PART II

MAJOR DESIGN STRATEGIES

6

The Greedy Method

The *greedy method* for solving optimization problems follows the philosophy of greedily maximizing (minimizing) short-term gain and hoping for the best without regard to long-term consequences. Algorithms based on the greedy method are usually very simple, easy to code, and efficient. Unfortunately, just as in real life, in the theory of algorithms the greedy method applied to a problem often leads to less than optimal results. However, there are important problems where the greedy method does yield optimal results, such as computing optimal data compression codes, finding a shortest path between two vertices in a weighted digraph or determining a minimum spanning tree in a weighted graph (the latter two problems will be discussed in Chapter 12). Moreover, even when the greedy method does not yield optimal results, there are important problems in which it yields solutions that in some sense are good approximations to optimal.

6.1 General Description

Because the subject is greed, it seems appropriate to illustrate the greedy method with a problem about money: making change. Suppose that we have just purchased a sleeve of golf balls, and the salesperson wishes to give back (exact) change using the *fewest* number of coins. We assume that there is a sufficient amount of pennies (1¢), nickels (5¢), dimes (10¢), quarters (25¢), and half-dollars (50¢) to make change in any manner whatsoever.

A greedy algorithm for making change uses as many coins of the largest denomination as possible, then uses as many coins of the next largest denomination, and so forth. For example, if the change was 74 cents, the greedy solution uses one half-dollar, two dimes, and four pennies, for a total of seven coins. It is easy to see that seven coins is the fewest number of coins that makes 74 cents change. For any amount of change, this greedy method uses the fewest coins for coins of the above denominations. However, when other denominations of coins are used, this greedy method of making change does not always yield the fewest coins. For example, suppose we use the same denomination coins as before, except that we do not have any nickels. For 30 cents change, the greedy method yields one quarter and five pennies, whereas three dimes does the job.

The coin-changing example serves as a nice illustration of the general type of problem that might be amenable to the greedy method. These problems generally have the goal of maximizing or minimizing an associated *objective function* over all feasible solutions. In most cases this objective function is a real-valued function defined on the subsets of a given base set *S*. For example, coin changing is a minimization problem, where the base set *S* is the set of denominations, a feasible solution is a set of coins making correct change, and the objective function is the number of coins used.

The greedy method might be applied to any problem whose solution can be viewed as the result of building up solutions to the problem via a sequence of partial

solutions in the following manner. Partial solutions are built up by choosing elements $x_1, x_2, \ldots$ from a set R, where R is initially equal to some base set S. Whenever the greedy method makes a choice x_i, then x_i is either added to the current partial solution or its addition is found to be infeasible and is rejected forever. In either case, x_i is removed from the set R.

In the following general paradigm for the greedy method, choices are made by invoking a suitably defined function *GreedySelect*(R). For example, *GreedySelect*(R) might be an element that optimizes (minimizes or maximizes) the change in the value of the objective function or some closely related function. When there are several choices that are equivalent, we assume *GreedySelect* chooses arbitrarily from the equivalent choices.

```
procedure Greedy(S,Solution)
Input: S (base set)     //it is assumed that there is an associated objective
                        //function f defined on (possibly ordered) subsets of S
Output:  Solution       (an ordered subset of S that potentially optimizes the
                        objective function f, or a message that Greedy doesn't even
                        produce a solution (optimal or not))
    PartialSolution ← ∅    //initialize the partial solution to be empty
    R ← S
    while PartialSolution is not a solution .and. R ≠ ∅ do
        x ← GreedySelect(R)
        R ← R \ {x}
        if PartialSolution ∪ {x} is feasible then
            PartialSolution ← PartialSolution ∪ {x}
        endif
    endwhile
    if PartialSolution is a solution then
        Solution ← PartialSolution
    else
        write("Greedy fails to produce a solution")
    endif
end Greedy
```

