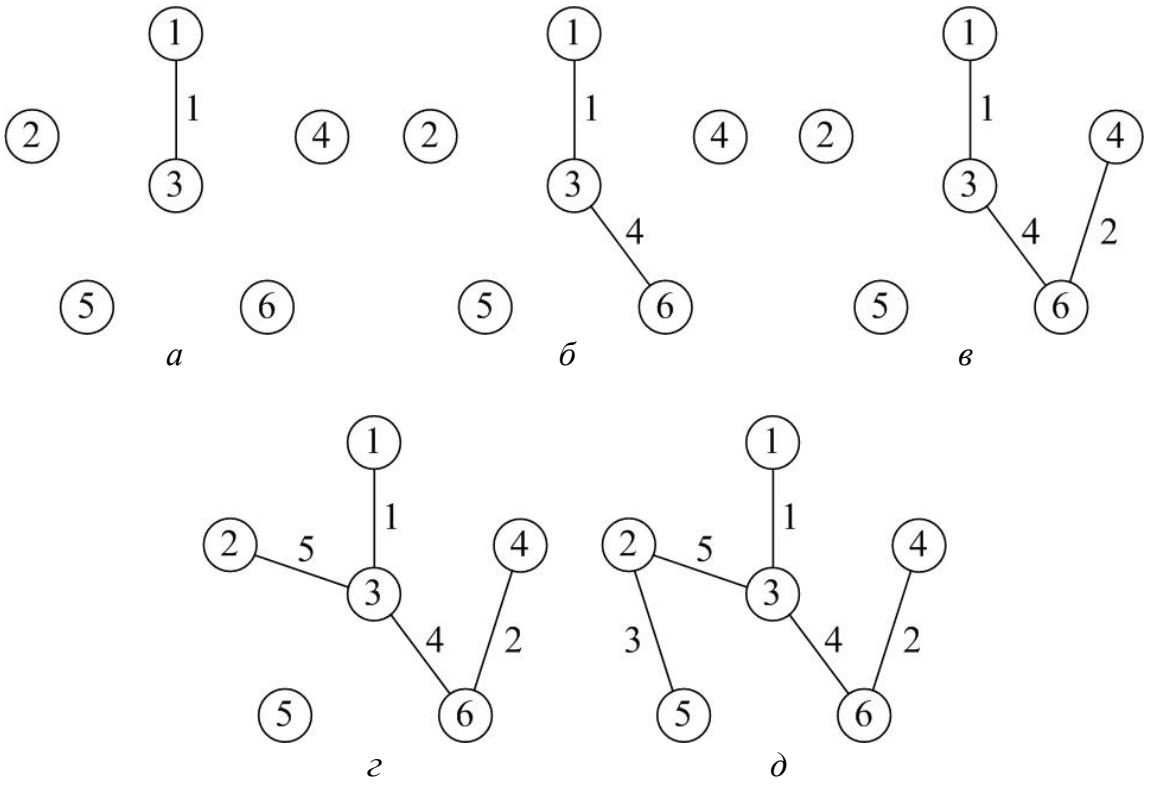


Рис. 6.6. Последовательность построения остовного дерева алгоритмом Прима

Если ввести два массива, то можно сравнительно просто организовать на каждом шаге алгоритма выбор ребра с наименьшей стоимостью, соединяющего множества U и $V \setminus U$. Массив $CLOSEST[i]$ для каждой вершины i из множества $V \setminus U$ содержит вершину из U, с которой он соединен ребром минимальной стоимости (это ребро выбирается среди ребер, инцидентных вершине i, и которые ведут в множество U). Другой массив $LOWCOST[i]$ хранит значение стоимости ребра $(i, CLOSEST[i])$.

На каждом шаге алгоритма просматривается массив $LOWCOST$, находится минимальное значение $LOWCOST[k]$. Вершина k принадлежит множеству $V \setminus U$ и соединена ребром с вершиной из множества U. Затем выводится на печать ребро $(k, CLOSEST[k])$. Так как вершина k присоединяется к множеству U, то вследствие этого изменяются массивы $LOWCOST$ и $CLOSEST$. На вход алгоритма Прима поступает массив C размера n x n, чьи элементы $C[i, j]$ равны стоимости ребер (i, j). Если ребра (i, j) не существуют, то элемент $C[i, j]$ полагается равным некоторому достаточно большому числу.

После нахождения очередной вершины k остовного дерева $LOWCOST[k]$ приравнивается значению равному *infinity* (бесконечность), очень большому числу, такому, чтобы эта вершина уже в дальнейшем не рассматривалась. Значение числа *infinity* должно быть больше стоимости любого ребра графа.

Время выполнения алгоритма Прима имеет порядок $O(n^2)$. Если значение n достаточно большое, то использование этого алгоритма не рационально. Далее рассматривается алгоритм Крускала нахождения остовного дерева минимальной стоимости, который выполняется за время порядка $O(e \log e)$, где

e – количество ребер в данном графе. Если e значительно меньше n^2, то алгоритм Крускала предпочтительнее, но если e близко к n^2, рекомендуется применять алгоритм Прима.

Алгоритм Крускала

Снова предположим, что есть связный граф G = (V, E) с множеством вершин V = {1, 2, ..., n} и функцией стоимости c, определенной на множестве ребер E. В алгоритме Крускала (*Kruskal*) построение остовного дерева минимальной стоимости для графа G начинается с графа T = (V, ∅), состоящего только из n вершин графа G и не имеющего ребер. Таким образом, каждая вершина является связной (с самой собой) компонентой. В процессе выполнения алгоритма имеем набор связных компонент, постепенно объединяя которые формируем остовное дерево.

При построении связных, постепенно возрастающих компонент поочередно проверяются ребра из множества E в порядке возрастания их стоимости. Если очередное ребро связывает две вершины из разных компонент, тогда оно добавляется в граф T. Если это ребро связывает две вершины из одной компоненты, то оно отбрасывается, так как его добавление в связную компоненту, являющуюся свободным деревом, приведет к образованию цикла. Когда все вершины графа G будут принадлежать одной компоненте, построение остовного дерева минимальной стоимости T для этого графа заканчивается.

Рассмотрим помеченный граф, представленный на рис. 6.4, *а*. Последовательность добавления ребер в формирующееся дерево T показана на рис. 6.7. Ребра стоимостью 1, 2, 3 и 4 рассмотрены первыми и все включены в T, поскольку их добавление не приводит к циклам. Ребра (1, 4) и (3, 4) стоимостью 5 нельзя включить в T, так как они соединяют вершины одной и той же компоненты (рис. 6.7, *г*) и поэтому замыкают цикл. Но оставшееся ребро (2, 3) также стоимостью 5 не создает цикл. После включения этого ребра в T формирование остовного дерева завершается.

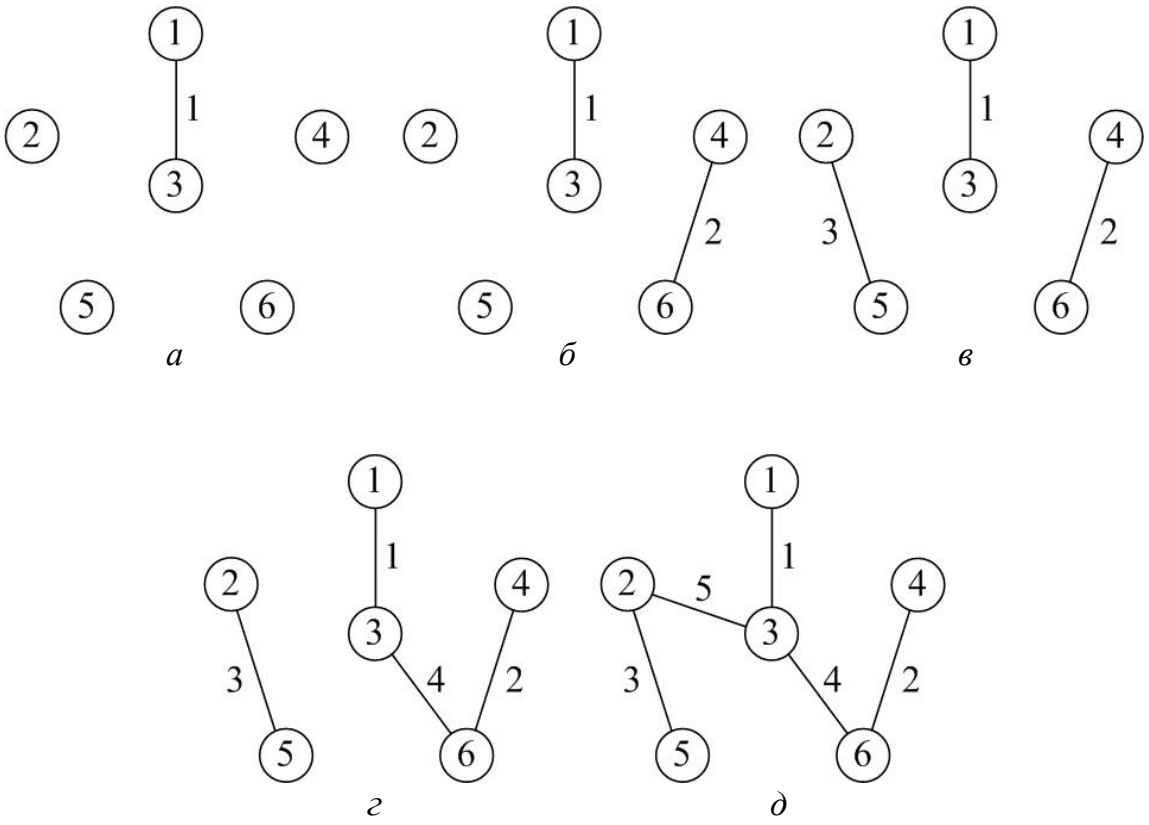


Рис. 6.7. Последовательное формирование остовного дерева минимальной стоимости посредством алгоритма Крускала

Этот алгоритм можно реализовать, основываясь на множествах (для вершин и ребер) и операторах работы с ними. Прежде всего, необходимо множество ребер *E*, к которому можно было бы последовательно применять оператор *DELETEMIN* для отбора ребер в порядке возрастания их стоимости. Поэтому множество ребер целесообразно представить в виде очереди с приоритетами и использовать для нее частично упорядоченное дерево в качестве структуры данных.

Необходимо также поддерживать набор *C* связных компонент, для чего можно использовать следующие операторы:

1. Оператор *MERGE*(*A*, *B*, *C*) объединяет компоненты *A* и *B* из набора *C* и результат объединения помещает или в *A*, или в *B*.

2. Функция *FIND*(*v*, *C*) возвращает имя той компоненты из набора *C*, которая содержит вершину *v*.

