
ГЛАВА 16

Жадные алгоритмы

Алгоритмы, предназначенные для решения задач оптимизации, обычно представляют собой последовательность шагов, на каждом из которых предоставляется некоторое множество выборов. Определение наилучшего выбора, руководствуясь принципами динамического программирования, во многих задачах оптимизации напоминает стрельбу из пушки по воробьям; другими словами, для этих задач лучше подходят более простые и эффективные алгоритмы. В *жадном алгоритме* (greedy algorithm) всегда делается выбор, который кажется самым лучшим в данный момент — т.е. производится локально оптимальный выбор в надежде, что он приведет к оптимальному решению глобальной задачи. В этой главе рассматриваются задачи оптимизации, пригодные для решения с помощью жадных алгоритмов. Перед освоением главы следует ознакомиться с динамическим программированием, изложенным в главе 15, в частности, с разделом 15.3.

Жадные алгоритмы не всегда приводят к оптимальному решению, но во многих задачах они дают нужный результат. В разделе 16.1 рассматривается простая, но нетривиальная задача о выборе процессов, эффективное решение которой можно найти с помощью жадного алгоритма. Чтобы прийти к жадному алгоритму, сначала будет рассмотрено решение, основанное на парадигме динамического программирования, после чего будет показано, что оптимальное решение можно получить, исходя из принципов жадных алгоритмов. В разделе 16.2 представлен обзор основных элементов подхода, в соответствии с которым разрабатываются жадные алгоритмы. Это позволит упростить обоснование корректности жадных алгоритмов, не используя при этом сравнение с динамическим программированием, как это делается в разделе 16.1. В разделе 16.3 приводится важное приложение методов жадного программирования: разработка кодов Хаффмана (Huffman) для

сжатия данных. В разделе 16.4 исследуются теоретические положения, на которых основаны комбинаторные структуры, известные под названием “матроиды”, для которых жадный алгоритм всегда дает оптимальное решение. Наконец, в разделе 16.5 матроиды применяются для решения задачи о составлении расписания заданий равной длительности с предельным сроком и штрафами.

Жадный метод обладает достаточной мощностью и хорошо подходит для довольно широкого класса задач. В последующих главах представлены многие алгоритмы, которые можно рассматривать как применение жадного метода, включая алгоритмы поиска минимальных остовных деревьев (minimum-spanning-tree) (глава 23), алгоритм Дейкстры (Dijkstra) для определения кратчайших путей из одного источника (глава 24) и эвристический жадный подход Чватала (Chvatal) к задаче о покрытии множества (set-covering) (глава 35). Алгоритмы поиска минимальных остовных деревьев являются классическим примером применения жадного метода. Несмотря на то, что эту главу и главу 23 можно читать независимо друг от друга, может оказаться полезно читать их одна за другой.

16.1 Задача о выборе процессов

В качестве первого примера рассмотрим задачу о составлении расписания для нескольких конкурирующих процессов, каждый из которых безраздельно использует общий ресурс. Цель этой задачи — выбор набора взаимно совместимых процессов, образующих множество максимального размера. Предположим, имеется множество $S = \{a_1, a_2, \dots, a_n\}$, состоящее из n *процессов* (activities). Процессам требуется некоторый ресурс, который одновременно может использоваться лишь одним процессом. Каждый процесс a_i характеризуется *начальным моментом* s_i и *конечным моментом* f_i , где $0 \leq s_i < f_i < \infty$. Будучи выбран, процесс a_i длится в течение полуоткрытого интервала времени $[s_i, f_i)$. Процессы a_i и a_j *совместимы* (compatible), если интервалы $[s_i, f_i)$ и $[s_j, f_j)$ не перекрываются (т.е. если $s_i \geq f_j$ или $s_j \geq f_i$). **Задача о выборе процессов** (activity-selection problem) заключается в том, чтобы выбрать подмножество взаимно совместимых процессов, образующих множество максимального размера. Например, рассмотрим описанное ниже множество S процессов, отсортированных в порядке возрастания моментов окончания:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

(Вскоре станет понятно, почему удобнее рассматривать процессы, расположенные именно в таком порядке.) В данном примере из взаимно совместимых процессов можно составить подмножество $\{a_3, a_9, a_{11}\}$. Однако оно не является максималь-

ным, поскольку существует подмножество $\{a_1, a_4, a_8, a_{11}\}$, состоящее из большего количества элементов. Еще одно такое подмножество — $\{a_2, a_4, a_9, a_{11}\}$.

Разобьем решение этой задачи на несколько этапов. Начнем с того, что сформулируем решение рассматриваемой задачи, основанное на принципах динамического программирования. Это оптимальное решение исходной задачи получается путем комбинирования оптимальных решений подзадач. Рассмотрим несколько вариантов выбора, который делается в процессе определения подзадачи, использующейся в оптимальном решении. Впоследствии станет понятно, что заслуживает внимания лишь один выбор — жадный — и что когда делается этот выбор, одна из подзадач гарантированно получается пустой. Остается лишь одна непустая подзадача. Исходя из этих наблюдений, мы разработаем рекурсивный жадный алгоритм, предназначенный для решения задачи о составлении расписания процессов. Процесс разработки жадного алгоритма будет завершен его преобразованием из рекурсивного в итерационный. Описанные в этом разделе этапы сложнее, чем те, что обычно имеют место при разработке жадных алгоритмов, однако они иллюстрируют взаимоотношение между жадными алгоритмами и динамическим программированием.

Оптимальная подструктура задачи о выборе процессов

Как уже упоминалось, мы начнем с того, что разработаем решение для задачи о выборе процессов по методу динамического программирования. Как и в главе 15, первый шаг будет состоять в том, чтобы найти оптимальную подструктуру и с ее помощью построить оптимальное решение задачи, пользуясь оптимальными решениями подзадач.

В главе 15 мы убедились, что для этого нужно определить подходящее пространство подзадач. Начнем с определения множеств

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}.$$

S_{ij} — подмножество процессов из множества S , которые можно успеть выполнить в промежутке времени между завершением процесса a_i и началом процесса a_j . Фактически множество S_{ij} состоит из всех процессов, совместимых с процессами a_i и a_j , а также теми, которые оканчиваются не позже процесса a_i и теми, которые начинаются не ранее процесса a_j . Для представления всей задачи добавим фиктивные процессы a_0 и a_{n+1} и примем соглашение, что $f_0 = 0$ и $s_{n+1} = \infty$. Тогда $S = S_{0,n+1}$, а индексы i и j находятся в диапазоне $0 \leq i, j \leq n+1$.

Дополнительные ограничения диапазонов индексов i и j можно получить с помощью приведенных ниже рассуждений. Предположим, что процессы отсортированы в порядке возрастания соответствующих им конечных моментов времени:

$$f_0 \leq f_1 \leq \dots \leq f_n \leq f_{n+1}. \quad (16.1)$$

Утверждается, что в этом случае $S_{ij} = \emptyset$, если $i \geq j$. Почему? Предположим, что существует процесс $a_k \in S_{ij}$ для некоторого $i \geq j$, так что процесс a_i следует после процесса a_j , если расположить их в порядке сортировки. Однако из определения S_{ij} следует соотношение $f_i \leq s_k < f_k \leq s_j < f_j$, откуда $f_i < f_j$, что противоречит предположению о том, что процесс a_i следует после процесса a_j , если расположить их в порядке сортировки. Можно прийти к выводу, что если процессы отсортированы в порядке монотонного возрастания времени их окончания, то пространство подзадач образуется путем выбора максимального подмножества взаимно совместимых процессов из множества S_{ij} для $0 \leq i < j \leq n + 1$ (мы знаем, что все прочие S_{ij} — пустые).

Чтобы понять подструктуру задачи о выборе процессов, рассмотрим некоторую непустую подзадачу S_{ij} ¹ и предположим, что ее решение включает некоторый процесс a_k , так что $f_i \leq s_k < f_k \leq s_j$. С помощью процесса a_k генерируются две подзадачи, S_{ik} (процессы, которые начинаются после окончания процесса a_i и заканчиваются перед началом процесса a_k) и S_{kj} (процессы, которые начинаются после окончания процесса a_k и заканчиваются перед началом процесса a_j), каждая из которых состоит из подмножества процессов, входящих в S_{ij} . Решение задачи S_{ij} представляет собой объединение решений задач S_{ik} , S_{kj} и процесса a_k . Таким образом, количество процессов, входящее в состав решения задачи S_{ij} , — это сумма количества процессов в решении задачи S_{ik} , количества процессов в решении задачи S_{kj} и еще одного процесса (a_k).