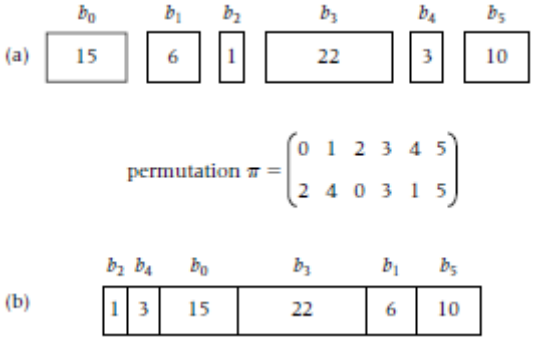
In most applications of the greedy method, an associated weighting w of the elements of the base set S is given, and *GreedySelect*(R) merely chooses the smallest weight or largest weight element in R. For convenience of discussion, assume that *GreedySelect* chooses the smallest weight element. The choice of a smallest element is facilitated by preconditioning the base set by sorting with respect to w or by maintaining the elements in the base set in a priority queue where w determines the priority value of the elements.

Even though the greedy method paradigm can be applied in a given situation, and even though it might output a feasible solution it still might not output an *optimal* solution (recall the problem of making 30 cents change without nickels). Thus, proofs of optimality should always accompany solutions generated by greedy algorithms. In spite of its somewhat limited applicability, the greedy method solves a number of classic

optimization problems. Moreover, when the greedy method does the job optimally, it generally leads to elegant and efficient code for solving the problem.

6.2 Optimal Sequential Storage Order

Suppose we are given a set of objects $b_0, b_1, \ldots b_{n-1}$ arranged sequentially in some order $b_{\pi(0)}, b_{\pi(1)},\ldots, b_{\pi(n-1)}$, where π is some permutation of $\{0,1,\ldots,n-1\}$, such as files stored on a sequential access tape. We assume that there is a weight c_i associated with object b_i (for example, the length of a file) that represents the individual cost of processing b_i. We also assume that to process object $b_{\pi(i)}$, we must first process the objects $b_{\pi(0)}, b_{\pi(1)},\ldots, b_{\pi(i-1)}$ that precede $b_{\pi(i)}$ in the sequential arrangement. For example, to read the i^{th} file on a sequential tape (assuming that the tape is rewound to the beginning), we must first read the preceding $i-1$ files on the tape. Thus the total cost of processing $b_{\pi(i)}$ is $c_{\pi(0)} + c_{\pi(1)} + \ldots + c_{\pi(i)}$. In Figure 6.1, we illustrate this example with six files $b_0,b_1,\ldots,b_5$ of lengths 15,6,1,22,3,10, respectively. A sample storage order of these files on a tape is given in Figure 6.1b. If we wish to read the file of length 22, we first have to read the files of length 1,3,15 that precede this file on the tape. Thus, reading the file of length 22 results in reading files whose total length is 1 + 3 + 15 + 22 = 41.



(a) Files of lengths 15, 6, 1, 22, 3, 10; (b) a sample storage order of files on a sequential access tap corresponding to the permutation π

Figure 6.1

The *optimal sequential storage* problem for processing objects $b_0,b_1,\ldots,b_{n-1}$ is to find an arrangement (permutation) of the objects $b_{\pi(0)},b_{\pi(1)},\ldots,b_{\pi(n-1)}$ that minimizes the average time it takes to process an object. Let p_i denote the probability that the object b_i is to be processed, $i = 0,\ldots,n-1$. Given these probabilities and a permutation π, clearly the average total cost of processing an object is given by

$$AveCost(\pi) = p_{\pi(0)}(c_{\pi(0)}) + p_{\pi(1)}(c_{\pi(0)} + c_{\pi(1)}) + \cdots + p_{\pi(n-1)}(c_{\pi(0)} + \cdots + c_{\pi(n-1)}). \quad (6.2.1)$$

We first consider the special case where each object is equally likely to be processed, so that $p_i = 1/n$, $i = 1,\ldots, n$. Then (6.2.1) becomes

$$\begin{aligned} AveCost(\pi) &= \frac{1}{n}[(c_{\pi(0)}) + (c_{\pi(0)} + c_{\pi(1)}) + \cdots + (c_{\pi(0)} + c_{\pi(1)} + \cdots + c_{\pi(n-1)})] \\ &= \frac{1}{n}\sum_{i=0}^{n-1} (n-i)c_{\pi(i)}. \end{aligned} \tag{6.2.2}$$