3. Оператор *INITIAL*(*A*, *v*, *C*) создает в наборе *C* новую компоненту с именем *A*, содержащую только одну вершину *v*.

Введем для множеств абстрактный тип данных *MFSET*, поддерживающий операторы *MERGE* и *FIND*. Будем использовать этот тип данных в эскизе программы, реализующей алгоритм Крускала:

```
procedure Kruskal (V: SET; E: SET; var T: SET ) ;
```

```
{V – множество вершин, E и T — множества дуг}
var
    ncomp: integer; {текущее количество компонент}
    edges: PRIORITYQUEUE;
            {множество дуг, реализованное как очередь с приоритетами}
     components: SET;
            {множество V, сгруппированное во множество компонент}
    u, v: вершина;
    e: ребро;
    nextcomp: integer; {имя (номер) новой компоненты}
    ucomp, vcomp: integer; {имена (номера) компонент}

begin
     MAKENULL(T);
     MAKENULL(edges);
     nextcomp := 0;
     ncomp := число элементов множества V;
     for v ∈ V do begin   {инициализация компонент,
                           содержащих по одной вершине из V}
          nextcomp := nextcomp + 1;
          INITIAL(nextcomp, v, components)
      end;
     for e ∈ E do     {инициализация очереди с приоритетами,
                       содержащей ребра }
          INSERT(e, edges);
     while ncomp > 1 do begin {рассматривается следующее ребро}
          e := DELETEMIN(edges);
          пусть e = (u, v);
          ucomp := FIND(u, components);
          vcomp := FIND(v, components);
          if ucomp <> vcomp then begin
                           {ребро e соединяет две различные компоненты}
               MERGE(ucomp, vcomp, components);
               ncomp := ncomp – 1;
               INSERT(e, T)
          end
     end
end;
```

Время выполнения этой программы зависит от двух факторов. Если в исходном графе G всего e ребер, то для вставки их в очередь с приоритетами потребуется время порядка $O(e \log e)$. Каждая итерация цикла *while* для нахождения ребра с наименьшей стоимостью в очереди *edges* требует времени порядка $O(\log e)$. Поэтому выполнение всего этого цикла в самом худшем

случае потребует времени $O(e \log e)$. Вторым фактором, влияющим на время выполнения программы, является общее время выполнения операторов *MERGE* и *FIND*, которое зависит от метода реализации абстрактный тип данных *MFSET*. В любом случае алгоритм Крускала может быть выполнен за время $O(e \log e)$.

6.3. Обход неориентированных графов

Во многих задачах, связанных с графами, требуется организовать систематический обход всех вершин графа. Существуют два наиболее часто используемых метода обхода графов: поиск в глубину и поиск в ширину, о которых речь пойдет в этом разделе. Оба этих метода можно эффективно использовать для поиска вершин, смежных с данной вершиной.

Поиск в глубину

Напомним, что в параграфе 5.5 был построен алгоритм *dfs* для обхода вершин ориентированного графа. Этот же алгоритм можно использовать для обхода вершин и неориентированных графов, поскольку неориентированное ребро (v, w) можно представить в виде пары ориентированных дуг $v \rightarrow w$ и $w \rightarrow v$.

Фактически построить глубинный остовный лес для неориентированного графа (т. е. совершить обход его вершин) даже проще, чем для ориентированного. Во-первых, заметим, что каждое дерево остовного леса соответствует одной связной компоненте исходного графа, поэтому, если граф связный, его глубинный остовный лес будет состоять только из одного дерева. Во-вторых, при построении остовного леса для орграфа мы различали четыре типа дуг: дуги дерева, передние, обратные и поперечные дуги, а для неориентированного графа в этой ситуации выделяют только два типа ребер: ребра дерева и обратные ребра. Так как для неориентированного графа не существует направления ребер, то, естественно, прямые и обратные ребра здесь не различаются и объединены общим названием обратные ребра. Аналогично, так как в остовном дереве для неориентированного графа вершины не делятся на потомков и предков, то нет и перекрестных ребер. В самом деле, пусть есть ребро (v, w) и предположим, что при обходе графа вершина v достигнута раньше, чем вершина w. Тогда процедура $dfs(v)$ не может закончиться раньше, чем будет рассмотрена вершина w. Поэтому в остовном дереве вершину w можно считать потомком вершины v. Но подобным образом, если сначала вызывается $dfs(w)$, вершина v становится потомком вершины w.

Итак, при обходе вершин неориентированного графа методом поиска в глубину все ребра делятся на следующие группы:

1. Ребра дерева – это такие ребра (v, w), что при обходе графа процедура *dfs*(v) вызывается непосредственно перед процедурой *dfs*(w) или, наоборот, сначала вызывается процедура *dfs*(w), а затем сразу процедура *dfs*(v).

2. Обратные ребра – такие ребра (v, w), что ни процедура *dfs*(w) не следует непосредственно за процедурой *dfs*(v), ни процедура *dfs*(v) не следует непосредственно за процедурой *dfs*(w) (т. е. между вызовами этих процедур следуют вызовы нескольких других процедур *dfs*(x)).[1]

Рассмотрим связный граф G на рис. 6.8, *а*. Остовное дерево для этого графа, полученное методом поиска в глубину, показано на рис. 6.8, *б*. Поиск был начат с вершины a, ребра дерева изображены сплошными линиями, а обратные ребра – пунктирными. Изображение остовного дерева начато от корня, и сыновья каждой вершины рисовались слева направо в том порядке, в каком они впервые посещались в процедуре *dfs*.

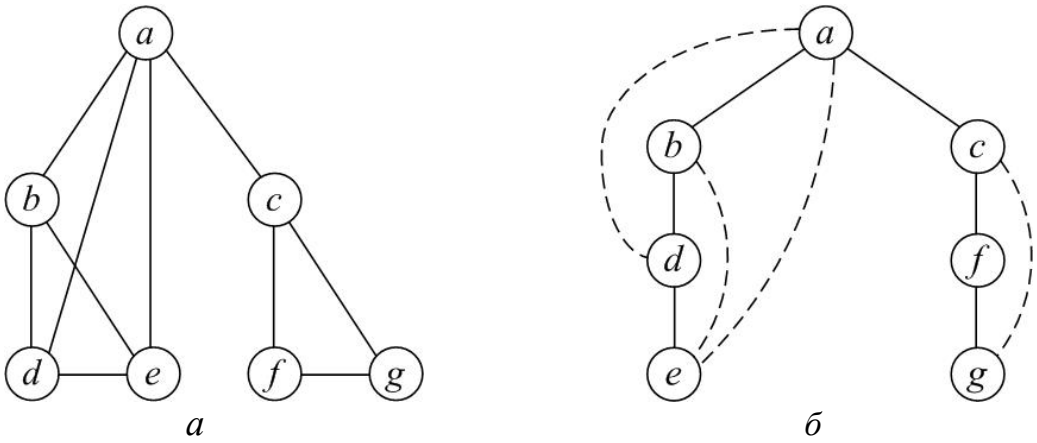


Рис. 6.8. Граф и остовное дерево, полученное при обходе его вершин методом поиска в глубину

Опишем несколько шагов поиска в глубину для данного графа. Процедура *dfs*(a) добавляет ребро (a, b) в остовное дерево T, поскольку вершина b ранее не посещалась, и вызывает процедуру *dfs*(b). Процедура *dfs*(b) добавляет ребро (b, d) в остовное дерево T и в свою очередь вызывает процедуру *dfs*(d). Далее процедура *dfs*(d) добавляет в остовное дерево ребро (d, e) и вызывает *dfs*(e). С вершиной e смежны вершины a, b и d, но они помечены как посещенные вершины. Поэтому процедура *dfs*(e) заканчивается без добавления ребер в дерево T. Процедура *dfs*(d) также находит среди вершин, смежных с вершиной d, только вершины, помеченные как «посещенные». Проце-

[1] Здесь приведены конструктивные определения ребер дерева и обратных ребер. Можно дать следующие, менее конструктивные, но более общие, определения: если при обходе графа достигнута вершина v, имеющая инцидентное ребро (v, w), то в случае, когда вершина w еще не посещалась во время этого обхода, данное ребро является ребром дерева, если же вершина w уже посещалась ранее, то ребро (v, w) – обратное ребро.

дура *dfs*(*d*) завершается без добавления новых ребер в остовное дерево. Процедура *dfs*(*b*) проверяет оставшиеся смежные вершины *a* и *e* (до этого была проверена только вершина *d*). Эти вершины посещались ранее, поэтому процедура *dfs*(*b*) заканчивается без добавления новых ребер в дерево *T*. Процедура *dfs*(*a*), продолжая работу, находит новые вершины, которые ранее не посещались. Это вершины *c*, *f* и *g*.

Поиск в ширину

Другой метод систематического обхода вершин графа называется поиском в ширину. Он получил свое название из-за того, что при достижении во время обхода любой вершины *v* далее рассматриваются все вершины, смежные с вершиной *v*, т. е. осуществляется просмотр вершин «в ширину». Этот метод также можно применить и к ориентированным графам.

Так же, как и при применении поиска вглубь, посредством метода поиска в ширину при обходе графа создается остовный лес. Если после достижения вершины *x* при рассмотрении ребра (*x*, *y*) вершина *y* не посещалась ранее, то это ребро считается ребром дерева. Если же вершина *y* уже посещалась ранее, то ребро (*x*, *y*) будет поперечным ребром, так как оно соединяет вершины, не связанные наследованием друг друга.

При выполнении алгоритма поиска в ширину, ребра дерева помещаются в первоначально пустой массив *T*, формирующий остовный лес. Посещенные вершины графа заносятся в очередь *Q*. Массив *mark* (метка) отслеживает состояние вершин: если вершина *v* пройдена, то элемент *mark*[*v*] принимает значение *visited* (посещалась), первоначально все элементы этого массива имеют значение *unvisited* (не посещалась). Отметим, что в этом алгоритме во избежание повторного помещения вершины в очередь пройденная вершина помечается как *visited* до помещения ее в очередь. Процедура поиска в ширину работает на одной связной компоненте. Если граф не односвязный, то эта процедура должна вызываться для вершин каждой связной компоненты. Эскиз процедуры *bfs*[1], реализующей алгоритм поиска в ширину представлен ниже.

```
procedure bfs (v);
        {bfs обходит все вершины, достижимые из вершины v}
var
    Q: QUEUE  {очередь для вершин}
    x, y: вершина;
begin
    mark[v]:= visited;
    ENQUEUE(v, Q);
    while not EMPTY(Q) do begin
        x := FRONT(Q) ;
```

[1] Название процедуры *bfs* является сокращением от *breadth-first search*, что обозначает «поиск в ширину».

```
            DEQUEUE(Q);
            for для каждой вершины y, смежной с вершиной x do
                  if mark[y] = unvisited then begin
                        mark[y] := visited;
                        ENQUEUE(y, Q);
                        INSERT((x, y) , T)
                  end
      end
end;
```

Остовное дерево для графа из рис. 6.8, *а*, построенное методом поиска в ширину, показано на рис. 6.9. Здесь обход графа начат с вершины *a*. Как и ранее, ребра дерева показаны сплошными линиями, а остальные ребра – пунктирными. Отметим также, что дерево нарисовано от корня, а все сыновья любой вершины располагаются слева направо в порядке их посещения.

