Previous Chapter Return to Table of Contents Next Chapter

CHAPTER 17: GREEDY ALGORITHMS

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do. A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. This chapter explores optimization problems that are solvable by greedy algorithms.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We shall first examine in Section 17.1 a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes a solution. Next, Section 17.2 reviews some of the basic elements of the greedy approach. Section 17.3 presents an important application of greedy techniques: the design of data-compression (Huffman) codes. In Section 17.4, we investigate some of the theory underlying combinatorial structures called "matroids" for which a greedy algorithm always produces an optimal solution. Finally, Section 17.5 illustrates the application of matroids using the problem of scheduling unit-time tasks with deadlines and penalties.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that can be viewed as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 24), Dijkstra's algorithm for shortest paths form a single source (Chapter 25), and Chvátal's greedy set-covering heuristic (Chapter 37). Minimum spanning trees form a classic example of the greedy method. Although this chapter and Chapter 24 can be read independently of each other, you may find it useful to read them together.

17.1 An activity-selection problem

Our first example is the problem of scheduling a resource among several competing activities. We shall find that a greedy algorithm provides an elegant and simple method for selecting a maximum-size set of mutually compatible activities.

Suppose we have a set $S = \{ 1, 2, \ldots, n\}$ of n proposed ***activities*** that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity i has a ***start time*** s_i and a ***finish time*** $\hat{a}_i$, where $s_i \le \hat{a}_i$. If selected, activity i takes place during the half-open time interval $[s_i,\hat{a}_i)$. Activities i and j are ***compatible*** if the intervals $[s_i, \hat{a}_i)$ and $[s_j,\hat{a}_j)$ do not overlap (i.e., i and j are compatible if $s_i \ge \hat{a}_j$ or $s_j \ge \hat{a}_i$). The ***activity-selection problem*** is to select a maximum-size set of mutually compatible activities.

A greedy algorithm for the activity-selection problem is given in the following pseudocode. We assume that the input activities are in order by increasing finishing time:

$\hat{a}_1 \leq \hat{a}_2 \leq \ldots \leq \hat{a}_n$.

(17.1)

If not, we can sort them into this order in time $O(n \lg n)$, breaking ties arbitrarily. The pseudocode assumes that inputs s and â are represented as arrays.

```
GREEDY-ACTIVITY-SELECTOR(s, f)

1 n ← length[s]

2 A ← {1}

3 j ← 1

4 for i ← 2 to n

5       do if s_i ≥ â_j

6             then A ← A ∪ {i}

7                  j ← i

8 return A
```

The operation of the algorithm is shown in Figure 17.1. The set A collects the selected activities. The variable j specifies the most recent addition to A. Since the activities are considered in order of nondecreasing finishing time, f_j is always the maximum finishing time of any activity in A. That is,

$\hat{a}_j = \max\{f_k : K \in A\}$.

(17.2)

Lines 2-3 select activity 1, initialize A to contain just this activity, and initialize j to this activity. Lines 4-7 consider each activity i in turn and add i to A if it is compatible with all previously selected activities. To see if activity i is compatible with every activity currently in A, it suffices by equation (17.2) to check (line 5) that its start time s_i is not earlier than the finish time f_j of the activity most recently added to A. If activity i is compatible, then lines 6-7 add it to A and update j. The GREEDY-ACTIVITY-SELECTOR procedure is quite efficient. It can schedule a set S of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

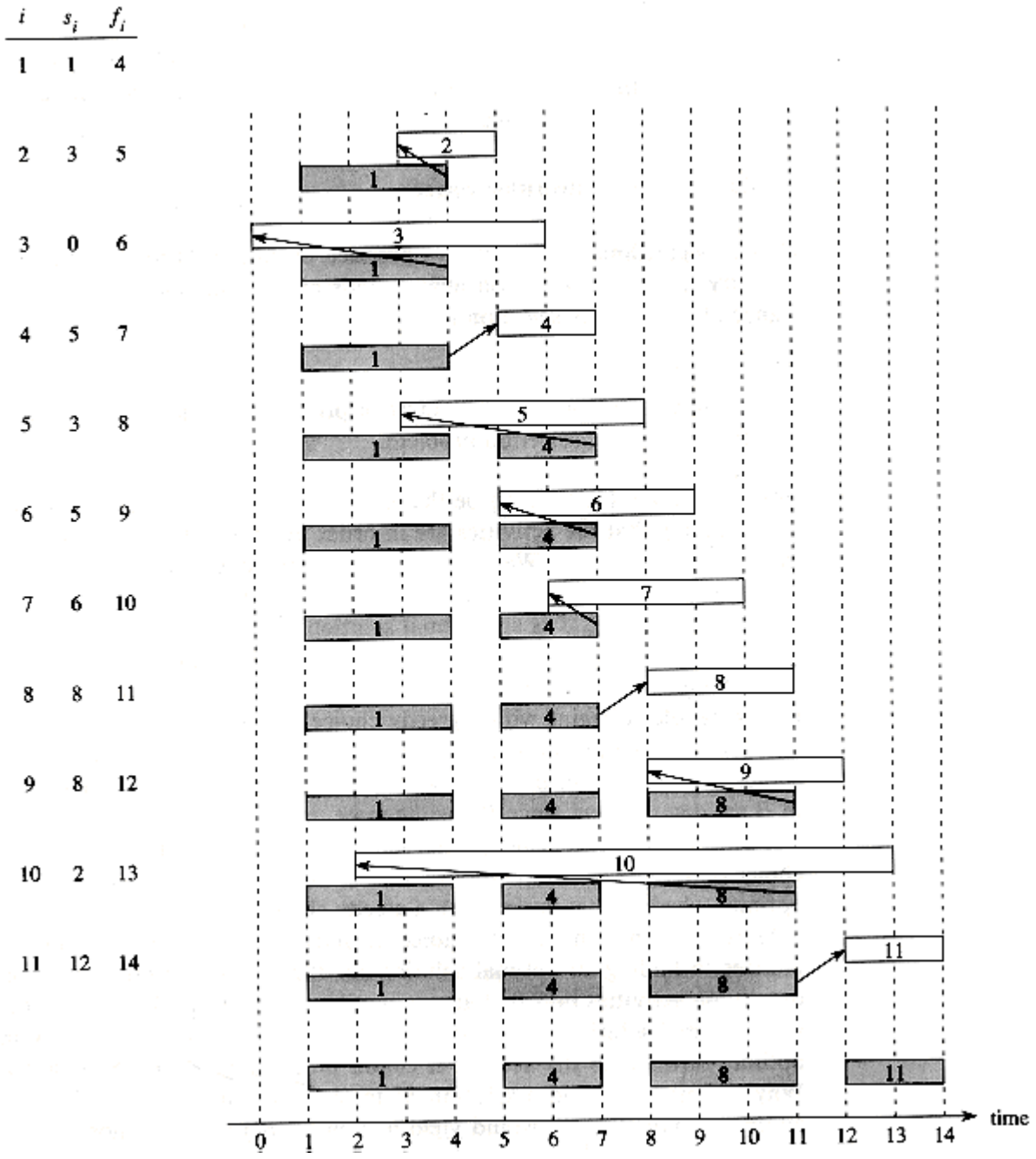

Figure 17.1 The operation of GREEDY-ACTIVITY-SELECTOR on 11 activities given at the left. Each row of the figure corresponds to an iteration of the for loop in lines 4-7. The activities that have been selected to be in set A are shaded, and activity i, shown in white, is being considered. If the starting time s_i of activity i occurs before the finishing time f_j of the most recently selected activity j (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is accepted and put into set A.