Looking at (6.2.2), we see that the coefficients of $c_{\pi(i)}$, $i = 0,...,n-1$, decrease as i increases. Thus, it is intuitively obvious that $AveCost(\pi)$ is minimized when $c_{\pi(0)} \le c_{\pi(1)} \le ... \le c_{\pi(n-1)}$, so that the greedy strategy we use is to select the objects in increasing order of their costs.

Now consider the problem of minimizing $AveCost(\pi)$ as given by (6.2.1) for a general set of probabilities p_i, $i = 1,...,n$. Various greedy strategies come to mind. For example, we might use the same greedy strategy given previously, which give the optimal solution when all the probabilities are equal. However, it is not hard to find examples where this strategy yields suboptimal results. We might also arrange the objects in decreasing order of probabilities (placing the objects most likely to be processed near the beginning), but, again, it is not hard to find examples where this strategy fails. Alas, is there a greedy strategy that works? Yes, we leave it as an exercise to show that arranging the objects in increasing order of the ratios c_i/p_i yields an optimal solution.

6.3 The Knapsack Problem

The *Knapsack problem* involves n objects $b_0,b_1,...,b_{n-1}$ and a knapsack of capacity C. We assume that each object b_i has a given positive weight (or volume) w_i, and a given positive value v_i (or profit, cost, and so forth) $i = 0,..., n-1$. We place the objects or fractions of objects in the knapsack, taking care that the capacity of the knapsack is not exceeded. If a fraction f_i of object b_i is placed in the knapsack ($0 \le f_i \le 1$), then it contributes $f_i v_i$ to the total value in the knapsack. The Knapsack problem is to place objects or fractions of objects in the knapsack, without exceeding the capacity, so that total value of the objects in the knapsack is maximized. A more formal statement of the Knapsack problem is

$$\begin{aligned} &\text{maximize} \quad \sum_{i=0}^{n-1} f_i v_i, \\ &\text{subject to the constraints:} \quad \sum_{i=0}^{n-1} f_i w_i \le C, \\ &\qquad 0 \le f_i \le 1,\ i = 0, \ldots, n-1. \end{aligned} \tag{6.3.1}$$

As with the sequential storage order problem, several possible greedy methods for solving the Knapsack problem come to mind. For example, we might put as much as possible of the object with highest value into the knapsack, then as much as possible of the object of second-highest value into the knapsack, and so forth, until the knapsack is filled to capacity. Another greedy strategy would be to follow this same scheme, but where the objects are placed in the knapsack according to decreasing weights. But we can easily find examples where neither of these strategies yields optimal solutions. We will show that the strategy of putting the objects into the knapsack according to decreasing *ratios* (*densities*) v_i/w_i until the knapsack is full yields an optimal solution.

The following interpretation of the Knapsack problem makes it clear that the ratio v_i/w_i is the critical issue. Suppose the (greedy) owner of a gourmet coffee shop wishes to mix a 10-pound bag of coffee using various types of coffee beans in such a way as to produce the coffee blend of maximum cost. .The weights of the objects in the Knapsack problem correspond to the quantity in pounds that are available of each type of coffee bean. The value of each quantity of coffee bean is the total cost of that quantity in dollars. In Figure 6.2 we list the types and amounts of coffee beans that are available.

Type	Quantity Available In Pounds	Total Cost In Dollars for That Quantity
Colombian	8	32
Jamaican	2	32
Java	4	25
Bicentennial	1	10
Mountain	2	9
Roast	6	18
Dark	10	55
Special	50	100

Coffee shop inventory

Figure 6.2

Intuitively, we obtain the most expensive 10-pound bag of coffee if we first use as much as possible of the bean whose cost/pound ratio is the largest, then as much as possible of the bean whose cost/pound ratio is the second-largest, and so on until we reach 10 pounds. Figure 6.3 lists the coffee beans in decreasing order of their cost/pound ratios and the quantity of each type of bean used as determined by this greedy method.