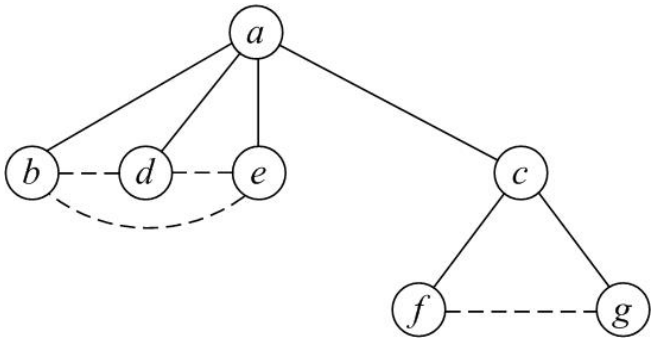


Рис. 6.9. Остовное дерево для графа из рис. 6.8, *а*, полученное методом поиска в ширину

Время выполнения алгоритма поиска в ширину такое же, как и для алгоритма поиска в глубину. Каждая пройденная вершина помещается в очередь только один раз, поэтому количество выполнении цикла *while* совпадает с количеством просмотренных вершин. Каждое ребро (x, y) просматривается дважды, один раз для вершины x и один раз для вершины y. Поэтому, если граф имеет n вершин и e ребер, а также если для представления ребер используются списки смежности, общее время обхода такого графа составит $O(\max(n, e))$. Поскольку обычно $e \geq n$, то получаем время выполнения алгоритма поиска в ширину порядка $O(e)$, т. е. такое же, как и для алгоритма поиска в глубину.

Метод поиска в ширину можно использовать для обнаружения циклов в произвольном графе, причем за время $O(n)$ (n – количество вершин графа), которое не зависит от количества ребер. Как показано в параграфе 6.1, граф с n вершинами и с n или большим количеством ребер обязательно имеет цикл. Однако если в графе $n - 1$ или меньше ребер, то цикл может быть только в том случае, когда граф состоит из двух или более связных компонент. Один

из способов нахождения циклов состоит в построении остовного леса методом поиска в ширину. Тогда каждое поперечное ребро (v, w) замыкает простой цикл, состоящий из ребер дерева, ведущих к вершинам v и w от общего предка этих вершин, как показано на рис. 6.10.

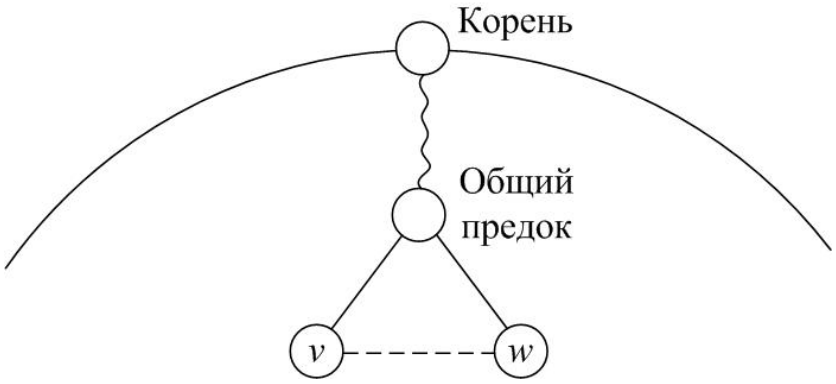


Рис. 6.10. Цикл, найденный методом поиска в ширину

Методы поисков в глубину и в ширину часто используются как основа при разработке различных эффективных алгоритмов работы с графами. Например, оба этих метода можно применить для нахождения связных компонент графа, поскольку связные компоненты представимы отдельными деревьями в остовном лесу графа.

6.4. Точки сочленения и двусвязные компоненты

Точкой сочленения графа называется такая вершина v, когда при удалении этой вершины и всех ребер, инцидентных вершине v, связная компонента графа разбивается на две или несколько частей. Например, точками сочленения для графа из рис. 6.8, *а* являются вершины a и c. Если удалить вершину a, то граф, который является одной связной компонентой, разбивается на два «треугольника» $\{b, d, e\}$ и $\{c, f, g\}$. Если удалить вершину c, то граф расчленяется на подграфы $\{a, b, d, e\}$ и $\{f, g\}$. Но если удалить какую-либо другую вершину, то в этом случае не удастся разбить связную компоненту на несколько частей. Связный граф, не имеющий точек сочленения, называется двусвязным. Для нахождения двусвязных компонент графа часто используется метод поиска в глубину.

Метод нахождения точек сочленения часто применяется для решения важной проблемы, касающейся k-связности графов. Граф называется k-связным, если удаление любых k — 1 вершин не приведет к расчленению

графа.[1] В частности, граф имеет связность 2 или выше тогда и только тогда, когда он не имеет точек сочленения, т. е. только тогда, когда он является двусвязным. Чем выше связность графа, тем больше можно удалить вершин из этого графа, не нарушая его целостности, т. е. не разбивая его на отдельные компоненты.

Опишем простой алгоритм нахождения всех точек сочленения связного графа, основанный на методе поиска в глубину. При отсутствии этих точек граф, естественно, будет двусвязным, т. е. этот алгоритм можно рассматривать так же, как тест на двусвязность неориентированного графа.

1. Выполняется обход графа методом поиска в глубину, при этом для всех вершин v вычисляются числа $dfnumber[v]$, введенные в разделе 5.5. В сущности, эти числа фиксируют последовательность обхода вершин в прямом порядке вершин глубинного остовного дерева.

2. Для каждой вершины v вычисляется число $low[v]$ равное минимуму чисел *dfnumber* потомков вершины v, включая и саму вершину v, и предков w вершины v, для которых существует обратное ребро (x, w), где x – потомок вершины v. Числа $low[v]$ для всех вершин v вычисляются при обходе остовного дерева в обратном порядке, поэтому при вычислении $low[v]$ для вершины v уже подсчитаны числа $low[x]$ для всех потомков x вершины v. Следовательно, $low[v]$ вычисляется как минимум следующих чисел:

а) $dfnumber[v]$;

б) $dfnumber[z]$ всех вершин z, для которых существует обратное ребро (v, z);

в) $low[x]$ всех потомков x вершины v.

3. Теперь точки сочленения определяются следующим образом:

а) корень остовного дерева будет точкой сочленения тогда и только тогда, когда он имеет двух или более сыновей. Так как в остовном дереве, которое получено методом поиска вглубь, нет поперечных ребер, то удаление такого корня расчленит остовное дерево на отдельные поддеревья с корнями, являющимися сыновьями корня построенного остовного дерева;

б) вершина v, отличная от корня, будет точкой сочленения тогда и только тогда, когда имеет такого сына w, что $low[w] \geq dfnumber[v]$. В этом случае удаление вершины v (и, конечно, всех инцидентных ей ребер) отделит вершину w и всех ее потомков от остальной части графа. Если же $low[w] < dfnumber[v]$, то существует путь по ребрам дерева к потомкам вершины w и обратное ребро от какого-нибудь из этих потомков к истинному предку вершины v (именно значению *dfnumber* для этого предка равно $low[w]$). Поэтому в данном случае удаление вершины v не отделит от графа поддерево с корнем w.

[1] Существует другое, более конструктивное определение k-связности. Граф называется k-связным, если между любой парой вершин v и w существует не менее k разных путей, таких, что, за исключением вершин v и w, ни одна из вершин, входящих в один путь, не входит ни в какой другой из этих путей.

Числа *dfnumber* и *low*, подсчитанные для вершин графа из рис. 6.8, *a*, показаны на рис. 6.11. В этом примере вычисление чисел *low* начато в обратном порядке с вершины *e*. Из этой вершины выходят обратные ребра (*e*, *a*) и (*e*, *b*), поэтому *low*[*e*] = min(*dfnumber*[*e*], *dfnumber*[*a*], *dfnumber*[*b*]) = 1. Затем рассматривается вершина *d*, для нее *low*[*d*] находится как минимум чисел *dfnumber*[*d*], *low*[*e*] и *dfnumber*[*a*]. Здесь число *low*[*e*] участвует потому, что вершина *e* является сыном вершины *d*, а число *dfnumber*[*a*] – из-за того, что существует обратное ребро (*d*, *a*).

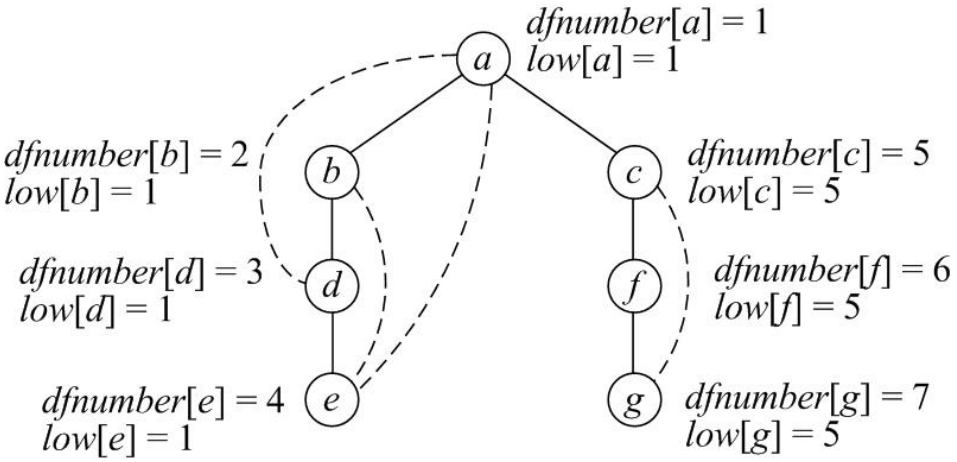


Рис. 6.11. Числа *dfnumber* и *low*, подсчитанные для графа из рис. 6.8, *a*

Для определения точек сочленения после вычисления всех чисел *low* просматривается каждая вершина остовного дерева. Корень *a* является точкой сочленения, так как имеет двух сыновей. Вершина *c* также является точкой сочленения – для ее сына, вершины *f*, выполняется неравенство *low*[*f*] ≥ *dfnumber*[*c*]. Другие вершины не являются точками сочленения.