Опишем оптимальную подструктуру этой задачи. Предположим, что оптимальное решение A_{ij} задачи S_{ij} включает в себя процесс a_k . Тогда решения A_{ik} задачи S_{ik} и A_{kj} задачи S_{kj} в рамках оптимального решения S_{ij} тоже должны быть оптимальными. Как обычно, это доказывается с помощью рассуждений типа “вырезания и вставки”. Если бы существовало решение A'_{ik} задачи S_{ik} , включающее в себя большее количество процессов, чем A_{ik} , в решение A_{ij} можно было бы вместо A_{ik} подставить A'_{ik} , что привело бы к решению задачи S_{ij} , в котором содержится больше процессов, чем в A_{ij} . Поскольку предполагается, что A_{ij} — оптимальное решение, мы приходим к противоречию. Аналогично, если бы существовало решение A'_{kj} задачи S_{kj} , содержащее больше процессов, чем A_{kj} , то путем замены A_{kj} на A'_{kj} можно было бы получить решение задачи S_{ij} , в которое входит больше процессов, чем в A_{ij} .

Теперь с помощью сформулированной выше оптимальной подструктуры покажем, что оптимальное решение задачи можно составить из оптимальных решений подзадач. Мы уже знаем, что в любое решение непустой задачи S_{ij} входит некоторый процесс a_k и что любое оптимальное решение задачи содержит в себе

¹Иногда о множествах S_{ij} мы будем говорить не как о множествах процессов, а как о вспомогательных задачах. Из контекста всегда будет понятно, что именно обозначает S_{ij} — множество процессов или вспомогательную задачу, входными данными для которой является это множество.

оптимальные решения подзадач S_{ik} и S_{kj} . Таким образом, максимальное подмножество взаимно совместимых процессов множества S_{ij} можно составить путем разбиения задачи на две подзадачи (определение подмножеств максимального размера, состоящих из взаимно совместимых процессов в задачах S_{ik} и S_{kj}) — собственно нахождения максимальных подмножеств A_{ik} и A_{kj} взаимно совместимых процессов, являющихся решениями этих подзадач, и составления подмножества A_{ij} максимального размера, включающего в себя взаимно совместимые задачи:

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}. \quad (16.2)$$

Оптимальное решение всей задачи представляет собой решение задачи $S_{0,n+1}$.

Рекурсивное решение

Второй этап разработки решения по методу динамического программирования — определение значения, соответствующего оптимальному решению. Пусть в задаче о выборе процесса $c[i, j]$ — количество процессов в подмножестве максимального размера, состоящем из взаимно совместимых процессов в задаче S_{ij} . Эта величина равна нулю при $S_{ij} = \emptyset$; в частности, $c[i, j] = 0$ при $i \geq j$.

Теперь рассмотрим непустое подмножество S_{ij} . Мы уже знаем, что если процесс a_k используется в максимальном подмножестве, состоящем из взаимно совместимых процессов задачи S_{ij} , то для подзадач S_{ik} и S_{kj} также используются подмножества максимального размера. С помощью уравнения (16.2) получаем рекуррентное соотношение

$$c[i, j] = c[i, k] + c[k, j] + 1.$$

В приведенном выше рекурсивном уравнении предполагается, что значение k известно, но на самом деле это не так. Это значение можно выбрать из $j - i - 1$ возможных значений: $k = i + 1, \dots, j - 1$. Поскольку в подмножестве максимального размера для задачи S_{ij} должно использоваться одно из этих значений k , нужно проверить, какое из них подходит лучше других. Таким образом, полное рекурсивное определение величины $c[i, j]$ принимает вид:

$$c[i, j] = \begin{cases} 0 & \text{при } S_{ij} = \emptyset, \\ \max_{\substack{i < k < j \\ a_k \in S_{ij}}} \{c[i, k] + c[k, j] + 1\} & \text{при } S_{ij} \neq \emptyset. \end{cases} \quad (16.3)$$

Преобразование решения динамического программирования в жадное решение

На данном этапе не составляет труда написать восходящий алгоритм динамического программирования, основанный на рекуррентном уравнении (16.3). Это

предлагается сделать читателю в упражнении (16.1-1). Однако можно сделать два наблюдения, позволяющие упростить полученное решение.

Теорема 16.1. Рассмотрим произвольную непустую задачу S_{ij} и пусть a_m — процесс, который оканчивается раньше других:

$$f_m = \min \{f_k : a_k \in S_{ij}\}.$$

В этом случае справедливы такие утверждения.

1. Процесс a_m используется в некотором подмножестве максимального размера, состоящем из взаимно совместимых процессов задачи S_{ij} .
2. Подзадача S_{im} пустая, поэтому в результате выбора процесса a_m непустой остается только подзадача S_{mj} .

Доказательство. Сначала докажем вторую часть теоремы, так как она несколько проще. Предположим, что подзадача S_{im} непустая и существует некоторый процесс a_k , такой что $f_i \leq s_k < f_k \leq s_m < f_m$. Тогда процесс a_k также входит в решение подзадачи S_{im} , причем он оканчивается раньше, чем процесс a_m , что противоречит выбору процесса a_m . Таким образом, мы приходим к выводу, что решение подзадачи S_{im} — пустое множество.

Чтобы доказать первую часть, предположим, что A_{ij} — подмножество максимального размера взаимно совместимых процессов задачи S_{ij} , и упорядочим процессы этого подмножества в порядке возрастания времени их окончания. Пусть a_k — первый процесс множества A_{ij} . Если $a_k = a_m$, то доказательство завершено, поскольку мы показали, что процесс a_m используется в некотором подмножестве максимального размера, состоящем из взаимно совместимых процессов задачи S_{ij} . Если же $a_k \neq a_m$, то составим подмножество $A'_{ij} = A_{ij} - \{a_k\} \cup \{a_m\}$. Процессы в A'_{ij} не перекрываются, поскольку процессы во множестве A_{ij} не перекрываются, a_k — процесс из множества A_{ij} , который оканчивается раньше всех, и $f_m \leq f_k$. Заметим, что количество процессов во множестве A'_{ij} совпадает с количеством процессов во множестве A_{ij} , поэтому A'_{ij} — подмножество максимального размера, состоящее из взаимно совместимых процессов задачи S_{ij} и включающее в себя процесс a_m . \square

Почему теорема 16.1 имеет такое большое значение? Из раздела 15.3 мы узнали, что оптимальные подструктуры различаются количеством подзадач, использующихся в оптимальном решении исходной задачи, и количеством возможностей, которые следует рассмотреть при определении того, какие подзадачи нужно выбрать. В решении, основанном на принципах динамического программирования, в оптимальном решении используется две подзадачи, а также до $j - i - 1$ вариантов выбора для подзадачи S_{ij} . Благодаря теореме 16.1 обе эти величины значительно

уменьшаются: в оптимальном решении используется лишь одна подзадача (вторая подзадача гарантированно пустая), а в процессе решения подзадачи S_{ij} достаточно рассмотреть только один выбор — тот процесс, который оканчивается раньше других. К счастью, его легко определить.

Кроме уменьшения количества подзадач и количества выборов, теорема 16.1 дает еще одно преимущество: появляется возможность решать каждую подзадачу в нисходящем направлении, а не в восходящем, как это обычно происходит в динамическом программировании. Чтобы решить подзадачу S_{ij} , в ней выбирается процесс a_m , который оканчивается раньше других, после чего в это решение добавляется множество процессов, использующихся в оптимальном решении подзадачи S_{mj} . Поскольку известно, что при выбранном процессе a_m в оптимальном решении задачи S_{ij} безусловно используется решение задачи S_{mj} , нет необходимости решать задачу S_{mj} до задачи S_{ij} . Чтобы решить задачу S_{ij} , можно *сначала* выбрать a_m , который представляет собой процесса из S_{ij} , который заканчивается раньше других, а *потом* решать задачу S_{mj} .