The activity picked next by GREEDY-ACTIVITY-SELECTOR is always the one with the earliest finish time that can be legally seheduled. The activity picked is thus a "greedy" choice in the sense that, intuitively, it leaves as much opportunity as possible for the remaining activities to be scheduled. That is, the greedy choice is the one that maximizes the amount of unscheduled time remaining.

Proving the greedy algorithm correct

Greedy algorithms do not always produce optimal solutions. However, GREEDY-ACTIVITY-SELECTOR always finds an optimal solution to an instance of the activity-selection problem.

Theorem 17.1

Algorithm GREEDY-ACTIVITY-SELECTOR produces solutions of maximum size for the activity-selection problem.

Proof Let $S = \{1, 2, \ldots, n\}$ be the set of activities to schedule. Since we are assuming that the activities are in order by finish time, activity 1 has the earliest finish time. We wish to show that there is an optimal solution that begins with a greedy choice, that is, with activity 1.

Suppose that $A \subseteq S$ is an optimal solution to the given instance of the activity-selection problem, and let us order the activities in A by finish time. Suppose further that the first activity in A is activity k. If $k = 1$, then schedule A begins with a greedy choice. If $k \neq 1$, we want to show that there is another optimal solution B to S that begins with the greedy choice, activity 1. Let $B = A - \{k\} \cup \{1\}$. Because $f_i \leq f_k$, the activities in B are disjoint, and since B has the same number of activities as A, it is also optimal. Thus, B is an optimal solution for S that contains the greedy choice of activity 1. Therefore, we have shown that there always exists an optimal schedule that begins with a greedy choice.

Moreover, once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with activity 1. That is, if A is an optimal solution to the original problem S, then $A' = A - \{1\}$ is an optimal solution to the activity-selection problem $S' = \{i \in S: s_i \geq f_1\}$. Why? If we could find a solution B' *to* S' with more activities than A', *adding activity 1 to* B' would yield a solution B to S with more activities than A, thereby contradicting the optimality of A. Therefore, after each greedy choice is made, we are left with an optimization problem of the same form as the original problem. By induction on the number of choices made, making the greedy choice at every step produces an optimal solution.

Exercises

17.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on computing m_i iteratively for $i = 1, 2, \ldots, n$, where m_i is the size of the largest set of mutually compatible activities among activities $\{1, 2, \ldots, i\}$. Assume that the inputs have been sorted as in equation (17.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

17.1-2

Suppose that we have a set of activities to schedule among a large number of lecture halls. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This is also known as the ***interval-graph coloring problem.*** We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices are given the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

17.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from those that are compatible with previously selected activities does not work. Do the same for the approach of always selecting the activity that overlaps the fewest other remaining activities.

17.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point in the algorithm, the choice that seems best at the moment is chosen. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

How can one tell if a greedy algorithm will solve a particular optimization problem? There is no way in general, but there are two ingredients that are exhibited by most problems that lend themselves to a greedy strategy: the greedy-choice property and optimal substructure.

Greedy-choice property

The first key ingredient is the ***greedy-choice property***: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice. Here is where greedy algorithms differ from dynamic programming. In dynamic programming, we make a choice at each step, but the choice may depend on the solutions to subproblems. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.

Of course, we must prove that a greedy choice at each step yields a globally optimal solution, and this is where cleverness may be required. Typically, as in the case of

Theorem 17.1, the proof examines a globally optimal solution. It then shows that the solution can be modified so that a greedy choice is made as the first step, and that this choice reduces the problem to a similar but smaller problem. Then, induction is applied to show that a greedy choice can be used at every step. Showing that a greedy choice results in a similar but smaller problem reduces the proof of correctness to demonstrating that an optimal solution must exhibit optimal substructure.

Optimal substructure

A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms. As an example of optimal substructure, recall that the proof of Theorem 17.1 demonstrated that if an optimal solution A to the activity selection problem begins with activity 1, then the set of activities $A' = A - \{1\}$ is an optimal solution to the activity-selection problem $S' = \{i \in S : s_i \geq \hat{a}_1\}$.

Greedy versus dynamic programming

Because the optimal-substructure property is exploited by both greedy and dynamic-programming strategies, one might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices, or one might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

The ***0-1 knapsack problem*** is posed as follows. A thief robbing a store finds n items; the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take? (This is called the 0-1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once.)

In the ***fractional knapsack problem***, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot, while an item in the fractional knapsack problem is more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j. For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Although the problems are similar, the fractional knapsack problem is solvable by a greedy strategy, whereas the 0-1 problem is not. To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth until he can't carry any more. Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. The proof that the fractional knapsack problem has the greedy-choice property is left as Exercise 17.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 17.2(a). There are 3 items, and the knapsack can hold 50 pounds. Item 1 weighs 10 pounds and is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3 weighs 30 pounds and is worth 120 dollars. Thus, the value per pound of item 1 is 6 dollars per pound, which is greater than the value per pound of either item 2 (5 dollars per pound) or item 3 (4 dollars per pound). The greedy strategy, therefore, would take item 1 first. As can be seen from the case analysis in Figure 17.2(b), however, the optimal solution takes items 2 and 3, leaving 1 behind. The two possible solutions that involve item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 17.2 (c). Taking item 1 doesn't work in the 0-1 problem because the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load. In the 0-1 problem, when we consider an item for inclusion in the knapsack, we must compare the solution to the subproblem in which the item is included with the solution to the subproblem in which the item is excluded before we can make the choice. The problem formulated in this way gives rise to many overlapping subproblems--a hallmark of dynamic programming, and indeed, dynamic programming can be used to solve the 0-1 problem. (See Exercise 17.2-2.)