Время выполнения описанного алгоритма на графе с *n* вершинами и *e* ребрами имеет порядок *O*(*e*). Можно легко проверить, что время, затрачиваемое на выполнение каждого этапа алгоритма, зависит или от количества посещаемых вершин, или от количества ребер, инцидентных этим вершинам, т. е. в любом случае время выполнения, с точностью до константы пропорциональности, является функцией как количества вершин, так и количества ребер. Поэтому общее время выполнения алгоритма имеет порядок *O*(*n* + *e*) или *O*(*e*), что то же самое при *n* ≤ *e*.

6.5. Паросочетания графов

В этом параграфе будет описан алгоритм решения «задачи о паросочетании» графов. Простым примером, приводящим к такой задаче, может слу-

жить ситуация распределения преподавателей по множеству учебных курсов. Надо назначить на чтение каждого курса преподавателя определенной квалификации так, чтобы ни на какой курс не было назначено более одного преподавателя. С другой стороны, желательно использовать максимально возможное количество преподавателей из общего состава.

Описанную ситуацию можно представить в виде графа, показанного на рис. 6.12, где все вершины разбиты на два множества V_1 и V_2 так, что вершины из множества V_1 соответствуют преподавателям, а вершины из множества V_2 – учебным курсам. Тот факт, что преподаватель v может вести курс w, отражается посредством ребра (v, w). Граф, у которого множество вершин распадается на два непересекающихся подмножества V_1 и V_2 таких, что каждое ребро графа имеет один конец из V_1, а другой – из V_2, называется двудольным. Таким образом, задача распределения преподавателей по учебным курсам сводится к задаче выбора определенных ребер в двудольном графе, имеющем множество вершин-преподавателей и множество вершин-учебных курсов.

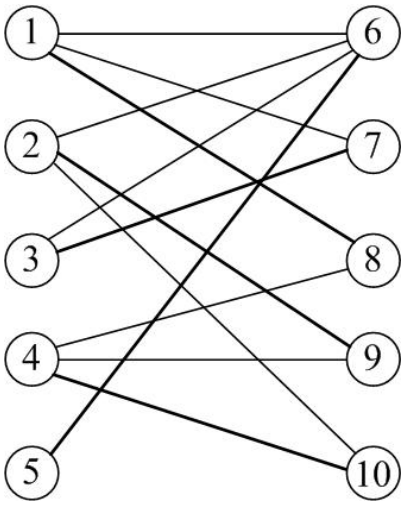


Рис. 6.12. Двудольный граф

Задачу паросочетания можно сформулировать следующим образом. Есть граф $G = (V, E)$. Подмножество его ребер такое, что никакие два ребра из этого подмножества не инциденты какой-либо одной вершине из V, называется паросочетанием. Задача определения максимального подмножества таких ребер называется задачей нахождения максимального паросочетания. Ребра, выделенные толстыми линиями на рис. 6.12, составляют одно возможное максимальное паросочетание для этого графа. Полным паросочетанием называется паросочетание, в котором участвуют (в качестве концов ребер) все вершины графа. Очевидно, что любое полное паросочетание является также максимальным паросочетанием.

Существуют прямые способы нахождения максимальных паросочетаний. Например, можно последовательно генерировать все возможные паросочетания, а затем выбрать то, которое содержит максимальное количество ребер.

Но такой подход имеет существенный недостаток – время его выполнения является экспоненциальной функцией от числа вершин.

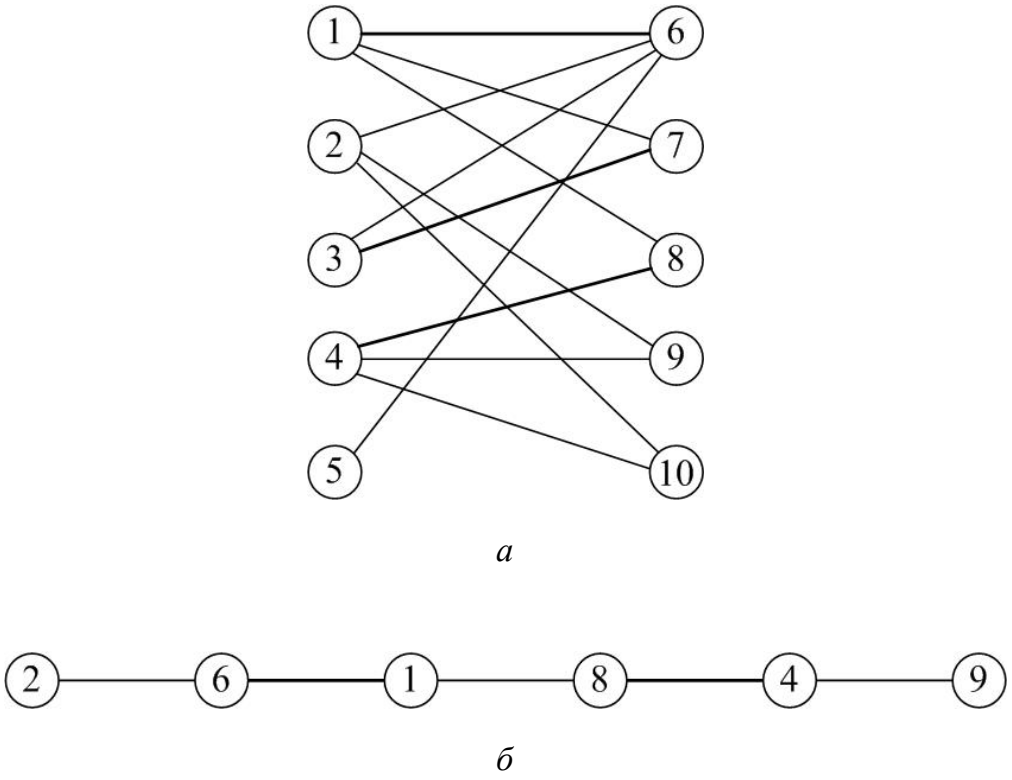
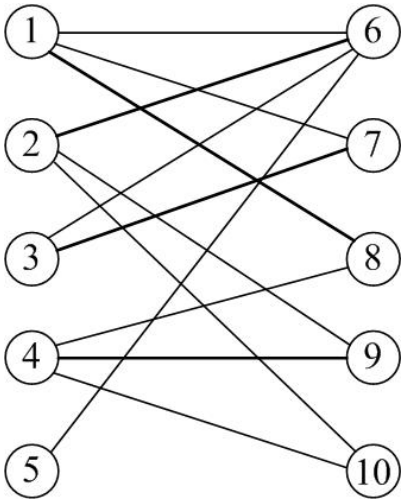


Рис. 6.13. Паросочетание (*а*) и чередующаяся цепь (*б*)

В настоящее время разработаны более эффективные алгоритмы нахождения максимальных паросочетаний. Эти алгоритмы используют в основном метод, известный как «чередующиеся цепи». Пусть M – паросочетание в графе G. Вершину v будем называть парной, если она является концом одного из ребер паросочетания M. Путь, соединяющий две непарные вершины, в котором чередуются ребра, входящие и не входящие в множество M, называется чередующейся (аугментальной) цепью относительно M. Очевидно, что чередующаяся цепь должна быть нечетной длины, начинаться и заканчиваться ребрами, не входящими в множество M. Также ясно, что, имея чередующуюся цепь P, можно увеличить паросочетание M, удалив из него те ребра, которые входят в цепь P, и вместо удаленных ребер добавить ребра из цепи P, которые первоначально не входили в паросочетание M. Это новое паросочетание можно определить как $M \Delta P$, где Δ обозначает симметрическую разность множеств M и P, т. е. в новое множество паросочетаний войдут те ребра, которые входят или в множество M, или в множество P, но не в оба сразу.

На рис. 6.13, *а* показаны граф и паросочетание M, состоящее из ребер (на рисунке они выделены толстыми линиями) (1, 6), (3, 7) и (4, 8). На рис. 6.13, *б* представлена чередующаяся цепь относительно M, состоящая из вер-

шин 2, 6, 1, 8, 4, 9. На рис. 6.14 изображено паросочетание (1, 8), (2, 6), (3, 7), (4, 9), которое получено удалением из паросочетания M ребер, входящих в эту чередующуюся цепь, и добавлением новых ребер, также входящих в эту цепь.



6.14. Увеличенное паросочетание

Паросочетание M будет максимальным тогда и только тогда, когда не будет существовать чередующейся цепи относительно M. Это ключевое свойство максимальных паросочетаний будет основой следующего алгоритма нахождения максимального паросочетания.

Пусть M и N – паросочетания в графе G, причем $|M| < |N|$ (здесь $|M|$ обозначает количество элементов множества M). Для того чтобы показать, что $M \Delta N$ содержит чередующуюся цепь относительно M, рассмотрим граф $G' = (V', M \Delta N)$, где V' – множество концевых вершин ребер, входящих в $M \Delta N$. Нетрудно заметить, что каждая связная компонента графа G' формирует простой путь (возможно, цикл), в котором чередуются ребра из M и N. Каждый цикл имеет равное число ребер, принадлежащих M и N, а каждый путь, не являющийся циклом, представляет собой чередующуюся цепь относительно или M, или N, в зависимости от того, ребер какого паросочетания больше в этой цепи. Поскольку $|M| < |N|$, то множество $M \Delta N$ содержит больше ребер из паросочетания N, чем M, и, следовательно, существует хотя бы одна чередующаяся цепь относительно M.

Теперь можно описать алгоритм нахождения максимального паросочетания M для графа $G = (V, E)$.

1. Сначала положим $M = \varnothing$.

2. Далее ищем чередующуюся цепь P относительно M, и множество M заменяется на множество $M \Delta P$.

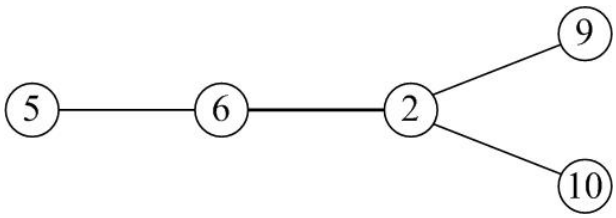
3. Шаг 2 повторяется до тех пор, пока существуют чередующиеся цепи относительно M. Если таких цепей больше нет, то M – максимальное паросочетание.