Заметим также, что существует единый шаблон для всех подзадач, которые подлежат решению. Наша исходная задача — $S = S_{0,n+1}$. Предположим, что в качестве процесса задачи $S_{0,n+1}$, который оканчивается раньше других, выбран процесс a_{m_1} . (Поскольку процессы отсортированы в порядке монотонного возрастания времени их окончания, и $f_0 = 0$, должно выполняться равенство $m_1 = 1$.) Следующая подзадача — $S_{m_1,n+1}$. Теперь предположим, что в качестве процесса задачи $S_{m_1,n+1}$, который оканчивается раньше других, выбран процесс a_{m_2} (не обязательно, чтобы $m_2 = 2$). Следующая на очереди подзадача — $S_{m_2,n+1}$. Продолжая рассуждения, нетрудно убедиться в том, что каждая подзадача будет иметь вид $S_{m_i,n+1}$ и ей будет соответствовать некоторый номер процесса m_i . Другими словами, каждая подзадача состоит из некоторого количества процессов, завершающихся последними, и это количество меняется от одной подзадачи к другой.

Для процессов, которые первыми выбираются в каждой подзадаче, тоже существует свой шаблон. Поскольку в задаче $S_{m_i,n+1}$ всегда выбирается процесс, который оканчивается раньше других, моменты окончания процессов, которые последовательно выбираются во всех подзадачах, образуют монотонно возрастающую последовательность. Более того, каждый процесс в процессе решения задачи можно рассмотреть лишь один раз, воспользовавшись сортировкой в порядке возрастания времен окончания процессов.

В ходе решения подзадачи всегда выбирается процесс a_m , который оканчивается раньше других. Таким образом, выбор является жадным в том смысле, что интуитивно для остальных процессов он оставляет как можно больше возможностей войти в расписание. Таким образом, жадный выбор — это такой выбор, который максимизирует количество процессов, пока что не включенных в расписание.

Рекурсивный жадный алгоритм

После того как стал понятен путь упрощения решения, основанного на принципах динамического программирования, и преобразования его в нисходящий метод, можно перейти к рассмотрению алгоритма, работающего исключительно в жадной нисходящей манере. Здесь приводится простое рекурсивное решение, реализованное в виде процедуры `RECURSIVE_ACTIVITY_SELECTOR`. На ее вход подаются значения начальных и конечных моментов процессов, представленные в виде массивов s и f , а также индексы i и n , определяющие подзадачу $S_{i,n+1}$, которую требуется решить. (Параметр n идентифицирует последний реальный процесс a_n в подзадаче, а не фиктивный процесс a_{n+1} , который тоже входит в эту подзадачу.) Процедура возвращает множество максимального размера, состоящее из взаимно совместимых процессов задачи $S_{i,n+1}$. В соответствии с уравнением (16.1), предполагается, что все n входных процессов расположены в порядке монотонного возрастания времени их окончания. Если это не так, их можно отсортировать в указанном порядке за время $O(n \lg n)$. Начальный вызов этой процедуры имеет вид `RECURSIVE_ACTIVITY_SELECTOR(s, f, 0, n)`.

`RECURSIVE_ACTIVITY_SELECTOR(s, f, i, n)`

```

1   $m \leftarrow i + 1$ 
2  while  $m \leq n$  и  $s_m < f_i$             $\triangleright$  Поиск первого процесса в  $S_{i,n+1}$ .
3      do  $m \leftarrow m + 1$ 
4  if  $m \leq n$ 
5      then return  $\{a_m\} \cup \text{RECURSIVE\_ACTIVITY\_SELECTOR}(s, f, m, n)$ 
6      else return  $\emptyset$ 
```

Работа представленного алгоритма для рассматривавшихся в начале этого раздела 11 процессов проиллюстрирована на рис. 16.1. Процессы, которые рассматриваются в каждом рекурсивном вызове, разделены горизонтальными линиями. Фиктивный процесс a_0 оканчивается в нулевой момент времени, и в начальном вызове, который имеет вид `RECURSIVE_ACTIVITY_SELECTOR(s, f, 0, 11)`, выбирается процесс a_1 . В каждом рекурсивном вызове заштрихованы процессы, которые уже были выбраны, а белым цветом обозначен рассматриваемый процесс. Если начальный момент процесса наступает до конечного момента процесса, который был добавлен последним (т.е. стрелка, проведенная от первого процесса ко второму, направлена влево), такой процесс отвергается. В противном случае (стрелка направлена прямо вверх или вправо) процесс выбирается. В последнем рекурсивном вызове процедуры, имеющем вид `RECURSIVE_ACTIVITY_SELECTOR(s, f, 11, 11)`, возвращается \emptyset . Результирующее множество выбранных процессов — $\{a_1, a_4, a_8, a_{11}\}$.

В рекурсивном вызове процедуры `RECURSIVE_ACTIVITY_SELECTOR(s, f, i, n)` в цикле **while** в строках 2–3 осуществляется поиск первого процесса задачи $S_{i,n+1}$. В цикле проверяются процессы $a_{i+1}, a_{i+2}, \dots, a_n$ до тех пор, пока не будет найден

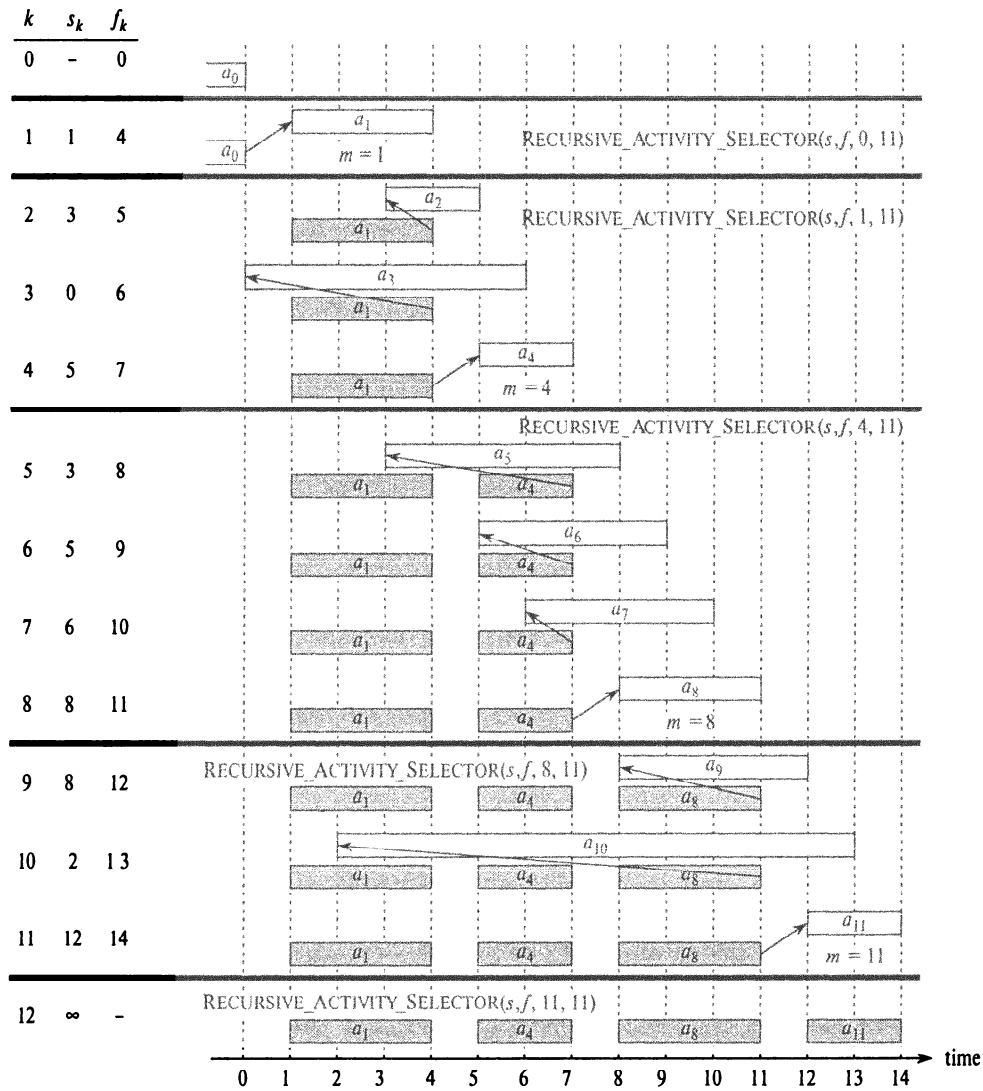


Рис. 16.1. Обработка алгоритмом `RECURSIVE_ACTIVITY_SELECTOR` множества из одиннадцати процессов

первый процесс a_m , совместимый с процессом a_i ; для такого процесса справедливо соотношение $s_m \geq f_i$. Если цикл завершается из-за того, что такой процесс найден, то происходит возврат из процедуры в строке 5, в которой процесс $\{a_m\}$ объединяется с подмножеством максимального размера для задачи $S_{m,n+1}$, которое возвращается рекурсивным вызовом `RECURSIVE_ACTIVITY_SELECTOR(s, f, m, n)`. Еще одна возможная причина завершения процесса — достижение условия $m > n$.