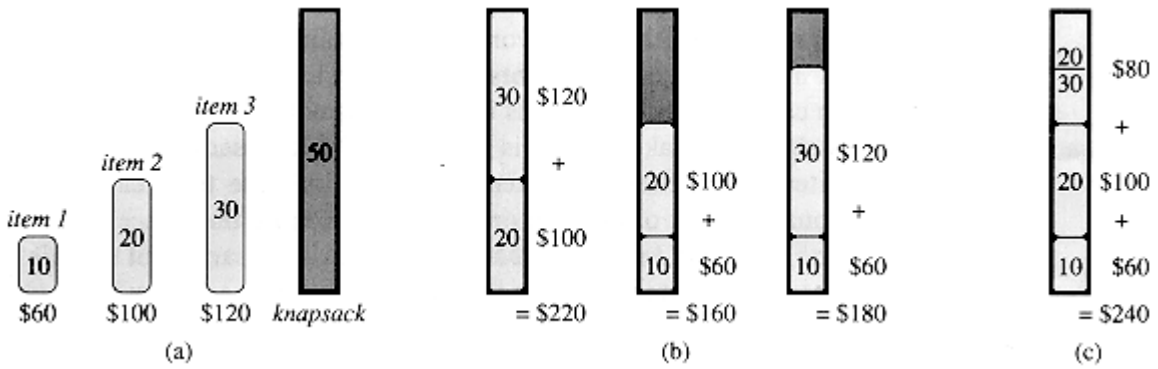

Figure 17.2 The greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Exercises

17.2-1

Prove that the fractional knapsack problem has the greedy-choice property.

17.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(n\ W)$ time, where n is number of items and W is the maximum weight of items that the thief can put in his knapsack.

17.2-3

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

17.2-4

Professor Midas drives an automobile from Newark to Reno along Interstate 80. His car's gas tank, when full, holds enough gas to travel n miles, and his map gives the distances between gas stations on his route. The professor wishes to make as few gas stops as possible along the way. Give an efficient method by which Professor Midas can determine at which gas stations he should stop, and prove that your strategy yields an optimal solution.

17.2-5

Describe an efficient algorithm that, given a set $\{x_1,x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

17.2-6

Show how to solve the fractional knapsack problem in $O(n)$ time. Assume that you have a solution to Problem 10-2.

17.3 Huffman codes

Huffman codes are a widely used and very effective technique for compressing data; savings of 20% to 90% are typical, depending on the characteristics of the file being compressed. Huffman's greedy algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string.

Suppose we have a 100,000-character data file that we wish to store compactly. We

observe that the characters in the file occur with the frequencies given by Figure 17.3. That is, only six different characters appear, and the character `a` occurs 45,000 times.

There are many ways to represent such a file of information. We consider the problem of designing a ***binary character code*** (or ***code*** for short) wherein each character is represented by a unique binary string. If we use a ***fixed-length code***, we need 3 bits to represent six characters: `a` = 000, `b` = 001, . . . , `f` = 101. This method requires 300,000 bits to code the entire file. Can we do better?

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 17.3 A character-coding problem. A data file of 100,000 characters contains only the characters `a-f`, with the frequencies indicated. If each character is assigned a 3-bit codeword, the file can be encoded in 300,000 bits. Using the variable-length code shown, the file can be encoded in 224,000 bits.

A ***variable-length code*** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. Figure 17.3 shows such a code; here the 1-bit string 0 represents `a`, and the 4-bit string 1100 represents `f`. This code requires (45 · 1 + 13 · 3 + 12 · 3 + 16 · 3 + 9 · 4 + 5 · 4) · 1,000 = 224,000 bits to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix codes.***[1] It is possible to show (although we won't do so here) that the optimal data compression achievable by a character code can always be achieved with a prefix code, so there is no loss of generality in restricting attention to prefix codes.

[1] Perhaps "prefix-free codes" would be a better name, but the term "prefix codes" is standard in the literature.

Prefix codes are desirable because they simplify encoding (compression) and decoding. Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with the variable-length prefix code of Figure 17.3, we code the 3-character file `abc` as 0 · 101 · 100 = 0101100, where we use "·" to denote concatenation.

Decoding is also quite simple with a prefix code. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. We can simply identify the initial codeword, translate it back to the original character, remove it from the encoded file, and repeat the decoding process on the remainder of the encoded file. In

our example, the string 001011101 parses uniquely as 0 • 0 • 101 • 1101, which decodes to aabe.

The decoding process needs a convenient representation for the prefix code so that the initial codeword can be easily picked off. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Figure 17.4 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children (see Exercise 17.3-1). The fixed-length code in our example is not optimal since its tree, shown in Figure 17.4(a), is not a full binary tree: there are codewords beginning 10 . . . , but none beginning 11 . . . . Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn, then the tree for an optimal prefix code has exactly $|C|$ *leaves, one for each letter of the alphabet, and exactly* $|C| - 1$ internal nodes.

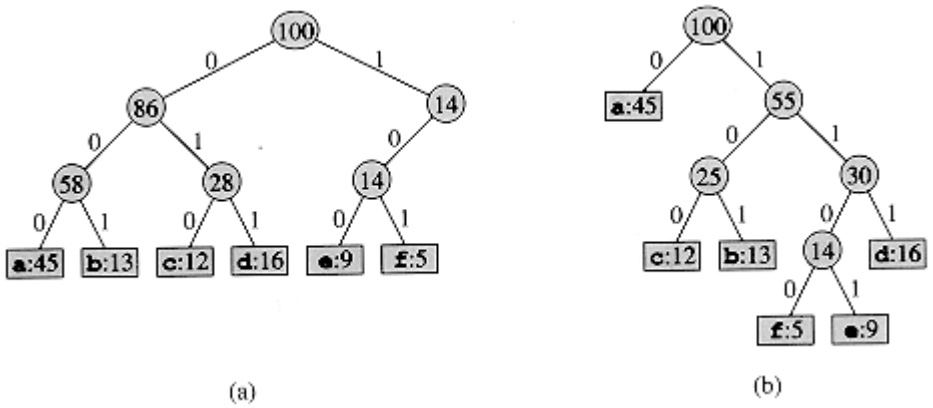

Figure 17.4 Trees corresponding to the coding schemes in Figure 17.3. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the weights of the leaves in its subtree. (a) The tree corresponding to the fixed-length code a = 000, . . . , f = 100. (b) The tree corresponding to the optimal prefix code a = 0, b = 101, . . . , f = 1100

Given a tree T corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file. For each character c in the alphabet C, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} f(c) d_T(c) , \qquad (17.3)$$

(17.3)

which we define as the ***cost*** of the tree T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix code called a ***Huffman code.*** The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree.