Type	Cost/Pound	Fraction of Available Quantity Chosen	Quantity Chosen
Jamaican	16	1	2
Bicentennial	10	1	1
Java	6.25	1	4
Dark	5.5	0.3	3
Mountain	4.5	0	0
Colombian	4	0	0
Roast	3	0	0
Special	2	0	0

Greedy solution for 10-pound bag of coffee

Figure 6.3

Generalizing the greedy strategy for making expensive coffee leads to a greedy algorithm for the Knapsack problem. The algorithm *Knapsack* assumes as a precondition that the objects have been sorted in decreasing order of the ratios v_i/w_i, $i = 0,...,n-1$.

Procedure *Knapsack*($V[0:n-1]$, $W[0:n-1]$, C, $F[0:n-1]$)
Input: $V[0:n-1]$ (an array of positive values)
$W[0:n-1]$ (an array of positive weights)
C (a positive capacity)

```
Output:   F[0:n – 1] (array of nonnegative fractions)
  Sort the arrays V[0:n – 1] and W[0:n – 1] in decreasing order of densities, so that
       V[0]/W[0] ≥ V[1]/W[1] ≥ ... ≥ V[n – 1]/W[n – 1]
   for i ← 0 to n – 1do
       F[i] ← 0
    endfor
    RemainCap = C
    i ← 0
    if W[0] ≤ C then
       Fits ← .true.
    else
       Fits ← .false.
    endif
    while Fits .and. i ≤ n – 1 do
       F[i] ← 1
       RemainCap ← RemainCap – W[i]
       i ← i + 1
       if W[i] ≤ RemainCap then
          Fits ← .true.
       else
          Fits ← .false.
       endif
    endwhile
    if i ≤ n – 1 then F[i] ← RemainCap/W[i] endif
end Knapsack
```

Clearly, procedure *Knapsack* generates an optimal solution when $\sum_{i=0}^{n-1} w_i \le C$, since all the objects are then placed in the knapsack. Using a proof by contradiction, we now show that *Knapsack* generates an optimal solution when $\sum_{i=0}^{n-1} w_i > C$. Suppose to the contrary that the solution $F = (f_0,f_1,..., f_{n-1})$ generated by *Knapsack* is suboptimal. Now consider an optimal solution $Y = (y_0,y_1,...,y_{n-1})$, and let j denote the first index i such that $f_i \neq y_i$ so that $1 = f_i = y_i$, $0 \le i \le j - 1$. We choose Y such that j is maximized over all optimal solutions. Note that $\sum_{i=0}^{n-1} y_i w_i = C$ (otherwise we could increase one of the y_i's and trivially increase the total value of the solution). Because of the greedy strategy used by *Knapsack*, we have $f_j > y_j$.

By altering Y, we now construct an optimal solution $Z = (z_1,z_2,...,z_n)$ that agrees with F in one more initial position. We set $z_i = f_i$, $1 \le i \le j$, and for $i > j$ the value of z_i is obtained by suitably decreasing y_i so that $\sum_{i=0}^{n-1} z_i w_i = C$. It follows that:

$$(z_j - y_j)w_j = \sum_{i=j+1}^{n} (y_i - z_i)w_i. \qquad (6.3.2)$$

Comparing the total values of the solutions Z and Y, we obtain

$$\sum_{i=0}^{n-1} z_i v_i - \sum_{i=0}^{n-1} y_i v_i = (z_j - y_j)v_j - \sum_{i=j+1}^{n-1} (y_i - z_i)v_i.$$

$$= (z_j - y_j)w_j v_j / w_j - \sum_{i=j+1}^{n-1} (y_i - z_i)w_i v_i / w_i$$

$$\geq \left[(z_j - y_j)w_j - \sum_{i=j+1}^{n-1} (y_i - z_i)w_i \right] v_j / w_j \quad \text{(since } y_i \geq z_i$$

$$\text{and } v_0 / w_0 \geq v_1 / w_1 \geq \cdots \geq v_{n-1} / w_{n-1})$$

$$= 0 \quad \text{(since the number inside the square brackets is 0 by (7.3.2)).}$$

Thus, the value of Z is not less than the value of Y, so that Z is also an optimal solution. But Z agrees with F in the first j initial positions, which contradicts the assumption that Y is an optimal solution that agrees with F in the most initial positions.