В этом случае проверены все процессы, но среди них не найден такой, который был бы совместим с процессом a_i . Таким образом, $S_{i,n+1} = \emptyset$ и процедура возвращает значение \emptyset в строке 6.

Если процессы отсортированы в порядке, заданном временем их окончания, время, которое затрачивается на вызов процедуры `RECURSIVE_ACTIVITY_SELECTOR($s, f, 0, n$)`, равно $\Theta(n)$. Это можно показать следующим образом. Сколько бы ни было рекурсивных вызовов, каждый процесс проверяется в цикле **while** в строке 2 ровно по одному разу. В частности, процесс a_k проверяется в последнем вызове, когда $i < k$.

Итерационный жадный алгоритм

Представленную ранее рекурсивную процедуру легко преобразовать в итеративную. Процедура `RECURSIVE_ACTIVITY_SELECTOR` почти подпадает под определение конечной рекурсии (см. задачу 7-4): она оканчивается рекурсивным вызовом самой себя, после чего выполняется операция объединения. Преобразование процедуры, построенной по принципу конечной рекурсии, в итерационную — обычно простая задача. Некоторые компиляторы разных языков программирования выполняют эту задачу автоматически. Как уже упоминалось, процедура `RECURSIVE_ACTIVITY_SELECTOR` выполняется для подзадач $S_{i,n+1}$, т.е. для подзадач, состоящих из процессов, которые оканчиваются позже других.

Процедура `GREEDY_ACTIVITY_SELECTOR` — итеративная версия процедуры `RECURSIVE_ACTIVITY_SELECTOR`. В ней также предполагается, что входные процессы расположены в порядке монотонного возрастания времен окончания. Выбранные процессы объединяются в этой процедуре в множество A , которое и возвращается процедурой после ее окончания.

`GREEDY_ACTIVITY_SELECTOR(s, f)`

```

1   $n \leftarrow \text{length}[s]$ 
2   $A \leftarrow \{a_1\}$ 
3   $i \leftarrow 1$ 
4  for  $m \leftarrow 2$  to  $n$ 
5      do if  $s_m \geq f_i$ 
6          then  $A \leftarrow A \cup \{a_m\}$ 
7               $i \leftarrow m$ 
8  return  $A$ 
```

Процедура работает следующим образом. Переменная i индексирует самое последнее добавление к множеству A , соответствующее процессу a_i в рекурсивной версии. Поскольку процессы рассматриваются в порядке монотонного возрастания моментов их окончания, f_i — всегда максимальное время окончания всех

процессов, принадлежащих множеству A :

$$f_i = \max \{f_k : a_k \in A\} \quad (16.4)$$

В строках 2–3 выбирается процесс a_1 , инициализируется множество A , содержащее только этот процесс, а переменной i присваивается индекс этого процесса. В цикле **for** в строках 4–7 происходит поиск процесса задачи $S_{i,n+1}$, оканчивающегося раньше других. В этом цикле по очереди рассматривается каждый процесс a_m , который добавляется в множество A , если он совместим со всеми ранее выбранными процессами; этот процесс оканчивается раньше других в задаче $S_{i,n+1}$. Чтобы узнать, совместим ли процесс a_m с процессами, которые уже содержатся во множестве A , в соответствии с уравнением (16.4) достаточно проверить (строка 5), что его начальный момент s_m наступает не раньше момента f_i окончания последнего из добавленных в множество A процессов. Если процесс a_m удовлетворяет сформулированным выше условиям, то в строках 6–7 он добавляется в множество A и переменной i присваивается значение m . Множество A , возвращаемое процедурой $\text{GREEDY_ACTIVITY_SELECTOR}(s, f)$, совпадает с тем, которое возвращается процедурой $\text{RECURSIVE_ACTIVITY_SELECTOR}(s, f, 0, n)$.

Процедура $\text{GREEDY_ACTIVITY_SELECTOR}$, как и ее рекурсивная версия, составляет расписание для n -элементного множества в течение времени $\Theta(n)$. Это утверждение справедливо в предположении, что процессы уже отсортированы в порядке возрастания времени их окончания.

Упражнения

- 16.1-1. Разработайте алгоритм динамического программирования для решения задачи о выборе процессов, основанный на рекуррентном соотношении (16.3). В этом алгоритме должны вычисляться определенные выше размеры $c[i, j]$, а также выводиться подмножество процессов A максимального размера. Предполагается, что процессы отсортированы в порядке, заданном уравнением (16.1). Сравните время работы найденного алгоритма со временем работы процедуры $\text{GREEDY_ACTIVITY_SELECTOR}$.
- 16.1-2. Предположим, что вместо того, чтобы все время выбирать процесс, который оканчивается раньше других, выбирается процесс, который начинается позже других и совместим со всеми ранее выбранными процессами. Опишите этот подход как жадный алгоритм и докажите, что он позволяет получить оптимальное решение.
- 16.1-3. Предположим, что имеется множество процессов, для которых нужно составить расписание при наличии большого количества ресурсов. Цель — включить в расписание все процессы, используя при этом как можно меньше ресурсов. Сформулируйте эффективный жадный алгоритм,

позволяющий определить, какой ресурс должен использоваться тем или иным процессом.

(Эта задача известна также как задача о *раскрашивании интервального графа* (interval-graph colouring problem). Можно создать интервальный граф, вершины которого сопоставляются заданным процессам, а ребра соединяют несовместимые процессы. Минимальное количество цветов, необходимых для раскрашивания всех вершин таким образом, чтобы никакие две соединенные вершины не имели один и тот же цвет, будет равно минимальному количеству ресурсов, необходимых для работы всех заданных процессов.)

- 16.1-4. Не все жадные подходы к задаче о выборе процессов позволяют получить множество, состоящее из максимального количества взаимно совместимых процессов. Приведите пример, иллюстрирующий, что выбор самых коротких процессов среди тех, что совместимы с ранее выбранными, не дает правильного результата. Выполните такое же задание для подходов, в одном из которых всегда выбирается процесс, совместимый с ранее выбранными и перекрывающийся с минимальным количеством оставшихся, а в другом — совместимый с ранее выбранными процесс, который начинается раньше других.

16.2 Элементы жадной стратегии

Жадный алгоритм позволяет получить оптимальное решение задачи путем осуществления ряда выборов. В каждой точке принятия решения в алгоритме делается выбор, который в данный момент выглядит самым лучшим. Эта эвристическая стратегия не всегда дает оптимальное решение, но все же решение может оказаться и оптимальным, в чем мы смогли убедиться на примере задачи о выборе процессов. В настоящем разделе обсуждаются некоторые общие свойства жадных методов.

Процесс разработки жадного алгоритма, рассмотренный в разделе 16.1, несколько сложнее, чем обычно. Были пройдены перечисленные ниже этапы.

1. Определена оптимальная подструктура задачи.
2. Разработано рекурсивное решение.
3. Доказано, что на любом этапе рекурсии один из оптимальных выборов является жадным. Из этого следует, что всегда можно делать жадный выбор.
4. Показано, что все возникающие в результате жадного выбора подзадачи, кроме одной, — пустые.
5. Разработан рекурсивный алгоритм, реализующий жадную стратегию.
6. Рекурсивный алгоритм преобразован в итеративный.

В ходе выполнения этих этапов мы во всех подробностях смогли рассмотреть, как динамическое программирование послужило основой для жадного алгоритма. Однако обычно на практике при разработке жадного алгоритма эти этапы упрощаются. Мы разработали подструктуру так, чтобы в результате жадного выбора оставалась только одна подзадача, подлежащая оптимальному решению. Например, в задаче о выборе процессов сначала определяются подзадачи S_{ij} , в которых изменяются оба индекса, — и i , и j . Потом мы выяснили, что если всегда делается жадный выбор, то подзадачи можно было бы ограничить видом $S_{i,n+1}$.