In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with a defined frequency $f[c]$. A priority queue Q, keyed on f, is used to identify the two least-frequent objects to merge together. The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN(C)

1 n ← |C|

2 Q ← C

3 for i ← 1 to n - 1

4      do z ← ALLOCATE-NODE()

5         x ← left[z] ← EXTRACT-MIN(Q)

6         y ← right[z] ← EXTRACT-MIN(Q)

7         f[z] ← f[x] + f[y]

8         INSERT(Q, z)

9 return EXTRACT-MIN(Q)
```

For our example, Huffman's algorithm proceeds as shown in Figure 17.5. Since there are 6 letters in the alphabet, the initial queue size is $n = 6$, and 5 merge steps are required to build the tree. The final tree represents the optimal prefix code. The codeword for a letter is the sequence of edge labels on the path from the root to the letter.

Line 2 initializes the priority queue Q with the characters in C. The **for** loop in lines 3-8 repeatedly extracts the two nodes x and y of lowest frequency from the queue, and replaces them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 7. The node z has x as its left child and y as its right child. (This order is arbitrary; switching the left and right child of any node yields a different code of the same cost.) After $n - 1$ mergers, the one node left in the queue--the root of the code tree--is returned in line 9.

The analysis of the running time of Huffman's algorithm assumes that Q is implemented as a binary heap (see Chapter 7). For a set C of n characters, the initialization of Q in line 2 can be performed in $O(n)$ time using the BUILD-HEAP procedure in Section 7.3. The **for** loop in lines 3-8 is executed exactly $|n| - 1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties. The next lemma shows that the greedy-choice property holds.

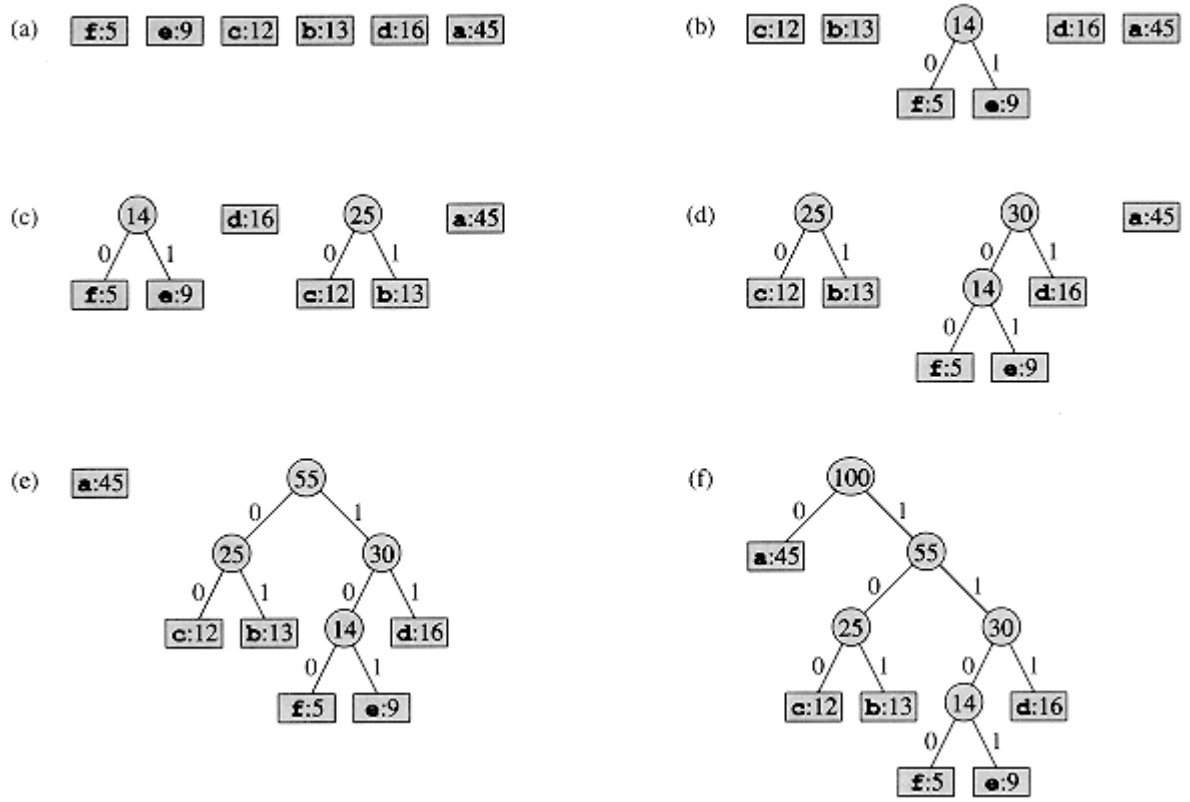

Figure 17.5 The steps of Huffman's algorithm for the frequencies given in Figure 17.3. Each part shows the contents of the queue sorted into increasing order by frequency. At each step, the two trees with lowest frequencies are merged. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of its children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. (a) The initial set of n = 6 nodes, one for each letter. (b)-(e) Intermediate stages.(f) The final tree.

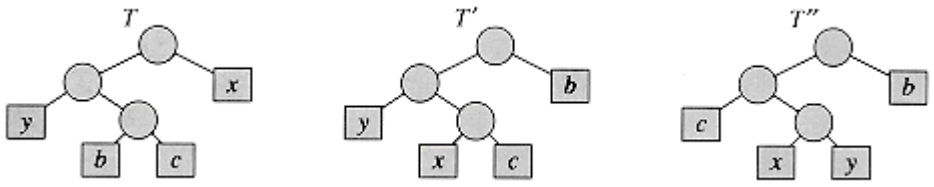

Figure 17.6 An illustration of the key step in the proof of Lemma 17.2. In the optimal tree T, leaves b and c are two of the deepest leaves and are siblings. Leaves x and y are the two leaves that Huffman's algorithm merges together first; they appear in arbitrary positions in T. Leaves b and x are swapped to obtain tree T'. Then, leaves c and y are swapped to obtain tree T\". Since each swap does not increase the cost, the

resulting tree T\" is also an optimal tree.

Lemma 17.2

Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. If we can do this, then their codewords will have the same length and differ only in the last bit.

Let b and c be two characters that are sibling leaves of maximum depth in T. Without loss of generality, we assume that $f[b] \le f[c]$ and $f[x] \le f[y]$. Since $f[x]$ and $f[y]$ are the two lowest leaf frequencies, in order, and $f[b]$ and $f[c]$ are two arbitrary frequencies, in order, we have $f[x] \le f[b]$ and $f[y] \le f[c]$. As shown in Figure 17.6, we exchange the positions in T of b and x to produce a tree T', and then we exchange the positions in T' of c and y to produce a tree $T\"$. By equation (17.3), the difference in cost between T and T' *is*

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

because both $f[b] - f[x]$ and $d_T[b] - d_T[x]$ are nonnegative. More specifically, $f[b] - f[x]$ is nonnegative because x is a minimum-frequency leaf, and $d_T[b] - d_T[x]$ is nonnegative because b is a leaf of maximum depth *in* T. Similarly, because exchanging y and c does not increase the cost, $B(T') - B(T\")$ is nonnegative. Therefore, $B(T\") \le B(T)$, and since T is optimal, $B(T) \le B(T\")$, which implies $B(T\") = B(T)$. Thus, $T\"$ is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows.

Lemma 17.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 17.3-3 shows that the total cost of the tree constructed is the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix codes has the optimal-substructure property.

Lemma 17.3

Let T be a full binary tree representing an optimal prefix code over an alphabet C, where frequency $f[c]$ is defined for each character $c \in C$. Consider any two characters x and y that appear as sibling leaves in T, and let z be their parent. Then, considering z as a character with frequency $f[z] = f[x] + f[y]$, the tree $T' = T - \{x,y\}$ *represents an optimal prefix code for the alphabet* $C' = C - \{x,y\} \cup \{z\}$.