Except for the $\Theta(n \log n)$ sorting step, the remainder of *Knapsack* has linear complexity in the worst case.

A natural variation of the Knapsack problem, called the *0/1 Knapsack problem*, is the same as the Knapsack problem except that fractional objects are not allowed. More precisely, the 0/1 Knapsack problem has the following formulation:

$$\text{maximize } \sum_{i=0}^{n-1} f_i v_i$$

$$\text{subject to the constraints: } \sum_{i=0}^{n-1} f_i w_i \leq C \qquad (6.3.3)$$

$$f_i \in \{0,1\}, \quad i = 0, \ldots, n-1.$$

The greedy strategy of placing the objects in the knapsack in decreasing order of the ratios v_i/w_i, $i = 0,...,n-1$, rejecting objects when their addition would overflow the knapsack, may yield a suboptimal solution to the 0/1 Knapsack problem. While the Knapsack problem is easily solved, the 0/1 Knapsack problem is NP-hard. Note that we took the input size for the Knapsack problems to be the number of objects n. This measure does not take into account the sizes of the weights and values of the objects. Assuming the capacity, weights and values are positive integers, a more precise measure of the input size (in binary) would be the sum of the number of digits of each of these integers; that is

$$b = \lceil \log_2 C \rceil + \sum_{i=0}^{n-1} \lceil \log_2 (w_i + 1) \rceil + \sum_{i=0}^{n-1} \lceil \log_2 (v_i + 1) \rceil.$$

We chose n as the input size because it is simpler to analyze the algorithm in terms of n. Also, since $b \geq n$, any upper bound formula for the complexity of the algorithm in terms of n applies when n is replaced with b. Sometimes it is useful to measure the input size in unary (base 1); for example, assuming the capacity, weights and values are positive integers, the input size to the Knapsack problems is measured in terms of $C + \sum_{i=0}^{n-1} w_i + \sum_{i=0}^{n-1} v_i$. For many NP-complete problems, such as the coin changing and 0/1 Knapsack problem, the problem as measured by unary inputs has polynomial

complexity. In exercises 8.x and 8.y of Chapter 8 polynomial-time dynamic programming solutions to coin changing and 0/1 Knapsack problem are developed when the input numbers are measured as unary integers.

6.4 Huffman Codes

With the emergence of the Internet, the need for data compression has become increasingly important. Transporting extremely large files across the internet is commonplace today, and these files must be compressed in order to save time and storage resources. For example, there are standard data compression algorithms such as jpeg or gif that are used to compress image files (such as photographs). There are also standard data compression algorithms for text and program files, such as those used to zip and unzip files. Even before the advent of the Internet, the need for compressing large text files was necessary in order to minimize storage requirements. A classical data compression algorithm, due to Huffman in a 1952 paper, is based on a greedy strategy for constructing a code for a given alphabet of symbols. Not only are Huffman codes useful for compressing text files, they also are used as part of image or audio file compression techniques.

For a given *alphabet* of symbols $A = \{a_0,a_1,...,a_{n-1}\}$, one approach to the problem of data compression involves encoding each symbol in the alphabet with a binary string. The alphabet *A* might be letters and punctuation symbols from the English language, the standard symbol set used by personal computers, Discrete Cosine Transform (*DCT*) coefficients in a jpeg compression of an image file, and so forth. We refer to the set of binary strings encoding the symbols in *A* as a *binary code* for *A*. Consider, for example, the *ASCII* encoding of the alphabet *A* of standard symbols used by computers consisting of alphabetical characters, numeric characters, punctuation characters, control characters, and so forth. Each symbol of this alphabet is represented by an 8-bit (1-byte) binary string. In this example, since each symbol is encoded with a fixed length binary string, there is basically no compression being used. Using a fixed length binary string to store symbols is very convenient for computer implementation, but has the disadvantage that it does not optimize storage usage. By allowing variable length binary strings for binary codes, we can save space when storing files (text, jpeg, mpeg, etc.) made up of symbols from our given alphabet.