Можно предложить альтернативный подход, в котором оптимальная подструктура приспособлялась бы специально для жадного выбора — т.е. второй индекс можно было бы опустить и определить подзадачи в виде $S_i = \{a_k \in S : f_i \leq s_k\}$. Затем можно было бы доказать, что жадный выбор (процесс, который оканчивается первым в задаче S_i) в сочетании с оптимальным решением для множества S_m остальных совместимых между собой процессов приводит к оптимальному решению задачи S_i . Обобщая сказанное, опишем процесс разработки жадных алгоритмов в виде последовательности перечисленных ниже этапов.

1. Привести задачу оптимизации к виду, когда после сделанного выбора остается решить только одну подзадачу.
2. Доказать, что всегда существует такое оптимальное решение исходной задачи, которое можно получить путем жадного выбора, так что такой выбор всегда допустим.
3. Показать, что после жадного выбора остается подзадача, обладающая тем свойством, что объединение оптимального решения подзадачи со сделанным жадным выбором приводит к оптимальному решению исходной задачи.

Описанный выше упрощенный процесс будет использоваться в последующих разделах данной главы. Тем не менее, заметим, что в основе каждого жадного алгоритма почти всегда находится более сложное решение в стиле динамического программирования.

Как определить, способен ли жадный алгоритм решить стоящую перед нами задачу оптимизации? Общего пути здесь нет, однако можно выделить две основные составляющие — свойство жадного выбора и оптимальную подструктуру. Если удастся продемонстрировать, что задача обладает двумя этими свойствами, то с большой вероятностью для нее можно разработать жадный алгоритм.

Свойство жадного выбора

Первый из названных выше основных составляющих жадного алгоритма — *свойство жадного выбора*: глобальное оптимальное решение можно получить, делая локальный оптимальный (жадный) выбор. Другими словами, рассуждая по поводу того, какой выбор следует сделать, мы делаем выбор, который кажется

самым лучшим в текущей задаче; результаты возникающих подзадач при этом не рассматриваются.

Рассмотрим отличие жадных алгоритмов от динамического программирования. В динамическом программировании на каждом этапе делается выбор, однако обычно этот выбор зависит от решений подзадач. Следовательно, методом динамического программирования задачи обычно решаются в восходящем направлении, т.е. сначала обрабатываются более простые подзадачи, а затем — более сложные. В жадном алгоритме делается выбор, который выглядит в данный момент наилучшим, после чего решается подзадача, возникающая в результате этого выбора. Выбор, сделанный в жадном алгоритме, может зависеть от сделанных ранее выборов, но он не может зависеть от каких бы то ни было выборов или решений последующих подзадач. Таким образом, в отличие от динамического программирования, где подзадачи решаются в восходящем порядке, жадная стратегия обычно разворачивается в нисходящем порядке, когда жадный выбор делается один за другим, в результате чего каждый экземпляр текущей задачи сводится к более простому.

Конечно же, необходимо доказать, что жадный выбор на каждом этапе приводит к глобальному оптимальному решению, и здесь потребуются определенные интеллектуальные усилия. Обычно, как это было в теореме 16.1, в таком доказательстве исследуется глобальное оптимальное решение некоторой подзадачи. Затем демонстрируется, что решение можно преобразовать так, чтобы в нем использовался жадный выбор, в результате чего получится аналогичная, но более простая подзадача.

Свойство жадного выбора часто дает определенное преимущество, позволяющее повысить эффективность выбора в подзадаче. Например, если в задаче о выборе процессов предварительно отсортировать процессы в порядке монотонного возрастания моментов их окончания, то каждый из них достаточно рассмотреть всего один раз. Зачастую оказывается, что благодаря предварительной обработке входных данных или применению подходящей структуры данных (нередко это очередь с приоритетами) можно ускорить процесс жадного выбора, что приведет к повышению эффективности алгоритма.

Оптимальная подструктура

Оптимальная подструктура проявляется в задаче, если в ее оптимальном решении содержатся оптимальные решения подзадач. Это свойство является основным признаком применимости как динамического программирования, так и жадных алгоритмов. В качестве примера оптимальной подструктуры напомним результаты раздела 16.1. В нем было продемонстрировано, что если оптимальное решение подзадачи S_{ij} содержит процесс a_k , то оно также содержит оптимальные решения подзадач S_{ik} и S_{kj} . После выявления этой оптимальной подструктуры

было показано, что если известно, какой процесс используется в качестве процесса a_k , то оптимальное решение задачи S_{ij} можно построить путем выбора процесса a_k и объединения его со всеми процессами в оптимальных решениях подзадач S_{ik} и S_{kj} . На основе этого наблюдения удалось получить рекуррентное соотношение (16.3), описывающее оптимальное решение.

Обычно при работе с жадными алгоритмами применяется более простой подход. Как уже упоминалось, мы воспользовались предположением, что подзадача получилась в результате жадного выбора в исходной задаче. Все, что осталось сделать, — это обосновать, что оптимальное решение подзадачи в сочетании с ранее сделанным жадным выбором приводит к оптимальному решению исходной задачи. В этой схеме для доказательства того, что жадный выбор на каждом шаге приводит к оптимальному решению, неявно используется индукция по вспомогательным задачам.

Сравнение жадных алгоритмов и динамического программирования

Поскольку свойство оптимальной подструктуры применяется и в жадных алгоритмах, и в стратегии динамического программирования, может возникнуть соблазн разработать решение в стиле динамического программирования для задачи, в которой достаточно применить жадное решение. Не исключена также возможность ошибочного вывода о применимости жадного решения в той ситуации, когда необходимо решать задачу методом динамического программирования. Чтобы проиллюстрировать тонкие различия между этими двумя методами, рассмотрим две разновидности классической задачи оптимизации.

Дискретная задача о рюкзаке (0-1 knapsack problem) формулируется следующим образом. Вор во время ограбления магазина обнаружил n предметов. Предмет под номером i имеет стоимость v_i грн. и вес w_i кг, где v_i и w_i — целые числа. Нужно унести вещи, суммарная стоимость которых была бы как можно большей, однако грузоподъемность рюкзака ограничивается W килограммами, где W — целая величина. Какие предметы следует взять с собой?

Ситуация в **непрерывной задаче о рюкзаке** (fractional knapsack problem) та же, но теперь тот или иной товар вор может брать с собой частично, а не делать каждый раз бинарный выбор — брать или не брать (0–1). В дискретной задаче о рюкзаке в роли предметов могут выступать, например, слитки золота, а для непрерывной задачи больше подходят такие товары, как золотой песок.

В обеих разновидностях задачи о рюкзаке проявляется свойство оптимальной подструктуры. Рассмотрим в целочисленной задаче наиболее ценную загрузку, вес которой не превышает W кг. Если вынуть из рюкзака предмет под номером j , то остальные предметы должны быть наиболее ценными, вес которых не превышает $W - w_j$ и которые можно составить из $n - 1$ исходных предметов, из множества

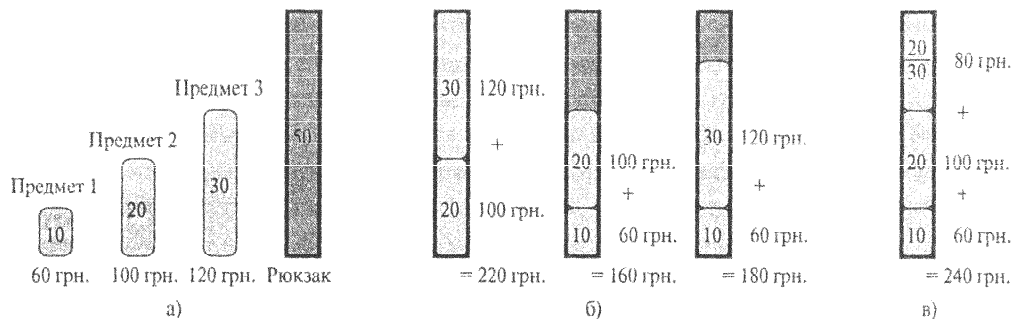


Рис. 16.2. Жадная стратегия не работает для дискретной задачи о рюкзаке

которых исключен предмет под номером j . Для аналогичной непрерывной задачи можно привести такие же рассуждения. Если удалить из оптимально загруженного рюкзака товар с индексом j , который весит w кг, остальное содержимое рюкзака будет наиболее ценным, состоящим из $n - 1$ исходных товаров, вес которых не превышает величину $W - w$ плюс $w_j - w$ кг товара с индексом j .