Proof We first show that the cost $B(T)$ of tree T can be expressed in terms of the cost $B(T')$ of tree T' by considering the component costs in equation (17.3). For each $c \in C - \{x,y\}$, we have $d_T(c) = d_{T'}(c)$, and hence $f[c]d_T(c) = f[c]d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$f[x]d_T(x) + f[y]d_T(y) = (f[x]) + (f[y])d_{T'}(z) + 1)$$

$$= f[z]d_{T'}(z) + (f[x] + f[y]),$$

from which we conclude that

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

If T' represents a nonoptimal prefix code for the alphabet C', then there exists a tree $T\"$ whose leaves are characters in C' such that $B(T\") < B(T')$. Since z is treated as a character in C', *it appears as a leaf in* $T|"$. If we add x and y as children of z in $T|"$, then we obtain a prefix code for C with cost $B(T|") + f[x] + f[y] < B(T)$, contradicting the optimality of T. Thus, T' must be optimal for the alphabet C'.

Theorem 17.4

Procedure HUFFMAN produces an optimal prefix code.

Proof Immediate from Lemmas 17.2 and 17.3.

Exercises

17.3-1

Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

17.3-2

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

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

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

17.3-3

Prove the total cost of a tree for a code can also be computed as the sum, over all internal nodes, of the combined frequencies of the two children of the node.

17.3-4

Prove that for an optimal code, if the characters are ordered so that their frequencies are nonincreasing, then their codeword lengths are nondecreasing.

17.3-5

Let $C = \{0, 1, \ldots, n - 1\}$ be a set of characters. Show that any optimal prefix code on C can be represented by a sequence of

$$2n - 1 + n\lceil \lg n \rceil$$

bits. (*Hint:* Use $2n$- 1 bits to specify the structure of the tree, as discovered by a walk of the tree.)

17.3-6

Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

17.3-7

Suppose a data file contains a sequence of 8-bit characters such that all 256 characters are about as common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

17.3-8

Show that no compression scheme can expect to compress a file of randomly chosen 8-bit characters by even a single bit. (*Hint:* Compare the number of files with the number of possible encoded files.)

* 17.4 Theoretical foundations for greedy methods

There is a beautiful theory about greedy algorithms, which we sketch in this section. This theory is useful in determining when the greedy method yields optimal solutions. It involves combinatorial structures known as "matroids." Although this theory does not cover all cases for which a greedy method applies (for example, it does not cover the activity-selection problem of Section 17.1 or the Huffman coding problem of Section 17.3), it does cover many cases of practical interest. Furthermore, this theory is being rapidly developed and extended to cover many more applications; see the notes at the end of this chapter for references.

17.4.1 Matroids

A ***matroid*** is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions.

1. S is a finite nonempty set.

2. $\mathcal{I}$ is a nonempty family of subsets of S, called the ***independent*** subsets of S, such that if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. We say that $\mathcal{I}$ is ***hereditary*** if it satisfies this property. Note that the empty set $\emptyset$ is necessarily a member of $\mathcal{I}$.

3. If $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there is some element $x \in B - A$ such that $A \cup \{x\} \in \mathcal{I}$. We say that M satisfies the ***exchange property***.

The word "matroid" is due to Hassler Whitney. He was studying ***matric matroids***, in which the elements of S are the rows of a given matrix and a set of rows is independent if they are linearly independent in the usual sense. It is easy to show that this structure defines a matroid (see Exercise 17.4-2).

As another illustration of matroids, consider the ***graphic matroid*** $M_G = (S_G, \mathcal{I}_G)$ defined in terms of a given undirected graph $G = (V, E)$ as follows.

- The set S_G is defined to be E, the set of edges of G.

- If A is a subset of E, then $A \in \mathcal{I}_G$ if and only if A is acyclic. That is, a set of edges is independent if and only if it forms a forest.

The graphic matroid M_G is closely related to the minimum-spanning-tree problem, which is covered in detail in Chapter 24.

Theorem 17.5

If G is an undirected graph, then $M_G = (S_G, \mathcal{I}_G)$ is a matroid.

Proof Clearly, $S_G = E$ is a finite set. Furthermore, $\mathcal{I}_G$ is hereditary, since a subset of a forest is a forest. Putting it another way, removing edges from an acyclic set of edges cannot create cycles.

Thus, it remains to show that M_G satisfies the exchange property. Suppose that A and B are forests of G and that $|B| > |A|$. That is, A and B are acyclic sets of edges, and B contains more edges than A does.