Теперь осталось показать способ построения чередующихся цепей относительно паросочетания M. Рассмотрим более простой случай, когда граф G является двудольным графом с множеством вершин, разбитым на два под-

множества V_1 и V_2. Будем строить граф чередующейся цепи, по уровням i = 0, 1, 2, …, используя процесс, подобный поиску в ширину. Граф чередующейся цепи уровня 0 содержит все непарные вершины из множества V_1. На уровне с нечетным номером i добавляются новые вершины, смежные с вершинами уровня i – 1 и соединенные ребром, не входящим в паросочетание (это ребро тоже добавляется в строящийся граф). На уровне с четным номером i также добавляются новые вершины, смежные с вершинами уровня i — 1, но которые соединены ребром, входящим в паросочетание (это ребро тоже добавляется в граф чередующейся цепи).

Процесс построения продолжается до тех пор, пока к графу чередующейся цепи можно присоединять новые вершины. Отметим, что непарные вершины присоединяются к этому графу только на нечетном уровне. В построенном графе путь от любой вершины нечетного уровня до любой вершины уровня 0 является чередующейся цепь относительно M.

На рис. 6.15 изображен граф чередующейся цепи, построенный для графа из рис. 6.13, а относительно паросочетания, показанного на рис. 6.14. На уровне 0 имеем одну непарную вершину 5. На уровне 1 добавлено ребро (5, 6), не входящее в паросочетание. На уровне 2 добавлено ребро (6, 2), входящее в паросочетание. На уровне 3 можно добавить или ребро (2, 9), или ребро (2, 10), не входящие в паросочетание. Поскольку вершины 9 и 10 пока в этом графе непарные, можно остановить процесс построения графа чередующейся цепи, добавив в него одну или другую вершину. Оба пути 9, 2, 6, 5 и 10, 2, 6, 5 являются чередующимися цепями относительно паросочетания из рис. 6.14.



6.15. Увеличенное паросочетание

Пусть граф G имеет n вершин и e ребер. Если используются списки смежности для представления ребер, то на построение графа чередующейся цепи потребуется времени порядка $O(e)$. Для нахождения максимального паросочетания надо построить не более n / 2 чередующихся цепей, поскольку каждая такая цепь увеличивает текущее паросочетание не менее чем на одно ребро. Поэтому максимальное паросочетание для двудольного графа можно найти за время порядка $O(n\ e)$.

8. СОВРЕМЕННЫЕ АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ

Настоящий раздел содержит краткое изложение некоторых идей, на основе которых разрабатываются современные алгоритмы. Подчеркнем тот факт, что сегодня наиболее интересные результаты в разработке эффективных алгоритмов связаны с привлечением аппарата других научных дисциплин, в частности – биологии и теоретических результатов специальных разделов современной математики [6].

8.1. Алгоритмы и простые числа

В не очень далеком прошлом теория простых чисел считалась разделом чисто теоретической математики, однако современные алгоритмические идеи решения разнообразных задач опираются на результаты этой теории. Значительное повышение интереса к простым числам в области информатики связано с известной криптосистемой *RSA*. При построении этой криптосистемы используется ряд алгоритмов, в том числе и алгоритм Евклида (III в. до н. э.), который до сих пор остается «современным» и востребованным в алгоритмической практике.

В целях дальнейшего изложения приведем некоторые сведения и обозначения из теории простых чисел.

Сравнения

Говорят, что два целых числа a и b сравнимы по модулю c, если они дают при делении на c равные остатки. Операция получения остатка от деления a на c записывается в виде: $a \bmod c = d$, что эквивалентно представлению: $a = k \times c + d$, $d \geq 0$, $k \geq 0$ – целые числа. Сравнимость двух чисел по модулю c означает, что $a \bmod c = b \bmod c$ и записывается в виде $a \equiv b \pmod c$. Если $a \equiv 0 \pmod c$, то число a делится на число c без остатка.

Простые числа

Число p называется простым, если оно не имеет других делителей, кроме единицы и самого себя. Очевидно, что при проверке в качестве возможных делителей есть смысл проверять только простые числа, меньшие или равные квадратному корню из проверяемого числа. Множество простых чисел счетно, доказательство принадлежит Евклиду. Пусть $p_1, \ldots, p_k$, – простые числа, и p_k – последнее из них, но тогда число $a = (p_1 \times p_2 \times \ldots \times p_{k+1})$ в силу основной теоремы арифметики должно разлагаться, и притом, единственно в

произведение простых. Но число a не делится нацело ни на одно из p_i. Приходим к противоречию.

Функция $\pi(n)$

Функция $\pi(n)$ в теории простых чисел обозначает количество простых чисел, не превосходящих n, например, $\pi(12) = 5$, так как существует 5 простых чисел не превосходящих 12. Асимптотическое поведение функции $\pi(n)$ было исследовано в конце XIX века. В 1896 году Адамар и Валле-Пуссен доказали, что

$$\pi(n) \approx \frac{n}{\ln n}, \qquad \lim_{n\to\infty} \pi(n) \cdot \frac{\ln n}{n} = 1.$$

Полученный результат означает, что простые числа не так уж «редки», получаем оценку 1 / (ln n) для вероятности того, что случайно взятое число, не превосходящее n, является простым.

Теорема Ферма

Введем операцию умножения чисел по модулю n, a x b *mod* n и определим степени числа: $a^2 = (a \text{ x } a)$ *mod* n, $a^{k+1} = (a^k \text{ x } a)$ *mod* n. Для любого простого числа p и числа a, $0 < a < p$, справедлива малая теорема Ферма (теорема Ферма-Эйлера): $a^{p-1} \equiv 1$ (*mod* p).

Алгоритм Евклида

Алгоритм Евклида асимптотически оптимально решает задачу нахождения наибольшего общего делителя двух чисел НОД(a, b). Заметим, что задача нахождения НОД является сегодня составной частью многих эффективных алгоритмов. Обоснование правильности этого алгоритма основано на свойствах НОД. Если d = НОД (a, b), то d есть наименьший положительный элемент множества целочисленных линейных комбинаций чисел a и b – (a x x + b x y), где x, y – целые числа; можно показать, что a *mod* b является целочисленной линейной комбинацией a и b, поэтому НОД(a, b) = НОД(b, a *mod* b).

В первой главе уже рассматривлась одна из реализаций алгоритма Евклида. Теперь приведем его рекурсивную реализацию:

```
Ec(a, b);
If b = 0
      then Ec = a;
      else Ec = Ec(b, a mod b);
end.
```

Трудоемкость алгоритма Евклида определяется значениями чисел. Лучшим случаем при фиксированной битовой длине чисел, очевидно, является ситуация, когда $b = 0$. Заметим, что в ситуации, когда $a < b$, алгоритм первой рекурсией переворачивает пару чисел. Анализ в худшем случае не так

очевиден – если $a > b \geq 0$, и $b < F_{k+1}$, где F_{k+1} – $(k + 1)$-ое число Фибоначчи, то процедура $Ec(a, b)$ выполняет менее k рекурсивных вызовов (теорема Ламе, начало XIX века). Из этой теоремы можно получить сложность алгоритма Евклида – поскольку $F_k \approx \varphi^k$, где

$$\varphi = (1+\sqrt{5})/2 \approx 1{,}618$$

(φ – величина, обратная к «золотому сечению»), то количество рекурсивных вызовов не превышает $1 + \log_\varphi b$. Если рассматривать числа a и b как двоичные β-битовые числа, то можно показать, что процедура $Ec(a, b)$ имеет сложность $O(\beta^2)$ битовых операций.

Вероятностный тест Миллера – Рабина

На практике, особенно в криптографических алгоритмах, возникает необходимость нахождения больших простых чисел. Один из эффективных алгоритмов решения этой задачи – вероятностный тест Миллера – Рабина (1980 г.).

Идея теста Миллера – Рабина состоит в последовательной генерации случайных чисел и проверке их на простоту с использованием теоремы Ферма – Эйлера. Основная задача теста связана с уменьшением вероятности ошибочного признания некоторого составного числа в качестве простого. Проблема состоит в том, что существуют составные числа n (числа Кармайкла), для которых сравнение $a^{n-1} \equiv 1 \ (mod\ n)$ справедливо для всех целых чисел a, удовлетворяющих неравенству $0 < a < n$. Решением проблемы является проверка на нетривиальный корень из 1 в момент вычисления a^{n-1} методом последовательного возведения в квадрат, т. е. проверка того, что $(x \times x)\ mod\ n = 1$, при $x \neq 1$, $x \neq n - 1$. Отсутствие нетривиальных корней свидетельствует в пользу того, что число n является простым.

Тест Миллера – Рабина

1. Цикл пока не будет найдено простое число
    2. Генерация случайного числа n (двоичное β-битовое число)
    3. Повторять s раз
        4. Случайный выбор числа $a \in \{2, \ldots, n-2\}$
            если $(a^{n-1})\ mod\ n \neq 1$, то, очевидно, что число n составное;
            если $(a^{n-1})\ mod\ n = 1$, то, необходимо проверить другое a;
            при вычислении $(a^{n-1})\ mod\ n$ проверить нетривиальный корень из 1
            если $(x \times x)\ mod\ n = 1$ при $x \neq 1$, $x \neq n - 1$, то число n – составное.
    5. Конец цикла по s
    6. Выход, если все выбранные числа a показали простоту числа n.
7. Конец теста.

Вероятность ошибки теста экспоненциально падает с ростом успешных проверок с различными значениями a, а именно, если выполнено s успешных проверок, то она составляет 2^{-s}, что приводит реально к выбору s в пределах

нескольких десятков. Сложность теста оценивается как $O(s \times \beta^3)$ битовых операций.

Алгоритм Рабина – Карпа

Одна из нетривиальных идей использования простых чисел для разработки и доказательства эффективности алгоритмов – идея предложенного в 1981 и опубликованного в 1987 году Рабином и Карпом алгоритма поиска подстроки в строке. Центральная идея алгоритма связана с переходом к сравнению чисел вместо посимвольного сравнения образца и части строки поиска.

Пусть дана символьная строка T длиной m (m-битовая строка) и образец P длиной n. Нам необходимо найти все вхождения образца в строку. Обозначим через T_r подстроку длиной n, начиная с бита r. Определим

$$H(P) = \sum_{i=1}^{n} 2^{n-i} P(i), \qquad H(T_r) = \sum_{i=1}^{n} 2^{n-i} T(r+i-1).$$

Мы рассматриваем тем самым подстроки как двоичные числа, очевидно, что образец P входит в строку T, начиная с позиции r, если $H(P) = H(T_r)$. Таким образом, мы сравниваем числа, а не символы, и если вычисление $H(T_r)$ при сдвиге может быть выполнено за $O(1)$, то сложность алгоритма составит $O(n + m)$, т. е. теоретическую границу сложности задачи поиска. Принципиальная проблема этой идеи – экспоненциальный рост чисел $H(P)$ и $H(T_r)$, с ростом длины образца. При обычном байтовом кодировании символов образец всего из 12 букв занимает 96 бит, и, следовательно, $H(P) \le 2^{96} - 1$. Необходимо было сохранить внешнюю привлекательность идеи, но уложиться в рамки реального диапазона представления целых чисел (машинного слова) в компьютере для образцов любой длины.