Несмотря на сходство сформулированных выше задач, непрерывная задача о рюкзаке допускает решение на основе жадной стратегии, а дискретная — нет. Чтобы решить непрерывную задачу, вычислим сначала стоимость единицы веса v_i/w_i каждого товара. Придерживаясь жадной стратегии, вор сначала загружает как можно больше товара с максимальной удельной стоимостью (за единицу веса). Если запас этого товара исчерпается, а грузоподъемность рюкзака — нет, он загружает как можно больше товара, удельная стоимость которого будет второй по величине. Так продолжается до тех пор, пока вес товара не достигает допустимого максимума. Таким образом, вместе с сортировкой товаров по их удельной стоимости время работы алгоритма равно $O(n \lg n)$. Доказательство того, что непрерывная задача о рюкзаке обладает свойством жадного выбора, предлагается провести в упражнении 16.2-1.

Чтобы убедиться, что подобная жадная стратегия не работает в целочисленной задаче о рюкзаке, рассмотрим пример, проиллюстрированный на рис. 16.2а. Имеется 3 предмета и рюкзак, способный выдержать 50 кг. Первый предмет весит 10 кг и стоит 60 грн. Второй предмет весит 20 кг и стоит 100 грн. Третий предмет весит 30 кг и стоит 120 грн. Таким образом, один килограмм первого предмета стоит 6 грн, что превышает аналогичную величину для второго (5 грн/кг) и третьего (4 грн/кг) предметов. Поэтому жадная стратегия должна состоять в том, чтобы сначала взять первый предмет. Однако, как видно из рис. 16.2б, оптимальное решение — взять второй и третий предмет, а первый — оставить. Оба возможных решения, включающих в себя первый предмет, не являются оптимальными.

Однако для аналогичной непрерывной задачи жадная стратегия, при которой сначала загружается первый предмет, позволяет получить оптимальное решение,

как видно из рис. 16.2а. Если же сначала загрузить первый предмет в дискретной задаче, то невозможно будет загрузить рюкзак до отказа и оставшееся пустое место приведет к снижению эффективной стоимости единицы веса при такой загрузке. Принимая в дискретной задаче решение о том, следует ли положить тот или иной предмет в рюкзак, необходимо сравнить решение подзадачи, в которую входит этот предмет, с решением подзадачи, в которой он отсутствует, и только после этого можно делать выбор. В сформулированной таким образом задаче возникает множество перекрывающихся подзадач, что служит признаком применимости динамического программирования. И в самом деле, целочисленную задачу о рюкзаке можно решить с помощью динамического программирования (см. упражнение 16.2-2).

Упражнения

- 16.2-1. Докажите, что непрерывная задача о рюкзаке обладает свойством жадного выбора.
- 16.2-2. Приведите решение дискретной задачи о рюкзаке с помощью динамического программирования. Время работы вашего алгоритма должно быть равно $O(nW)$, где n — количество элементов, а W — максимальный вес предметов, которые вор может положить в свой рюкзак.
- 16.2-3. Предположим, что в дискретной задаче о рюкзаке порядок сортировки по увеличению веса совпадает с порядком сортировки по уменьшению стоимости. Сформулируйте эффективный алгоритм для поиска оптимального решения этой разновидности задачи о рюкзаке и обоснуйте его корректность.
- 16.2-4. Профессор едет из Киева в Москву на автомобиле. У него есть карта, где обозначены все заправки с указанием расстояния между ними. Известно, что если топливный бак заполнен, то автомобиль может проехать n километров. Профессору хочется как можно реже останавливаться на заправках. Сформулируйте эффективный метод, позволяющий профессору определить, на каких заправках следует заправлять топливо, чтобы количество остановок оказалось минимальным. Докажите, что разработанная стратегия дает оптимальное решение.
- 16.2-5. Разработайте эффективный алгоритм, который по заданному множеству $\{x_1, x_2, \dots, x_n\}$ действительных точек на числовой прямой позволил бы найти минимальное множество закрытых интервалов единичной длины, содержащих все эти точки. Обоснуйте корректность алгоритма.
- ★ 16.2-6. Покажите, как решить непрерывную задачу о рюкзаке за время $O(n)$.
- 16.2-7. Предположим, что имеется два множества, A и B , каждое из которых состоит из n положительных целых чисел. Порядок элементов каждого

множества можно менять произвольным образом. Пусть a_i — i -й элемент множества A , а b_i — i -й элемент множества B после переупорядочения. Составим величину $\prod_{i=1}^n a_i^{b_i}$. Сформулируйте алгоритм, который бы максимизировал эту величину. Докажите, что ваш алгоритм действительно максимизирует указанную величину, и определите время его работы.

16.3 Коды Хаффмана

Коды Хаффмана (Huffman codes) — широко распространенный и очень эффективный метод сжатия данных, который, в зависимости от характеристик этих данных, обычно позволяет сэкономить от 20% до 90% объема. Мы рассматриваем данные, представляющие собой последовательность символов. В жадном алгоритме Хаффмана используется таблица, содержащая частоты появления тех или иных символов. С помощью этой таблицы определяется оптимальное представление каждого символа в виде бинарной строки.

Предположим, что имеется файл данных, состоящий из 100 000 символов, который требуется сжать. Символы в этом файле встречаются с частотой, представленной в табл. 16.1. Таким образом, всего файл содержит шесть различных символов, а, например, символ a встречается в нем 45 000 раз.

Таблица 16.1. Задача о кодировании последовательности символов

	a	b	c	d	e	f
Частота (в тысячах)	45	13	12	16	9	5
Кодовое слово фиксированной длины	000	001	010	011	100	101
Кодовое слово переменной длины	0	101	100	111	1101	1100

Существует множество способов представить подобный файл данных. Рассмотрим задачу по разработке *бинарного кода* символов (binary character code; или для краткости просто *кода*), в котором каждый символ представляется уникальной бинарной строкой. Если используется *код фиксированной длины*, или *равномерный код* (fixed-length code), то для представления шести символов понадобится 3 бита: $a = 000$, $b = 001$, ..., $f = 101$. При использовании такого метода для кодирования всего файла понадобится 300 000 битов. Можно ли добиться лучших результатов?

С помощью *кода переменной длины*, или *неравномерного кода* (variable-length code), удастся получить значительно лучшие результаты, чем с помощью кода фиксированной длины. Это достигается за счет того, что часто встречающимся символам сопоставляются короткие кодовые слова, а редко встречающимся — длинные. Такой код представлен в последней строке табл. 16.1. В нем символ a

представлен 1-битовой строкой 0, а символ *f* — 4-битовой строкой 1100. Для представления файла с помощью этого кода потребуется

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224\,000 \text{ битов.}$$

Благодаря этому дополнительно экономится 25% объема. Кстати, как мы вскоре убедимся, для рассматриваемого файла это оптимальная кодировка символов.

Префиксные коды

Мы рассматриваем только те коды, в которых никакое кодовое слово не является префиксом какого-то другого кодового слова. Такие коды называются *префиксными* (prefix codes)². Можно показать (хотя здесь мы не станем этого делать), что оптимальное сжатие данных, которого можно достичь с помощью кодов, всегда достижимо при использовании префиксного кода, поэтому рассмотрение одних лишь префиксных кодов не приводит к потере общности.

Для любого бинарного кода символов кодирование текста — очень простой процесс: надо просто соединить кодовые слова, представляющие каждый символ в файле. Например, в кодировке с помощью префиксного кода переменной длины, представленного в табл. 16.1, трехсимвольный файл *abc* имеет вид $0 \cdot 101 \cdot 100 = 0101100$, где символом “.” обозначена операция конкатенации.

Предпочтение префиксным кодам отдается из-за того, что они упрощают декодирование. Поскольку никакое кодовое слово не выступает в роли префикса другого, кодовое слово, с которого начинается закодированный файл, определяется однозначно. Начальное кодовое слово легко идентифицировать, преобразовать его в исходный символ и продолжить декодирование оставшейся части закодированного файла. В рассматриваемом примере строка 001011101 однозначно раскладывается на подстроки $0 \cdot 0 \cdot 101 \cdot 1101$, что декодируется как *aa**be*.