It follows from Theorem 5.2 that a forest having k edges contains exactly $|V|$-k *trees. (To prove this another way, begin with* $|V|$ trees and no edges. Then, each edge that is added to the forest reduces the number of trees by one.) Thus, forest A contains $|V| - |A|$ trees, and forest B contains $|V| - |B|$ trees.

Since forest B has fewer trees than forest A does, forest B must contain some tree T whose vertices are in two different trees in forest A. Moreover, since T is connected, it must contain an edge (u, v) such that vertices u and v are in different trees in forest A. Since the edge (u, v) connects vertices in two different trees in forest A, the edge (u, v) can

be added to forest A without creating a cycle. Therefore, M_G satisfies the exchange property, completing the proof that M_G is a matroid.

Given a matroid $M = (S, \mathcal{I})$, we call an element $x \notin A$ an ***extension*** of $A \in \mathcal{I}$ if x can be added to A while preserving independence; that is, x is an extension of A if $A \cup \{x\} \in \mathcal{I}$. As an example, consider a graphic matroid M_G. If A is an independent set of edges, then edge e is an extension of A if and only if e is not in A and the addition of x to A does not create a cycle.

If A is an independent subset in a matroid M, we say that A is ***maximal*** if it has no extensions. That is, A is maximal if it is not contained in any larger independent subset of M. The following property is often useful.

Theorem 17.6

All maximal independent subsets in a matroid have the same size.

Proof Suppose to the contrary that A is a maximal independent subset of M and there exists another larger maximal independent subset B of M. Then, the exchange property implies that A is extendable to a larger independent set $A \cup \{x\}$ for some $x \in B - A$, contradicting the assumption that A is maximal.

As an illustration of this theorem, consider a graphic matroid M_G for a connected, undirected graph G. Every maximal independent subset of M_G must be a free tree with exactly $|V| - 1$ *edges that connects all the vertices of* G. *Such a tree is called a* **spanning tree** *of* G.

We say that a matroid $M = (S, \mathcal{I})$ is ***weighted*** if there is an associated weight function w that assigns a strictly positive weight $w(x)$ to each element $x \in S$. The weight function w extends to subsets of S by summation:

$$w(A) = \sum_{x \in A} w(x)$$

for any $A \subseteq S$. For example, if we let $w(e)$ denote the length of an edge e in a graphic matroid M_G, then $w(A)$ is the total length of the edges in edge set A.

17.4.2 Greedy algorithms on a weighted matroid

Many problems for which a greedy approach provides optimal solutions can be formulated in terms of finding a maximum-weight independent subset in a weighted matroid. That is, we are given a weighted matroid $M = (S, \mathcal{I})$, and we wish to find an independent set $A \in \mathcal{I}$ such that $w(A)$ is maximized. We call such a subset that is independent and has maximum possible weight an ***optimal*** subset of the matroid. Because the weight $w(x)$ of any element $x \in S$ is positive, an optimal subset is always a maximal independent subset--it always helps to make A as large as possible.

For example, in the ***minimum-spanning-tree problem***, we are given a connected undirected graph $G = (V, E)$ and a length function w such that $w(e)$ is the (positive) length of edge e. (We use the term "length" here to refer to the original edge weights for the graph, reserving the term "weight" to refer to the weights in the associated matroid.) We are asked to find a subset of the edges that connects all of the vertices together and has minimum total length. To view this as a problem of finding an optimal subset of a matroid, consider the weighted matroid M_G with weight function w', *where* $w'(e) = w_0 - w(e)$ and w_0 is larger than the maximum length of any edge. In this weighted matroid, all weights are positive and an optimal subset is a spanning tree of minimum total length in the original graph. More specifically, each maximal independent subset A corresponds to a spanning tree, and since

$$w'(A) = (|V| - 1)w_0 - w(A)$$

for any maximal independent subset A, the independent subset that maximizes $w'(A)$ must minimize $w(A)$. Thus, any algorithm that can find an optimal subset A in an arbitrary matroid can solve the minimum-spanning-tree problem.

Chapter 24 gives algorithms for the minimum-spanning-tree problem, but here we give a greedy algorithm that works for any weighted matroid. The algorithm takes as input a weighted matroid $M = (S, \mathcal{I})$ with an an associated positive weight function w, and it returns an optimal subset A. In our pseudocode, we denote the components of M by $S[M]$ and $\mathcal{I}[M]$ and the weight function by w. The algorithm is greedy because it considers each element $x \in S$ in turn in order of nonincreasing weight and immediately adds it to the set A being accumulated if $A \cup \{x\}$ is independent.