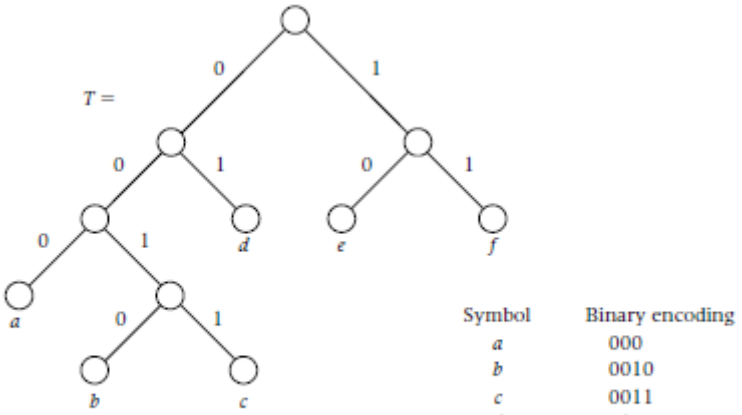
Given a binary code for *A*, we obtain a binary encoding of any string consisting of symbols from *A*. We simply replace each symbol in the string with its binary encoding. For example, suppose $A = \{a, b, c\}$ and symbols *a*, *b*, *c* are encoded as 1, 00, 01, respectively. Then the binary encoding for the string *cabbc* is 011000001.

To illustrate ambiguous codes, suppose *a*,*b*,*c* are encoded as 0, 1, 01, respectively. Then the binary string 011 can be decoded as either *abb* or *cb*. Note that the binary string 0 is a prefix for the binary string 01. It is easily verified that a binary code allows for unambiguous (unique) decoding if, and only if, no binary string in the code is a prefix of any other binary string in the code.

6.4.1 Binary Prefix Codes

Binary codes having the property that no binary string in the code is a prefix of any other binary string in the code are called *prefix codes*. We can readily create a prefix

code by utilizing a 2-tree T, which is a binary tree where every non-leaf node has exactly two children. The leaf nodes of T correspond to the n symbols in A. A high-level description of the process is as follows. Label each edge corresponding to a left child with a zero and each edge corresponding to a right child with a one. Assign to each symbol a_i the binary string obtained by reading the labels on the edges when following a path from the root to the leaf node corresponding to a_i. The binary strings generated by a sample tree for the alphabet $A = \{a,b,c,d,e,f\}$ are shown in Figure 6.4.