Решение проблемы, предложенное Рабином и Карпом, состоит в использовании остатков от деления $H(P)$ и $H(T_r)$ на некоторое простое число p. Эти остатки называются дактилограммами. Но действительная мощность предложенного метода состоит в доказательстве того, что вероятность ошибки (ложного срабатывания) может быть сделана малой, если простое число выбирается случайно из некоторого множества. Обозначим через

$$H_p(P) = H(P) \bmod p, \qquad H_p(T_r) = H(T_r) \bmod p.$$

Однако если вычислять вначале $H(P)$, а затем брать остаток, то мы снова окажемся за пределами машинного слова. Для эффективного вычисления $H_p(P)$ и $H_p(T_r)$ может быть использована схема Горнера, в результате чего во время вычислений ни один из промежуточных результатов не будет больше $2p$. Схема вычислений имеет вид

$$H_p(P) = ((((P(1) \times 2 \bmod p + P(2)) \times 2 \bmod p + P(3)) \times 2 \bmod p) + \ldots + P(n)) \bmod p.$$

Следующая задача на пути создания эффективного алгоритма – быстрое вычисление сдвига значения $H(T_r)$. Заметим, что

$$H(T_r) = 2 \text{ x } H(T_{r-1}) - 2n \text{ x } T(r-1) + T(r+n+1),$$

и, выполняя вычисления по *mod p*, можно вычислить $H_p(T_r)$ по $H_p(P)$ и $H_p(T_{r-1})$ с трудоемкостью $O(1)$. Очевидно, что $H_p(P)$ и $H_p(T_1)$ вычисляются однократно со сложностью $O(n)$. При каждом сдвиге по строке мы вычисляем $H_p(T_r)$ и сравниваем числа, что дает по порядку $O(m-n)$, и мы получаем общую линейную сложность. Обоснование того, что вероятность ложного совпадения (возможно, что числа $H(P)$ и $H(T_r)$ различны, в то время как $H_p(P)$ и $H_p(T_r)$ совпадают) мала, происходит на основе теорем и лемм теории простых чисел. Центральная теорема подхода Рабина – Карпа формулируется следующим образом:

Пусть P и T некоторые строки, причем $n \text{ x } m \geq 29$, где $n = |P|$, $m = |T|$. Пусть I – некоторое положительное число. Если p – случайно выбранное простое число, не превосходящее I, то вероятность ложного совпадения P и T не превосходит $\pi(n \text{ x } m) / \pi(I)$. Если выбрать $I = n \text{ x } m^2$, то вероятность ложного совпадения не превосходит $2{,}53 / m$.

Алгоритм Рабина – Карпа

1. Выбрать положительное целое I (например, $I = n \text{ x } m^2$) .
2. Случайно выбрать простое число p, не превосходящее I (например, используя тест Миллера – Рабина) .
3. По схеме Горнера вычислить $H_p(P)$ и $H_p(T_1)$.
4. Для каждой позиции r в T $(1 < r < m - n + 1)$ вычислить $H_p(T_r)$ и сравнить с $H_p(P)$. Если они равны, то либо объявить о вероятном совпадении, либо выполнить явную проверку, начиная с позиции r.

Существует несколько модификаций метода Рабина – Карпа, связанных с обнаружением ошибок. Один из них основан на методе Бойера – Мура и со сложностью $O(m)$ либо подтверждает отсутствие ложных совпадений, либо декларирует, что, по меньшей мере, одно из совпадений ложное. В случае ложных совпадений нужно выполнять алгоритм заново с новым значением p, вплоть до отсутствия ложных совпадений. Тем самым алгоритм Рабина – Карпа преобразуется из метода с малой вероятностью ошибки и со сложностью $O(m+n)$ в худшем случае в метод, который не делает ошибок, но обладает ожидаемой сложностью $O(m+n)$ – это преобразование алгоритма Монте-Карло в алгоритм Лас-Вегаса.

Отметим, что метод Рабина – Карпа успешно применяется (наряду с целым рядом других алгоритмов поиска подстрок) в таком разделе современной науки как вычислительная молекулярная биология, для поиска совпадающих цепочек ДНК и расшифровки генов.

8.2. Генетические алгоритмы

Практическая необходимость решения ряда *NP*-полных задач в оптимизационной постановке при проектировании и исследовании сложных систем привела разработчиков алгоритмического обеспечения к использованию биологических механизмов поиска наилучших решений. В настоящее время эффективные алгоритмы разрабатываются в рамках научного направления, которое можно назвать «природные вычисления», объединяющего такие разделы, как генетические алгоритмы, эволюционное программирование, нейросетевые вычисления, клеточные автоматы и ДНК-вычисления, муравьиные алгоритмы. Исследователи обращаются к природным механизмам, которые миллионы лет обеспечивают адаптацию биоценозов к окружающей среде. Одним из таких механизмов, имеющих фундаментальный характер, является механизм наследственности. Его использование для решения задач оптимизации привело к появлению генетических алгоритмов.

В живой природе особи в биоценозе конкурируют друг с другом за различные ресурсы, такие, как пища или вода. Кроме того, особи одного вида в популяции конкурируют между собой, например, за привлечение брачного партнера. Те особи, которые более приспособлены к окружающим условиям, будут иметь больше шансов на создание потомства. Слабо приспособленные либо не произведут потомства, либо их потомство будет очень немногочисленным. Это означает, что гены от высоко приспособленных особей будут распространяться в последующих поколениях. Комбинация хороших характеристик от различных родителей иногда может приводить к появлению потомка, приспособленность которого даже больше, чем приспособленность его родителей. Таким образом, вид в целом развивается, лучше и лучше приспосабливаясь к среде обитания.

Введение в генетические алгоритмы

Алгоритм решения задач оптимизации, основанный на идеях наследственности в биологических популяциях, был впервые предложен Джоном Холландом (1975 г.). Он получил название репродуктивного плана Холланда, и широко использовался как базовый алгоритм в эволюционных вычислениях. Дальнейшее развитие эти идеи, как собственно и свое название – генетические алгоритмы, получили в работах Гольдберга и Де Йонга.

Цель генетического алгоритма при решении задачи оптимизации состоит в том, чтобы найти лучшее возможное, но не гарантированно оптимальное решение. Для реализации генетического алгоритма необходимо выбрать подходящую структуру данных для представления решений. В постановке задачи поиска оптимума, экземпляр этой структуры должен содержать информацию о некоторой точке в пространстве решений.

Структура данных генетического алгоритма состоит из набора хромосом. Хромосома, как правило, представляет собой битовую строку, так что термин

строка часто заменяет понятие «хромосома». Вообще говоря, хромосомы генетических алгоритмов не ограничены только бинарным представлением. Известны другие реализации, построенные на векторах вещественных чисел. Несмотря на то, что для многих реальных задач, видимо, больше подходят строки переменной длины, в настоящее время структуры фиксированной длины наиболее распространены и изучены.

Для иллюстрации идеи мы ограничимся только структурами, которые являются битовыми строками. Каждая хромосома (строка) представляет собой последовательное объединение ряда подкомпонентов, которые называются генами. Гены расположены в различных позициях или локусах хромосомы, и принимают значения, называемые аллелями – это биологическая терминология. В представлении хромосомы бинарной строкой, ген является битом этой строки, локус – есть позиция бита в строке, а аллель – это значение гена, 0 или 1. Биологический термин «генотип» относится к полной генетической модели особи и соответствует структуре в генетическом алгоритме. Термин «фенотип» относится к внешним наблюдаемым признакам и соответствует вектору в пространстве параметров задачи. В генетике под мутацией понимается преобразование хромосомы, случайно изменяющее один или несколько генов. Наиболее распространенный вид мутаций – случайное изменение только одного из генов хромосомы. Термин кроссинговер обозначает порождение из двух хромосом двух новых путем обмена генами. В литературе по генетическим алгоритмам также употребляется термин кроссовер, скрещивание или рекомбинация. В простейшем случае кроссинговер в генетическом алгоритме реализуется так же, как и в биологии. При скрещивании хромосомы разрезаются в случайной точке и обмениваются частями между собой. Например, если хромосомы (11, 12, 13, 14) и (0, 0, 0, 0) разрезать между вторым и третьим генами и обменять их части, то получатся следующие потомки (11, 12, 0, 0) и (0, 0, 13, 14).

Основные структуры и фазы генетического алгоритма

Приведем простой иллюстративный пример – задачу максимизации некоторой функции двух переменных $f(x_1, x_2)$, при ограничениях: $0 < x_1 < 1$ и $0 < x_2 < 1$. Обычно методика кодирования реальных переменных x_1 и x_2 состоит в преобразовании их в двоичные целочисленные строки определенной длины, достаточной для того, чтобы обеспечить желаемую точность. Предположим, что 10-ти разрядное кодирование достаточно для x_1 и x_2. Установить соответствие между генотипом и фенотипом можно, разделив соответствующее двоичное целое число на $2^{10} - 1$. Например, 0000000000 соответствует 0 / 1023 или 0, тогда как 1111111111 соответствует 1023 / 1023 или 1.

Оптимизируемая структура данных есть 20-битовая строка, представляющая собой конкатенацию (объединение) кодировок x_1 и x_2. Пусть переменная x_1 размещается в крайних левых 10 битах строки, тогда как x_2 размещается в правой части генотипа особи. Таким образом, генотип пред-

ставляет собой точку в 20-мерном целочисленном пространстве (вершину единичного гиперкуба), которое исследуется генетическим алгоритмом. Для этой задачи фенотип будет представлять собой точку в двумерном пространстве параметров (x_1, x_2).

Чтобы решить задачу оптимизации нужно задать некоторую меру качества для каждой структуры в пространстве поиска. Для этой цели используется функция приспособленности. При максимизации целевая функция часто сама выступает в качестве функции приспособленности, для задач минимизации целевая функция инвертируется и смещается в область положительных значений.

Рассмотрим фазы работы простого генетического алгоритма. Вначале случайным образом генерируется начальная популяция (набор хромосом). Работа алгоритма представляет собой итерационный процесс, который продолжается до тех пор, пока не будет смоделировано заданное число поколений или выполнен некоторый критерий останова. В каждом поколении реализуется пропорциональный отбор приспособленности, одноточечная рекомбинация и вероятностная мутация. Пропорциональный отбор реализуется путем назначения каждой особи (хромосоме) i вероятности $P(i)$, равной отношению ее приспособленности к суммарной приспособленности популяции (по целевой функции):

$$P(i) = \frac{f(i)}{\sum_{i=1}^{n} f(i)}.$$

Затем происходит отбор (с замещением) всех n особей для дальнейшей генетической обработки, согласно убыванию величины $P(i)$. Простейший пропорциональный отбор реализуется с помощью рулетки (Гольдберг, 1989 г.). «Колесо» рулетки содержит по одному сектору для каждого члена популяции, а размер i-го сектора пропорционален соответствующей величине $P(i)$. При таком отборе члены популяции, обладающие более высокой приспособленностью, будут выбираться чаще по вероятности.