```
GREEDY(M, w)
1  A ← ∅
2  sort S[M] into nonincreasing order by weight w
3  for each x ∈ S[M], taken in nonincreasing order by weight w(x)
4       do if A ∪ {x} ∈ I[M]
5            then A ← A ∪ {x}
6  return A
```

The elements of S are considered in turn, in order of nonincreasing weight. If the element x being considered can be added to A while maintaining A's independence, it is. Otherwise, x is discarded. Since the empty set is independent by the definition of a matroid, and since x is only added to A if $A \cup \{x\}$ is independent, the subset A is always independent, by induction. Therefore, GREEDY always returns an independent subset A. We shall see in a moment that A is a subset of maximum possible weight, so that A is an optimal subset.

The running time of GREEDY is easy to analyze. Let n denote $|S|$. The sorting phase of GREEDY takes time $O(n \lg n)$. Line 4 is executed exactly n times, once for each element of S. Each execution of line 4 requires a check on whether or not the set $A \cup \{x\}$ is independent. If each such check takes time $O(f(n))$, the entire algorithm runs in time $O(n \lg n + nf(n))$.

We now prove that GREEDY returns an optima1 subset.

Lemma 17.7

Suppose that $M = (S, \mathcal{I})$ is a weighted matroid with weight function w and that S is sorted into nonincreasing order by weight. Let x be the first element of S such that $\{x\}$ is independent, if any such x exists. If x exists, then there exists an optimal subset A of S that contains x.

Proof If no such x exists, then the only independent subset is the empty set and we're done. Otherwise, let B be any nonempty optimal subset. Assume that $x \notin B$; otherwise, we let $A = B$ and we're done.

No element of B has weight greater than $w(x)$. To see this, observe that $y \in B$ implies that $\{y\}$ is independent, since $B \in \mathcal{I}$ and $\mathcal{I}$ and I is hereditary. Our choice of x therefore ensures that $w(x) \geq w(y)$ for any $y \in B$.

Construct the set A as follows. Begin with $A = \{x\}$. By the choice of x, A is independent. Using the exchange property, repeatedly find a new element of B that can be added to A until $|A| = |B|$ while preserving the independence of A. Then, $A = B - \{y\} \cup \{x\}$ for some $y \in B$, and so

$$w(A) = w(B) - w(y) + w(x)$$

$$\geq w(B) .$$

Because B is optimal, A must also be optimal, and because $x \in A$, the lemma is proven.

We next show that if an element is not an option initially, then it cannot be an option later.

Lemma 17.8

Let $M = (S, \mathcal{I})$ be any matroid. If x is an element of S such that x is not an extension of $\emptyset$, then x is not an extension of any independent subset A of S.

Proof The proof is by contradiction. Assume that x is an extension of A but not of $\emptyset$. Since x is an extension of A, we have that $A \cup \{x\}$ is independent. Since $\mathcal{I}$ is hereditary, $\{x\}$ must be independent, which contradicts the assumption that x is not an extension of $\emptyset$.

Lemma 17.8 says that any element that cannot be used immediately can never be used. Therefore, GREEDY cannot make an error by passing over any initial elements in S that are not an extension of $\emptyset$, since they can never be used.

Lemma 17.9

Let x be the first element of S chosen by GREEDY for the weighted matroid $M = (S, \mathcal{I})$. The remaining problem of finding a maximum-weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid

$M' = (S', \mathcal{I}')$, where

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in \mathcal{I}\} , \\ \mathcal{I}' &= \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\} , \text{ and} \end{aligned}$$

the weight function for *M' is the weight function for* M, *restricted to* S'. (We call *M' the* **contraction** *of* M *by the element* x.*)*

Proof If *A* is any maximum-weight independent subset of *M* containing *x*, then *A'* = A - *{*x*} is an independent subset of* M'. Conversely, any independent subset *A'* *of* M′ yields an independent subset *A* = *A'* ∪ *{*x*} of* M. *Since we have in both cases that* w(A*)* = w(A′) + *w*(*x*), a maximum-weight solution in *M* containing *x* yields a maximum-weight solution in *M'*, and vice versa.

Theorem 17.10

If $M = (S, \mathcal{I})$ is a weighted matroid with weight function *w*, then the call GREEDY(*M*, *w*) returns an optimal subset.

Proof By Lemma 17.8, any elements that are passed over initially because they are not extensions of $\emptyset$ can be forgotten about, since they can never be useful. Once the first element *x* is selected, Lemma 17.7 implies that GREEDY does not err by adding *x* to *A*, since there exists an optimal subset containing *x*. Finally, Lemma 17.9 implies that the remaining problem is one of finding an optimal subset in the matroid *M′* that is the contraction of *M* by *x*. After the procedure GREEDY sets *A* to{*x*}, all of its remaining steps can be interpreted as acting in the matroid $M' = (S', \mathcal{I}')$, because *B* is independent in *M′* if and only if $B \cup \{x\}$ is independent in *M*, for all sets $B \in \mathcal{I}'$. Thus, the subsequent operation of GREEDY will find a maximum-weight independent subset for *M'*, and the overall operation of GREEDY will find a maximum-weight independent subset for *M*.

Exercises

17.4-1

Show that $(S, \mathcal{I}_k)$ is a matroid, where *S* is any finite set and $\mathcal{I}_k$ is the set of all subsets of *S* of size at most *k*, where $k \leq |S|$.

17.4-2

Given an $n \times n$ real-valued matrix *T*, show that $(S, \mathcal{I})$ is a matroid, where *S* is the set of columns of *T* and $A \in \mathcal{I}$ if and only if the columns in *A* are linearly independent.

17.4-3

Show that if $(S, \mathcal{I})$ is a matroid, then $(S, \mathcal{I}')$ is a matroid, where $\mathcal{I}' = \{A' : S - A'$ contains some maximal $A \in \mathcal{I}\}$. That is, the maximal independent sets of $(S', \mathcal{I}')$ are just the complements of the maximal independent sets of $(S, \mathcal{I})$.

17.4-4

Let S be a finite set and let $S_1, S_2, \ldots, S_k$ be a partition of S into nonempty disjoint subsets. Define the structure $(S, \mathcal{I})$ by the condition that $\mathcal{I} = \{A : |A \cap S_i| \leq 1 \text{ for } i = 1, 2, \ldots, k\}$. Show that $(S, \mathcal{I})$ is a matroid. That is, the set of all sets A that contain at most one member in each block of the partition determines the independent sets of a matroid.

17.4-5

Show how to transform the weight function of a weighted matroid problem, where the desired optimal solution is a *minimum-weight* maximal independent subset, to make it an standard weighted-matroid problem. Argue carefully that your transformation is correct.

* 17.5 A task-scheduling problem

An interesting problem that can be solved using matroids is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a deadline and a penalty that must be paid if the deadline is missed. The problem looks complicated, but it can be solved in a surprisingly simple manner using a greedy algorithm.

A ***unit-time task*** is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a ***schedule*** for S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on.

The problem of ***scheduling unit-time tasks with deadlines and penalties for a single processor*** has the following inputs:

- a set $S = \{1, 2, \ldots, n\}$ of n unit-time tasks;
- a set of n integer ***deadlines*** $d_1, d_2, \ldots, d_n$, such that each d_i satisfies $1 \leq d_i \leq n$ and task i is supposed to finish by time d_i; and
- a set of n nonnegative weights or ***penalties*** $w_1, w_2, \ldots, w_n$, such that a penalty w_i is incurred if task i is not finished by time d_i and no penalty is incurred if a task finishes by its deadline.

We are asked to find a schedule for S that minimizes the total penalty incurred for missed deadlines.

Consider a given schedule. We say that a task is ***late*** in this schedule if it finishes after its deadline. Otherwise, the task is ***early*** in the schedule. An arbitrary schedule can always be put into ***early-first form***, in which the early tasks precede the late tasks. To see this, note that if some early task x follows some late task y, then we can switch the positions of x and y without affecting x being early or y being late.