Для упрощения процесса декодирования требуется удобное представление префиксного кода, чтобы кодовое слово можно было легко идентифицировать. Одним из таких представлений является бинарное дерево, листьями которого являются кодируемые символы. Бинарное кодовое слово, представляющее символ, интерпретируется как путь от корня к этому символу. В такой интерпретации 0 означает “перейти к левому дочернему узлу”, а 1 — “перейти к правому дочернему узлу”. На рис. 16.3 показаны такие деревья для двух кодов, взятых из нашего примера. Каждый лист на рисунке помечен соответствующим ему символом и частотой появления, а внутренний узел — суммой частот листьев его поддерева. В части *a* рисунка приведено дерево, соответствующее коду фиксированной длины, где $a = 000, \dots, f = 101$. В части *b* показано дерево, соответствующее оптимальному префиксному коду $a = 0, b = 101, \dots, f = 1100$. Заметим,

²Возможно, лучше подошло бы название “беспрефиксные коды”, однако “префиксные коды” — стандартный в литературе термин.

Построение кода Хаффмана

Хаффман изобрел жадный алгоритм, позволяющий составить оптимальный префиксный код, который получил название *код Хаффмана*. В соответствии с линией, намеченной в разделе 16.2, доказательство корректности этого алгоритма основывается на свойстве жадного выбора и оптимальной подструктуре. Вместо того чтобы демонстрировать, что эти свойства выполняются, а затем разрабатывать псевдокод, сначала мы представим псевдокод. Это поможет прояснить, как алгоритм осуществляет жадный выбор.

В приведенном ниже псевдокоде предполагается, что C — множество, состоящее из n символов, и что каждый из символов $c \in C$ — объект с определенной частотой $f(c)$. В алгоритме строится дерево T , соответствующее оптимальному коду, причем построение идет в восходящем направлении. Процесс построения начинается с множества, состоящего из $|C|$ листьев, после чего последовательно выполняется $|C| - 1$ операций “слияния”, в результате которых образуется конечное дерево. Для идентификации двух наименее часто встречающихся объектов, подлежащих слиянию, используется очередь с приоритетами Q , ключами в которой являются частоты f . В результате слияния двух объектов образуется новый объект, частота появления которого является суммой частот объединенных объектов:

HUFFMAN(C)

```

1   $n \leftarrow |C|$ 
2   $Q \leftarrow C$ 
3  for  $i \leftarrow 1$  to  $n - 1$ 
4      do Выделить память для узла  $z$ 
5           $left[z] \leftarrow x \leftarrow \text{EXTRACT\_MIN}(Q)$ 
6           $right[z] \leftarrow y \leftarrow \text{EXTRACT\_MIN}(Q)$ 
7           $f[z] \leftarrow f[x] + f[y]$ 
8           $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT\_MIN}(Q)$            ▷ Возврат корня дерева
```

Для рассмотренного ранее примера алгоритм Хаффмана выводит результат, приведенный на рис. 16.4. На каждом этапе показано содержимое очереди, элементы которой рассортированы в порядке возрастания их частот. На каждом шаге работы алгоритма объединяются два объекта (дерева) с самыми низкими частотами. Листья изображены в виде прямоугольников, в каждом из которых указана буква и соответствующая ей частота. Внутренние узлы представлены кругами, содержащими сумму частот дочерних узлов. Ребро, соединяющее внутренний узел с левым дочерним узлом, имеет метку 0, а ребро, соединяющее его с правым дочерним узлом, — метку 1. Слово кода для буквы образуется последовательностью меток на ребрах, соединяющих корень с листом, представляющим эту букву. По-

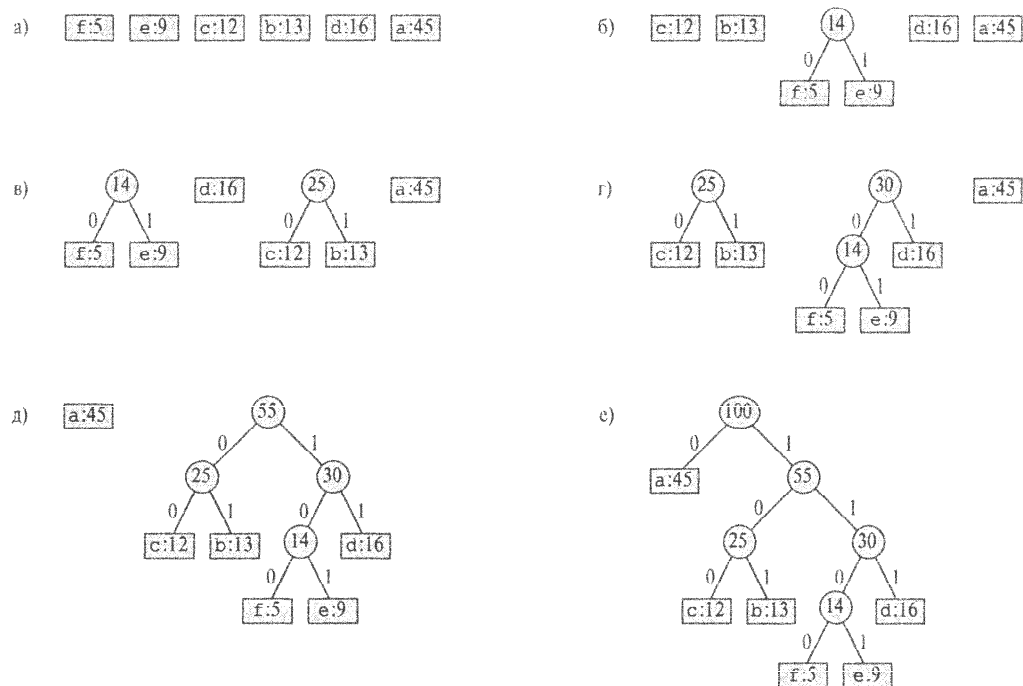


Рис. 16.4. Этапы работы алгоритма Хаффмана для частот, заданных в табл. 16.1

сколько данное множество содержит шесть букв, размер исходной очереди равен 6 (часть *a* рисунка), а для построения дерева требуется пять слияний. Промежуточные этапы изображены в частях *b–d*. Конечное дерево (рис. 16.4*e*) представляет оптимальный префиксный код. Как уже говорилось, слово кода для буквы — это последовательность меток на пути от корня к листу с этой буквой.

В строке 2 инициализируется очередь с приоритетами Q , состоящая из элементов множества C . Цикл **for** в строках 3–8 поочередно извлекает по два узла, x и y , которые характеризуются в очереди наименьшими частотами, и заменяет их в очереди новым узлом z , представляющим объединение упомянутых выше элементов. Частота появления z вычисляется в строке 7 как сумма частот x и y . Узел x является левым дочерним узлом z , а y — его правым дочерним узлом. (Этот порядок является произвольным; перестановка левого и правого дочерних узлов приводит к созданию другого кода с той же стоимостью.) После $n - 1$ объединений в очереди остается один узел — корень дерева кодов, который возвращается в строке 9.

При анализе времени работы алгоритма Хаффмана предполагается, что Q реализована как бинарная неубывающая пирамида (см. главу 6). Для множества C , состоящего из n символов, инициализацию очереди Q в строке 2 можно выполнить

за время $O(n)$ с помощью процедуры BUILD_MIN_HEAP из раздела 6.3. Цикл `for` в строках 3–8 выполняется ровно $n - 1$ раз, и поскольку для каждой операции над пирамидой требуется время $O(\lg n)$, вклад цикла во время работы алгоритма равен $O(n \lg n)$. Таким образом, полное время работы процедуры HUFFMAN с входным множеством, состоящим из n символов, равно $O(n \lg n)$.

Корректность алгоритма Хаффмана

Чтобы доказать корректность жадного алгоритма HUFFMAN, покажем, что в задаче о построении оптимального префиксного кода проявляются свойства жадного выбора и оптимальной подструктуры. В сформулированной ниже лемме показано соблюдение свойства жадного выбора.

Лемма 16.2. Пусть C — алфавит, каждый символ $c \in C$ которого встречается с частотой $f[c]$. Пусть x и y — два символа алфавита C с самыми низкими частотами. Тогда для алфавита C существует оптимальный префиксный код, кодовые слова символов x и y в котором имеют одинаковую длину и отличаются лишь последним битом.

Доказательство. Идея доказательства состоит в том, чтобы взять дерево T , представляющее произвольный оптимальный префиксный код, и преобразовать его в дерево, представляющее другой оптимальный префиксный код, в котором символы x и y являются листьями с общим родительским узлом, причем в новом дереве эти листья находятся на максимальной глубине.