Symbol	Binary encoding
a	000
b	0010
c	0011
d	01
e	10
f	11

Encoding of alphabet $A = \{a,b,c,d,e,f\}$ generated by a sample 2-tree T

Figure 6.4

For each node N of the 2-tree T, let $B(N)$ denote the binary string corresponding to following a path from the root to N in the manner described earlier. Let L and R denote the left and right children of N, respectively. Note that $B(L)$ and $B(R)$ can be computed from $B(N)$ by concatenating '0' and '1', respectively, to the end of $B(N)$; that is, $B(L) = B(N)$ + '0' and $B(R) = B(N)$ + '1' (where + denotes the operation of concatenation). We now give pseudocode for the algorithm *GenerateCode*, which utilizes this formula to compute the array *Code*, where *Code*[i] is the binary code for symbol a_i, $i = 1,...,n$. *GenerateCode* computes $B(N)$ for each node by performing a preorder traversal of the 2-tree, where the generalized visit operation at node N involves computing $B(L)$ and $B(R)$ from $B(N)$ using the formula described above. We assume that the 2-tree T is implemented using pointers and dynamic variables, where each node contains two pointer fields corresponding to the left and right children and two information fields *BinaryCode* and *SymbolIndex*. Before calling *GenerateCode*, we assume that the value of *SymbolIndex* for each leaf node has been initialized to contain the index i of the alphabet symbol a_i corresponding to the leaf node (*SymbolIndex* is left undefined for the internal nodes of T). The computed value $B(N)$ for each node N is stored in the variable *BinaryCode*. The root node's value of *BinaryCode* is initialized to the null string before calling *GenerateCode*.

```
procedure GenerateCode(Root,Code[0:n – 1]) recursive
Input: Root (a pointer to root of 2-tree T)        //Root→BinaryCode is //initialized to the
                null string. The leaf node corresponding to a_i //has its SymbolIndex field
                initialized to i, i = 0,...,n – 1
Output:  Code[0:n – 1] (array of binary strings, where Code[i] is the code for symbol
         a_i)
    if Root→LeftChild = null then        //a leaf is reached
        Code[Root→SymbolIndex] ← Root→BinaryCode
    else
        (Root→LeftChild)→BinaryCode ← Root→BinaryCode + '0'
        (Root→RightChild)→BinaryCode ← Root→BinaryCode + '1'
        GenerateCode(Root→LeftChild,Code)
        GenerateCode(Root→RightChild,Code)
    endif
end GenerateCode
```

The table *Code*[0:n – 1] of binary codes can be used to encode text consisting of strings of symbols from the alphabet A. Going the other direction, encoded text can be decoded using the 2-tree T in the obvious way.

6.4.2 The Huffman Algorithm

Now suppose we are encoding text composed of symbols from A, where the frequency of the occurrence of symbol a_i in such text is given by f_i, $i = 0,...,n-1$. An important problem in data compression is to find an *optimal* binary prefix code, that is, a binary prefix code that minimizes the expected length of a *coded symbol*. Clearly, an optimal binary prefix code also minimizes the expected length of text generated using the symbols from A. The problem of finding optimal binary prefix codes is equivalent to the following problem for 2-trees. Each unambiguous binary code can be generated by traversing a suitable 2-tree T. We let λ_i denote the length of the path in T from the root to (the leaf node corresponding to) a_i, $i = 0,...,n-1$. The expected length of the coded symbols determined by T is closely related to a generalization of the leaf path length called the *weighted leaf path length WLPL*(T) *of T*, defined by

$$WLPL(T) = \sum_{i=0}^{n-1} \lambda_i f_i. \tag{6.4.1}$$

Note that $WLPL(T)$ becomes $LPL(T)$ when all the f_i's equal 1. For example, given frequencies 9,8,5,3,15,2 for the symbols a,b,c,d,e,f, respectively, the weighted leaf path length for the binary tree T in Figure 6.4 is given by

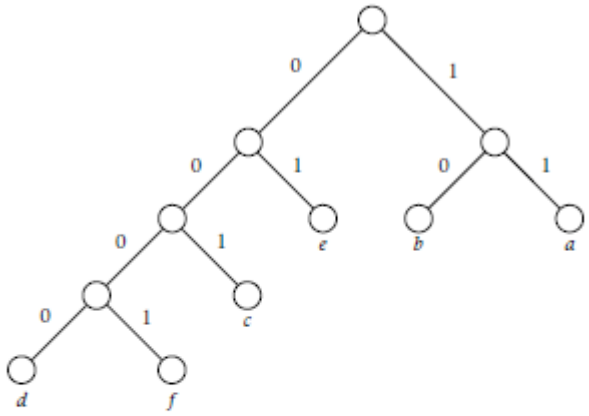
$$WLPL(T) = (3)(9) + (4)(8) + (4)(5) + (2)(3) + (2)(15) + (2)(2) = 119. \tag{6.4.2}$$

Since symbol a_i occurs with frequency f_i the probability p_i, $i = 0,...,n-1$ that symbol a_i occurs in a given text position is

$$p_i = \frac{f_i}{\sum_{j=0}^{n-1} f_j}, \quad i = 0,\ldots,n-1.$$

Each binary digit in the binary string encoding a_i corresponds to an edge of the path in T from the root node to the leaf node containing a_i. Hence, the length of the binary string encoding a_i (in the binary code determined by T) equals λ_i, $i = 0,...,n-1$, so that the expected length of a coded symbol is given by $WLPL(T)/(\sum_{i=0}^{n-1} f_i)$. Thus, the problem of finding an optimal binary prefix code is equivalent to finding a 2-tree having minimum $WLPL(T)$.