После отбора выбранные n особей подвергаются рекомбинации с заданной вероятностью P_c, при этом n хромосом случайным образом разбиваются на n / 2 пар. Для каждой пары с вероятность P_c может быть выполнена рекомбинация. Если рекомбинация происходит, то полученные потомки заменяют собой родителей. Одноточечная рекомбинация работает следующим образом. Случайным образом выбирается одна из точек разрыва, т. е. участок между соседними битами в строке. Обе родительские структуры разрываются на два сегмента по этому участку. Затем соответствующие сегменты различных родителей склеиваются и получаются два генотипа потомков.

После стадии рекомбинации выполняется фаза мутации. В каждой строке, которая подвергается мутации, каждый бит инвертируется с вероятностью P_m. Популяция, полученная после мутации, записывается поверх старой, и на этом завершается цикл одного поколения в генетическом алгоритме.

Полученное новое поколение обладает (по вероятности) более высокой приспособленностью, наследованной от «хороших» представителей предыдущего поколения. Таким образом, из поколения в поколение, хорошие характеристики распространяются по всей популяции. Скрещивание наиболее приспособленных особей приводит к тому, что исследуются наиболее перспективные участки пространства поиска. В результате популяция будет сходиться к локально оптимальному решению задачи, а иногда, может быть, благодаря мутации, и к глобальному оптимуму.

Теперь можно сформулировать основные шаги генетического алгоритма.

Генетический алгоритм

1. Создать начальную популяцию
2. Цикл по поколениям пока не выполнено условие останова
   // цикл жизни одного поколения
   3. Оценить приспособленность каждой особи
   4. Выполнить отбор по приспособленности
   5. Случайным образом разбить популяцию на две группы пар
   6. Выполнить фазу вероятностной рекомбинации для пар популяции и заменить родителей
   7. Выполнить фазу вероятностной мутации
   8. Оценить приспособленность новой популяции и вычислить условие останова
   9. Объявить потомков новым поколением
10. Конец цикла по поколениям

Модификации генетического алгоритма

Очевидно, что тонкая настройка базового генетического алгоритма может быть выполнена путем изменения значений вероятностей рекомбинации и мутации, существует много исследований и предложений в данной области. В настоящее время предлагаются разнообразные модификации генетических алгоритмов в части методов отбора по приспособленности, рекомбинации и мутации. Приведем несколько примеров.

Метод турнирного отбора (Бриндел, 1981 г.; Гольдберг и Деб, 1991 г.) реализуется в виде n турниров для выборки n особей. Каждый турнир состоит в выборе k элементов из популяции и отбора лучшей особи среди них. Элитные методы отбора (Де Ионг, 1975 г.) гарантируют, что при отборе обязательно будут выживать лучший или лучшие члены популяции. Наиболее распространена процедура обязательного сохранения только одной лучшей особи, если она не прошла через процесс отбора, рекомбинации и мутации. Этот метод может быть внедрен практически в любой стандартный метод отбора.

Двухточечная рекомбинация (Гольдберг 1989 г.) и равномерная рекомбинация (Сисверда, 1989 г.) являются вполне достойными альтернативами одноточечному оператору. При двухточечной рекомбинации выбираются две

точки разрыва, и родительские хромосомы обмениваются сегментом, который находится между двумя этими точками. Равномерная рекомбинация предполагает, что каждый бит первого родителя наследуется первым потомком с заданной вероятностью; в противном случае этот бит передается второму потомку.

Механизмы мутаций могут быть так же заимствованы из молекулярной биологии, например, обмен концевых участков хромосомы (механизм транслокации), обмен смежных сегментов (транспозиция). По мнению М.В. Ульянова, интерес представляет механизм инверсии, т. е. перестановки генов в хромосоме, управляющим параметром при этом может выступать инверсионное расстояние – минимальное количество единичных инверсий генов, преобразующих исходную хромосому в мутированную.

Применение генетических алгоритмов

Основная проблема, связанная с применением генетических алгоритмов – это их эвристический характер. Говоря более строго, какова вероятность достижения популяцией глобального оптимума в заданной области при данных настройках алгоритма? В настоящее время не существует строгого ответа и теоретически обоснованных оценок. Имеются предположения, что генетический алгоритм может стать эффективной процедурой поиска оптимального решения, если:

- пространство поиска достаточно велико, и предполагается, что целевая функция не является гладкой и унимодальной в области поиска, т. е. не содержит один гладкий экстремум;
- задача не требует нахождения глобального оптимума, необходимо достаточно быстро найти приемлемое «хорошее» решение, что довольно часто встречается в реальных задачах.

Если целевая функция обладает свойствами гладкости и унимодальности, то любой градиентный метод, такой как метод наискорейшего спуска, будет более эффективен. Генетический алгоритм является в определенном смысле универсальным методом, т. е. он явно не учитывает специфику задачи или должен быть на нее каким-то образом специально настроен. Поэтому если имеется некоторую дополнительную информацию о целевой функции и пространстве поиска (как, например, для хорошо известной задачи коммивояжера), то методы поиска, использующие эвристики, определяемые задачей, часто превосходят любой универсальный метод.

С другой стороны, при достаточно сложном рельефе функции приспособленности градиентные методы с единственным решением могут останавливаться в локальном решении. Наличие у генетических алгоритмов целой «популяции» решений, совместно с вероятностным механизмом мутации, позволяют предполагать меньшую вероятность нахождения локального оптимума и большую эффективность работы на многоэкстремальном ландшафте.

Сегодня генетические алгоритмы успешно применяются для решения классических *NP*-полных задач, задач оптимизации в пространствах с боль-

шим количеством измерений, ряда экономических задач оптимального характера, например, задач распределения инвестиций.

8.3. Муравьиные алгоритмы

Муравьиные алгоритмы представляют собой новый перспективный метод решения задач оптимизации, в основе которого лежит моделирование поведения колонии муравьев. Колония представляет собой систему с очень простыми правилами автономного поведения особей. Однако, несмотря на примитивность поведения каждого отдельного муравья, поведение всей колонии оказывается достаточно разумным. Эти принципы проверены временем – удачная адаптация к окружающему миру на протяжении миллионов лет означает, что природа выработала очень удачный механизм поведения. Исследования в этой области начались в середине 90-х годов XX века, автором идеи является Марко Дориго из Университета Брюсселя, Бельгия.

8.3.1. Биологические принципы поведения муравьиной колонии

Муравьи относятся к социальным насекомым, образующим коллективы. В биологии коллектив муравьев называется колонией. Число муравьев в колонии может достигать нескольких миллионов, на сегодня известны суперколонии муравьев (*Formica lugubrus*), протянувшиеся на сотни километров. Одним из подтверждений оптимальности поведения колоний является тот факт, что сеть гнезд суперколоний близка к минимальному остовному дереву графа их муравейников.

Основу поведения муравьиной колонии составляет самоорганизация, обеспечивающая достижения общих целей колонии на основе низкоуровневого взаимодействия. Колония не имеет централизованного управления, и ее особенностями является обмен локальной информацией только между отдельными особями (прямой обмен – пища, визуальные и химические контакты) и наличие непрямого обмена, который и используется в муравьиных алгоритмах.

Непрямой обмен – стигмержи (*stigmergy*), представляет собой разнесенное во времени взаимодействие, при котором одна особь изменяет некоторую область окружающей среды, а другие используют эту информацию позже, в момент, когда они в нее попадают. Биологи установили, что такое отложенное взаимодействие происходит через специальное химическое вещество – феромон (*pheromone*), секрет специальных желез, откладываемый при перемещении муравья. Концентрация феромона на тропе определяет предпочтительность движения по ней. Адаптивность поведения реализуется испарением феромона, который в природе воспринимается муравьями в течение нескольких суток. Можно провести некоторую аналогию между рас-

пределением феромона в окружающем колонию пространстве, и «глобальной» памятью муравейника, носящей динамический характер.

8.3.2. Идея муравьиного алгоритма

Идея муравьиного алгоритма – моделирование поведения муравьев, связанное с их способностью быстро находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь. При своем движении муравей метит свой путь феромоном, и эта информация используется другими муравьями для выбора пути. Это элементарное правило поведения и определяет способность муравьев находить новый путь, если старый оказывается недоступным. Дойдя до преграды, муравьи с равной вероятностью будут обходить ее справа и слева. То же самое будет происходить и на обратной стороне преграды. Однако, те муравьи, которые случайно выберут кратчайший путь, будут быстрее его проходить, и за несколько передвижений он будет более обогащен феромоном. Поскольку движение муравьев определяется концентрацией феромона, то следующие будут предпочитать именно этот путь, продолжая обогащать его феромоном, до тех пор, пока этот путь по какой-либо причине не станет доступен. Очевидная положительная обратная связь быстро приведет к тому, что кратчайший путь станет единственным маршрутом движения большинства муравьев. Моделирование испарения феромона – отрицательной обратной связи, гарантирует нам, что найденное локально оптимальное решение не будет единственным – муравьи будут искать и другие пути. Если мы моделируем процесс такого поведения на некотором графе, ребра которого представляют собой возможные пути перемещения муравьев, в течение определенного времени, то наиболее обогащенный феромоном путь по ребрам этого графа и будет являться решением задачи, полученным с помощью муравьиного алгоритма. Рассмотрим конкретный пример.

8.3.3. Формализация задачи коммивояжера в терминах муравьиного подхода

Задача формулируется как задача поиска минимального по стоимости замкнутого маршрута по всем вершинам без повторений на полном взвешенном графе с n вершинами. Содержательно вершины графа являются городами, которые должен посетить коммивояжер, а веса ребер отражают расстояния (длины) или стоимости проезда. Эта задача является *NP*-трудной, и точный переборный алгоритм ее решения имеет факториальную сложность. Приводимое здесь описание муравьиного алгоритма для задачи коммивояжера является кратким изложением статьи С.Д. Штовбы.

Моделирование поведения муравьев связано с распределением феромона на тропе – ребре графа в задаче коммивояжера. При этом вероятность включения ребра в маршрут отдельного муравья пропорциональна количеству феромона на этом ребре, а количество откладываемого феромона пропор-

ционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на его ребрах, следовательно, большее количество муравьев будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости – большинство муравьев двигается по локально оптимальному маршруту. Избежать этого можно, моделируя отрицательную обратную связь в виде испарения феромона. При этом если феромон испаряется быстро, то это приводит к потере памяти колонии и забыванию хороших решений, с дугой стороны, большое время испарения может привести к получению устойчивого локально оптимального решения.