Пусть a и b — два символа, представленные листьями с общим родительским узлом, которые находятся на максимальной глубине дерева T . Предположим без потери общности, что $f[a] \leq f[b]$ и $f[x] \leq f[y]$. Поскольку $f[x]$ и $f[y]$ — две самые маленькие частоты (в указанном порядке), а $f[a]$ и $f[b]$ — две произвольные частоты, то выполняются соотношения $f[x] \leq f[a]$ и $f[y] \leq f[b]$. Как показано на рис. 16.5, в результате перестановки в дереве T листьев a и x получается дерево T' , а при последующей перестановке в дереве T' листьев b и y получается дерево T'' . Согласно уравнению (16.5), разность стоимостей деревьев T и T' равна

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) = \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_{T'}(x) - f[a] d_{T'}(a) = \\ &= f[x] d_T(x) + f[a] d_T(a) - f[x] d_T(a) - f[a] d_T(x) = \\ &= (f[a] - f[x]) (d_T(a) - d_T(x)) \geq 0, \end{aligned}$$

поскольку величины $f[a] - f[x]$ и $d_T(a) - d_T(x)$ неотрицательны. Величина $f[a] - f[x]$ неотрицательна, потому что x — лист с минимальной частотой, величина $d_T(a) - d_T(x)$ неотрицательна, потому что a — лист на максимальной глубине

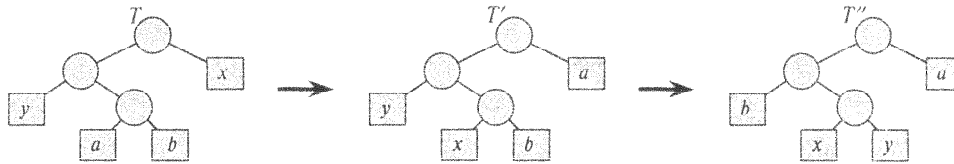


Рис. 16.5. Иллюстрация ключевых этапов доказательства леммы 16.2

в дереве T . Аналогично, перестановка листьев y и b не приведет к увеличению стоимости, поэтому величина $B(T') - B(T'')$ неотрицательна. Таким образом, выполняется неравенство $B(T'') \leq B(T)$, и поскольку T — оптимальное дерево, то должно также выполняться неравенство $B(T) \leq B(T'')$, откуда следует, что $B(T'') = B(T)$. Таким образом, T'' — оптимальное дерево, в котором x и y — находящиеся на максимальной глубине дочерние листья одного и того же узла, что и доказывает лемму. \square

Из леммы 16.2 следует, что процесс построения оптимального дерева путем объединения узлов без потери общности можно начать с жадного выбора, при котором объединению подлежат два символа с наименьшими частотами. Почему такой выбор будет жадным? Стоимость объединения можно рассматривать как сумму частот входящих в него элементов. В упражнении 16.3-3 предлагается показать, что полная стоимость сконструированного таким образом дерева равна сумме стоимостей его составляющих. Из всевозможных вариантов объединения на каждом этапе в процедуре HUFFMAN выбирается тот, в котором получается минимальная стоимость.

В приведенной ниже лемме показано, что задача о составлении оптимальных префиксных кодов обладает свойством оптимальной подструктуры.

Лемма 16.3. Пусть дан алфавит C , в котором для каждого символа $c \in C$ определены частоты $f[c]$. Пусть x и y — два символа из алфавита C с минимальными частотами. Пусть C' — алфавит, полученный из алфавита C путем удаления символов x и y и добавления нового символа z , так что $C' = C - \{x, y\} \cup \{z\}$. По определению частоты f в алфавите C' совпадают с частотами в алфавите C , за исключением частоты $f[z] = f[x] + f[y]$. Пусть T' — произвольное дерево, представляющее оптимальный префиксный код для алфавита C' . Тогда дерево T , полученное из дерева T' путем замены листа z внутренним узлом с дочерними элементами x и y , представляет оптимальный префиксный код для алфавита C .

Доказательство. Сначала покажем, что стоимость $B(T)$ дерева T можно выразить через стоимость $B(T')$ дерева T' , рассматривая стоимости компонентов из уравнения (16.5). Для каждого символа $c \in C - \{x, y\}$ выполняется соотношение

$d_T(c) = d_{T'}(c)$, следовательно, $f[c]d_T(c) = f[c]d_{T'}(c)$. Поскольку $d_T(x) = d_T(y) = d_{T'}(z) + 1$, получаем соотношение

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) = \\ &= f[z]d_{T'}(z) + (f[x] + f[y]), \end{aligned}$$

из которого следует равенство

$$B(T) = B(T') + f[x] + f[y],$$

или

$$B(T') = B(T) - f[x] - f[y].$$

Докажем лемму методом от противного. Предположим, дерево T не представляет оптимальный префиксный код для алфавита C . Тогда существует дерево T'' , для которого справедливо неравенство $B(T'') < B(T)$. Согласно лемме 16.2, x и y без потери общности можно считать дочерними элементами одного и того же узла. Пусть дерево T''' получено из дерева T'' путем замены элементов x и y листом z с частотой $f[z] = f[x] + f[y]$. Тогда можно записать:

$$B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T'),$$

что противоречит предположению о том, что дерево T' представляет оптимальный префиксный код для алфавита C' . Таким образом, дерево T должно представлять оптимальный префиксный код для алфавита C . \square

Теорема 16.4. Процедура HUFFMAN дает оптимальный префиксный код.

Доказательство. Справедливость теоремы непосредственно следует из лемм 16.2 и 16.3. \square

Упражнения

- 16.3-1. Докажите, что бинарное дерево, которое не является полным, не может соответствовать оптимальному префиксному коду.
- 16.3-2. Определите оптимальный код Хаффмана для представленного ниже множества частот, основанного на первых восьми числах Фибоначчи.

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Попытайтесь обобщить ответ на случай первых n чисел Фибоначчи.

- 16.3-3. Докажите, что полную стоимость дерева, представляющего какой-нибудь код, можно также вычислить как сумму комбинаций частот двух дочерних узлов по всем внутренним узлам.

- 16.3-4. Докажите, что если символы отсортированы в порядке монотонного убывания частот, то существует оптимальный код, длина слов в котором — монотонно неубывающая величина.
- 16.3-5. Предположим, имеется оптимальный префиксный код для множества $C = \{0, 1, \dots, n-1\}$ и мы хотим передать его, используя как можно меньше битов. Покажите, как представить любой оптимальный префиксный код для множества C с помощью всего $2n - 1 + n \lceil \lg n \rceil$ битов. (Указание: для представления структуры дерева, определяемой его обходом, достаточно $2n - 1$ битов.)
- 16.3-6. Обобщите алгоритм Хаффмана для кодовых слов в троичной системе счисления (т.е. слов, в которых используются символы 0, 1 и 2) и докажите, что с его помощью получается оптимальный троичный код.
- 16.3-7. Предположим, что файл данных содержит последовательность 8-битовых символов, причем все 256 символов встречаются достаточно часто: максимальная частота превосходит минимальную менее чем в два раза. Докажите, что в этом случае кодирование Хаффмана по эффективности не превышает обычный 8-битовый код фиксированной длины.
- 16.3-8. Покажите, что в среднем ни одна схема сжатия не в состоянии даже на один бит сжать файл, состоящий из случайных 8-битовых символов. (Указание: сравните количество возможных файлов с количеством сжатых файлов.)

★ 16.4 Теоретические основы жадных методов

В этом разделе в общих чертах рассматривается элегантная теория жадных алгоритмов. Она оказывается полезной, когда нужно определить, когда жадный метод дает оптимальное решение. Эта теория содержит комбинаторные структуры, известные под названием “матроиды”. Несмотря на то, что рассматриваемая теория не описывает все случаи, для которых применим жадный метод (например, она не описывает задачу о выборе процессов из раздела 16.1 или задачу о кодах Хаффмана из раздела 16.3), она находит применение во многих случаях, представляющих практический интерес. Более того, эта теория быстро развивается и обобщается, находя все больше приложений. В конце данной главы приводятся ссылки на литературу, посвященную этой теме.

Матроиды

Матроид (matroid) — это упорядоченная пара $M = (S, \mathcal{I})$, удовлетворяющая сформулированным ниже условиям.