We similarly claim that an arbitrary schedule can always be put into ***canonical form***, in which the early tasks precede the late tasks and the early tasks are scheduled in order of nondecreasing deadlines. To do so, we put the schedule into early-first form. Then, as long as there are two early tasks i and j finishing at respective times k and $k + 1$ in the schedule such that $dj < d_i$, we swap the positions of i and j. Since task j is early before the swap, $k + 1 \le d_j$. Therefore, $k + 1 < d_i$, and so task i is still early after the swap. Task j is moved earlier in the schedule, so it also still early after the swap.

The search for an optimal schedule thus reduces to finding a set A of tasks that are to be early in the optimal schedule. Once A is determined, we can create the actual schedule by listing the elements of A in order of nondecreasing deadline, then listing the late tasks (i.e., $S - A$) in any order, producing a canonical ordering of the optimal schedule.

We say that a set A of tasks is ***independent*** if there exists a schedule for these tasks such that no tasks are late. Clearly, the set of early tasks for a schedule forms an independent set of tasks. Let $\mathcal{I}$ denote the set of all independent sets of tasks.

Consider the problem of determining whether a given set A of tasks is independent. For $t = 1, 2, \ldots, n$, let $N_t(A)$ denote the number of tasks in A whose deadline is t or earlier.

Lemma 17.11

For any set of tasks A, the following statements are equivalent.

1. The set A is independent.

2. For $t = 1, 2, \ldots, n$, we have $N_t(A) \le t$.

3. If the tasks in A are scheduled in order of nondecreasing deadlines, then no task is late.

Proof Clearly, if $N_t(A) > t$ for some t, then there is no way to make a schedule with no late tasks for set A, because there are more than t tasks to finish before time t. Therefore, (1) implies (2). If (2) holds, then (3) must follow: there is no way to "get stuck" when scheduling the tasks in order of nondecreasing deadlines, since (2) implies that the ith largest deadline is at most i. Finally, (3) trivially implies (1).

Using property 2 of Lemma 17.11, we can easily compute whether or not a given set of tasks is independent (see Exercise 17.5-2).

The problem of minimizing the sum of the penalties of the late tasks is the same as the problem of maximizing the sum of the penalties of the early tasks. The following theorem thus ensures that we can use the greedy algorithm to find an independent set A of tasks with the maximum total penalty.

Theorem 17.12

If S is a set of unit-time tasks with deadlines, and $\mathcal{I}$ is the set of all independent sets of tasks, then the corresponding system $(S, \mathcal{I})$ is a matroid.

Proof Every subset of an independent set of tasks is certainly independent. To prove the exchange property, suppose that B and A are independent sets of tasks and that $|B| > |A|$. Let k be the largest t such that $N_t(B) \le N_t(A)$. Since $N_n(B) = |B|$ *and* $N_n(A) = |A|$, but $|B| > |A|$, we must have that $k < n$ and that $N_j(B) > N_j(A)$ for all j in the range $k + 1 \le j \le n$. *Therefore,* B *contains more tasks with deadline* k + *1 than* A *does. Let* x *be a task in* B - A *with deadline* k + *1. Let* $A' = A = A \cup \{x\}$.

We now show that A' must be independent by using property 2 of Lemma 17.11. For $1 \le t \le k$, we have $N_t(A') = N_t(A) \le t$,since A is independent. For $k < t \le n$, we have $N_t(A') \le N_t(B) \le t$, since B is independent. Therefore, A' is independent, completing our proof that $(S, \mathcal{I})$ is a matroid.

By Theorem 17.10, we can use a greedy algorithm to find a maximum- weight independent set of tasks A. We can then create an optimal schedule having the tasks in A as its early tasks. This method is an efficient algorithm for scheduling unit-time tasks with deadlines and penalties for a single processor. The running time is $O(n^2)$ using GREEDY, since each of the $O(n)$ independence checks made by that algorithm takes time $O(n)$ (see Exercise 17.5-2). A faster implementation is given in Problem 17-3.

	Task						
	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figure 17.7 An instance of the problem of scheduling unit-time tasks with deadlines and penalties for a single processor.

Figure 17.7 gives an example of a problem of scheduling unit-time tasks with deadlines and penalties for a single processor. In this example, the greedy algorithm selects tasks 1, 2, 3, and 4, then rejects tasks 5 and 6, and finally accepts task 7. The final optimal schedule is

$\langle 2, 4, 1, 3, 7, 5, 6 \rangle$,

which has a total penalty incurred of $w_5 + w_6 = 50$.

Exercises

17.5-1

Solve the instance of the scheduling problem given in Figure 17.7, but with each penalty w_i replaced by 80 - w_i.

17.5-2

Show how to use property 2 of Lemma 17.11 to determine in time $O(|A|)$ whether or not a given set A of tasks is independent.

Problems

17-1 Coin changing

Consider the problem of making change for n cents using the least number of coins.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in the denominations $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution.

17-2 Acyclic subgraphs

a. Let $G = (V, E)$ be an undirected graph. Using the definition of a matroid, show that $(E, \mathcal{I})$ is a matroid, where $A \in \mathcal{I}$ if and only if A is an acyclic subset of E.

b. The ***incidence matrix*** for an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix M such that $M_{ve} = 1$ if edge e is incident on vertex v, and $M_{ve} = 0$ otherwise. Argue that a set of columns of M is linearly independent if and only if the corresponding set of edges is acyclic. Then, use the result of Exercise 17.4-2 to provide an alternate proof that $(E, \mathcal{I})$ of part (a) is matroid.

c. Suppose that a nonnegative weight $w(e)$ is associated with each edge in an undirected graph $G = (V, E)$. Give an efficient algorithm to find an an acyclic subset of E of maximum total weight.

d. Let $G(V, E)$ be an arbitrary directed graph, and let $(E, \mathcal{I})$ be defined so that $A \in \mathcal{I}$ if and only if A does not contain any directed cycles. Give an example of a directed graph G such that the associated system $(E, \mathcal{I})$ is not a matroid. Specify which defining condition for a matroid fails to hold.

e. The ***incidence matrix*** for a directed graph $G = (V, E)$ is a $|V| \times |E|$ matrix M such that $M_{ve} = -1$ if edge e leaves vertex v, $M_{ve} = 1$ if edge e enters vertex v, and and $M_{ve} = 0$ otherwise. Argue that if a set of edges of G is linearly independent, then the corresponding set of edges does not contain a directed cycle.

f. Exercise 17.4-2 tells us that the set of linearly independent sets of columns of any matrix M forms a matroid. Explain carefully why the results of parts (d) and (e) are not contradictory. How can there fail to be a perfect correspondence between the notion of a set of edges being acyclic and the notion of the associated set of columns of the incidence matrix being linearly independent?

17-3 Scheduling variations

Consider the following algorithm for solving the problem in Section 17.5 of scheduling unit-time tasks with deadlines and penalties. Let all n time slots be initially empty, where time slot i is the unit-length slot of time that finishes at time i. We consider the jobs in order of monotonically decreasing penalty. When considering job j, if there exists a time slot at or before j's deadline d_j that is still empty, assign job j to the latest such slot, filling it. If there is no such slot, assign job j to the latest of the as yet unfilled slots.

a. Argue that this algorithm always gives an optional answer.

b. Use the fast disjoint-set forest presented in Section 22.3 to implement the algorithm efficiently. Assume that the set of input jobs has already been sorted into monotonically decreasing order by penalty. Analyze the running time of your implementation.

Chapter notes

Much more material on greedy algorithms and matroids can be found in Lawler [132] and Papadimitriou and Steiglitz [154].

The greedy algorithm first appeared in the combinatorial optimization literature in a 1971 article by Edmonds [62], though the theory of matroids dates back to a 1935 article by Whitney [200].

Our proof of the correctness of the greedy algorithm for the activity-selection problem follows that of Gavril [80]. The task-scheduling problem is studied in Lawler [132], Horowitz and Sahni [105], and Brassard and Bratley [33].

Huffman codes were invented in 1952 [107]; Lelewer and Hirschberg [136] surveys data-compression techniques known as of 1987.

An extension of matroid theory to greedoid theory was pioneered by Korte and Lovász [127, 128, 129, 130], who greatly generalize the theory presented here.

Go to Chapter 18 Back to Table of Contents