Теперь, с учетом особенностей задачи коммивояжера, можно описать локальные правила поведения муравьев при выборе пути.

1. Муравьи имеют собственную «память». Поскольку каждый город может быть посещен только один раз, у каждого муравья есть список уже посещенных городов – список запретов. Обозначим через $J_{i,\,k}$ список городов, которые необходимо посетить муравью k, находящемуся в городе i.

2. Муравьи обладают «зрением» – видимость есть эвристическое желание посетить город j, если муравей находится в городе i. Будем считать, что видимость обратно пропорциональна расстоянию между городами

$$\eta_{i,\,j} = 1 / D_{ij}.$$

3. Муравьи обладают «обонянием» – они могут улавливать след феромона, подтверждающий желание посетить город j из города i, на основании опыта других муравьев. Количество феромона на ребре (i, j) в момент времени t обозначим через $\tau_{ij}(t)$.

4. На этом основании можно сформулировать вероятностно-пропорциональное правило, определяющее вероятность перехода k-го муравья из города i в город j:

$$\begin{cases} P_{ij,k}(t) = \dfrac{\left[\tau_{ij}(t)\right]^{\alpha} \cdot \left[\eta_{ij}\right]^{\beta}}{\sum\limits_{l \in J_{i,k}} \left[\tau_{il}(t)\right]^{\alpha} \cdot \left[\eta_{il}\right]^{\beta}}, & j \in J_{i,\,k} \\ P_{ij,k}(t) = 0, & j \notin J_{i,\,k} \end{cases} \qquad (8.1)$$

где α, β – параметры, задающие веса следа феромона, при $\alpha = 0$ алгоритм вырождается до жадного алгоритма (будет выбран ближайший город). Заметим, что выбор города является вероятностным, правило (8.1) лишь определяет ширину зоны города j; в общую зону всех городов $J_{i,\,k}$ бросается случайное число, которое и определяет выбор муравья. Правило (8.1) не изменяется в ходе алгоритма, но у двух разных муравьев значение вероятности перехода будут отличаться, так как они имеют разный список разрешенных городов.

5. Пройдя ребро (i, j) муравей откладывает на нем некоторое количество феромона, которое должно быть связано с оптимальностью сделанного

выбора. Пусть $T_k(t)$ есть маршрут, пройденный муравьем k к моменту времени t, $L_k(t)$ – длина этого маршрута, а Q – параметр, имеющий значение порядка длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде

$$\Delta\tau_{ij,\,k}(t) = \begin{cases} \dfrac{Q}{L_k(t)}, & (i, j) \in T_k(t); \\ 0, & (i, j) \notin T_k(t). \end{cases}$$

Правила внешней среды определяют, в первую очередь, испарение феромона. Пусть $p \in [0, 1]$ есть коэффициент испарения, тогда правило испарения имеет вид

$$\tau_{ij}(t + 1) = (1 - p)\,\tau_{ij}(t) + \Delta\,\tau_{ij}(t); \qquad \Delta\,\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij,k}(t), \tag{8.2}$$

где m – количество муравьев в колонии.

В начале алгоритма количество феромона на ребрах принимается равным небольшому положительному числу. Общее количество муравьев остается постоянным и равным количеству городов, каждый муравей начинает маршрут из своего города.

Дополнительная модификация алгоритма может состоять во введении так называемых «элитных» муравьев, которые усиливают ребра наилучшего маршрута, найденного с начала работы алгоритма. Обозначим через T^* наилучший текущий маршрут, через L^* – его длину. Тогда если в колонии есть e элитных муравьев, ребра маршрута получат дополнительное количество феромона

$$\Delta\,\tau_e = e\,Q\,/\,L^*. \tag{8.3}$$

Муравьиный алгоритм для задачи коммивояжера

1. Ввод матрицы расстояний D.
2. Инициализация параметров алгоритма – α, β, e, Q.
3. Инициализация ребер – присвоение видимости η_{ij} и начальной концентрации феромона.
4. Размещение муравьев в случайно выбранные города без совпадений.
5. Выбор начального кратчайшего маршрута и определение L^*
   // основной цикл.
6. Цикл по времени жизни колонии $t = 1, tmax$.
   7. Цикл по всем муравьям $k = 1, m$
      8. Построить маршрут $T_k(t)$ по правилу (8.1) и рассчитать длину $L_k(t)$.
   9. конец цикла по муравьям.
   10. Проверка всех $L_k(t)$ на лучшее решение по сравнению с L^*.
      11. Если да, то обновить L^* и T^*.
   12. Цикл по всем ребрам графа.
      13. Обновить следы феромона на ребре по правилам (8.2) и (8.3).

14. конец цикла по ребрам.
15. конец цикла по времени.
16. Вывести кратчайший маршрут T^* и его длину L^*.

Сложность данного алгоритма определяется непосредственно из приведенного выше текста – $O(tmax, n^2, m)$, таким образом, сложность алгоритма зависит от времени жизни колонии ($tmax$), количества городов (n) и количества муравьев в колонии (m).

8.3.4. Области применения и возможные модификации

Поскольку в основе муравьиного алгоритма лежит моделирование передвижения муравьев по некоторым путям, то такой подход может стать эффективным способом поиска рациональных решений для задач оптимизации, допускающих графовую интерпретацию. Ряд экспериментов показывает, что эффективность муравьиных алгоритмов растет с ростом размерности решаемых задач оптимизации. Хорошие результаты получаются для нестационарных систем с изменяемыми во времени параметрами, например, для расчетов телекоммуникационных и компьютерных сетей. Муравьиный алгоритм применялся для разработки оптимальной структуры съемочных сетей *GPS*, в рамках создания высокоточных геодезических и съемочных технологий. В настоящее время на основе применения муравьиных алгоритмов получены хорошие результаты для таких сложных оптимизационных задач, как задача коммивояжера, транспортная задача, задача календарного планирования, задача раскраски графа, квадратичной задачи о назначениях, задачи оптимизации сетевых графиков и ряда других.

Качество получаемых решений во многом зависит от настроечных параметров в вероятностно-пропорциональном правиле выбора пути на основе текущего количества феромона и параметров правил откладывания и испарения феромона. Возможно, что динамическая адаптационная настройка этих параметров может способствовать получению лучших решений. Немаловажную роль играет и начальное распределение феромона, а также выбор условно оптимального решения на шаге инициализации.

Отмечается, что перспективными путями улучшения муравьиных алгоритмов является адаптация параметров с использованием базы нечетких правил и их гибридизация, например, с генетическими алгоритмами. Как вариант, такая гибридизация может состоять в обмене, через определенные промежутки времени, текущими наилучшими решениями.

Много полезной информации по муравьиным алгоритмам можно найти на специальных англоязычных сайтах по этому направлению.

ЗАКЛЮЧЕНИЕ

Создание компьютерной программы для решения какой-либо задачи состоит из нескольких этапов: формализация и создание технического задания на исходную задачу, разработка алгоритма решения задачи, написание кода программы, тестирование и отладка программы. Структуры данных и алгоритмы обработки данных можно рассматривать как строительные блоки создаваемых компьютерных программ. Таким образом, алгоритмы и структуры данных, приведенные в данном учебном пособии, являются фундаментом современного программирования на вычислительных машинах.

Очевидно, что эффективность компьютерной программы зависит от алгоритма, который в ней реализован. Использованный в пособии подход к оценке и анализу времени выполнения алгоритмов позволяет выбрать оптимальный вариант для нужд программиста. Как было показано, не всегда самый быстрый алгоритм является лучшим вариантом: в том случае, если объем обрабатываемых данных невелик, лучше воспользоваться более простым алгоритмом. Во-первых, это сэкономит время на его разработку, тестирование и отладку. Во-вторых, чем проще алгоритм для понимания самого программиста, тем меньше вероятность допустить логическую ошибку при его реализации.

В учебном пособии представлены алгоритмы, направленные на решение таких классических задач, как поиск необходимой информации в некотором информационном пространстве, сортировка данных, были рассмотрены задачи обработки графов.

Существенным достоинством данного пособия являются представленные алгоритмы параллельной обработки данных. Время, необходимое для решения указанных выше задач, может быть значительно сокращено при использовании распределенной обработки информации. Такой подход требует дополнительных алгоритмических решений. Однако теоретические исследования данной проблемы представляют большой интерес, а использование параллелизма при обработке данных имеет большую практическую значимость.

Другим достоинством учебного пособия являются такие современные алгоритмы: генетические и муравьиные алгоритмы. Отметим, что сегодня наиболее интересные результаты в разработке эффективных алгоритмов связаны с привлечением аппарата других научных дисциплин, таких, например, как биология.

Данное учебное пособие можно рассматривать как обзорное, поскольку оно охватывает довольно широкий спектр задач и методов их решения. Приведенные в пособии алгоритмы обработки данных позволяют создавать программы, направленные на решение вычислительных задач различного характера и масштаба. Практическая значимость задачи обработки данных по-

зволяет рассматривать представленные в данном учебном пособии алгоритмы как отправную точку для дальнейших исследований.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Ахо, А. Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. М.: Изд. дом «Вильямс», 2001. 384с.
2. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. СПб.: Невский диалект, 2001. 352с.
3. Вирт, Н. Алгоритмы + структуры данных = программы / Н. Вирт. М.: Мир, 1985. 360с.
4. Кнут, Д. Искусство программирования для ЭВМ. Поиск и сортировка / Д. Кнут. М.: Изд. дом «Вильямс», 2001. Т. 3. 844с.
5. Ковалев, И. В. Моделирование и оптимизация параллельных процессов в информационно-управляющих системах / И. В. Ковалев, Р. Ю. Царев. Красноярск: ИПЦ КГТУ, 2003. 111с.
6. Макконел, Дж. Основы современных алгоритмов / Дж. Макконел. М.: Техносфера, 2004. 368с.
7. Новиков, Ф. А. Дискретная математика для программистов / Ф. А. Новиков. СПб.: Питер, 2000. 304с.
8. Полякова, О. А. Методические указания для выполнения лабораторных работ по информатике для студентов специальности АСУ / О. А. Полякова. Пермь: РИО ПГТУ, 2001. 210с.
9. Царев, Р. Ю. Структуры и алгоритмы обработки данных. Поиск и сортировка данных / Р.Ю. Царев. Красноярск: ИПЦ КГТУ, 2005. 60с.
10. Материалы Интернет-сайта http://algolist.manual.ru.