Given the set of frequencies 9,8,5,3,15,2 for the symbols a,b,c,d,e,f, respectively, the 2-tree in Figure 6.4, which has weighted leaf path length equal to 119, is not optimal. The binary tree given in Figure 6.5, which has weighted leaf path length equal to 99, turns out to be an optimal 2-tree with respect to these frequencies.



Symbol	Frequency	Encoding
a	9	11
b	8	10
c	5	001
d	3	0000
e	15	01
f	2	0001

$WLPL(T) = (2)(9)+(2)(8)+(3)(5)+(4)(3)+(2)(15)+(4)(2) = 99$

A tree having minimum weighted leaf path length for alphabet $A = \{a,b,c,d,e,f\}$ with respected to given frequencies

Figure 6.5

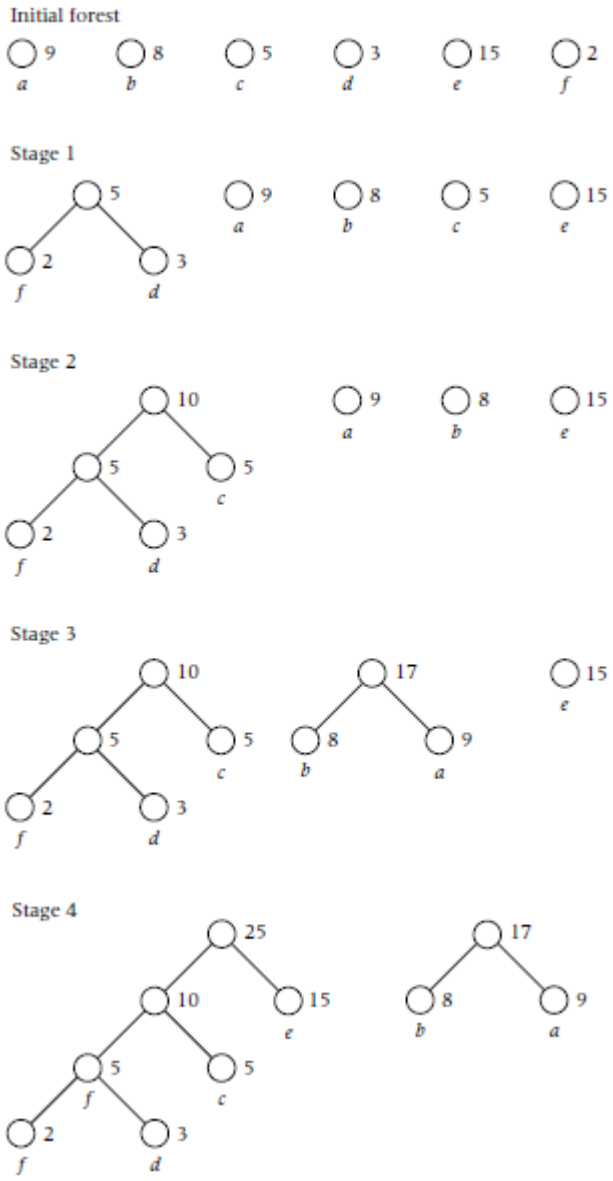
We now describe Huffman's greedy algorithm for computing an optimal binary prefix code with respect to a given set of frequencies f_i, $i = 0,...,n-1$. The procedure *HuffmanCode* computes a 2-tree T minimizing $WLPL(T)$ by constructing a sequence of forests as follows. At each stage in the algorithm, the root of each tree in the forest is assigned a frequency. The initial forest F consists of the n single node trees corresponding to the n elements of A. At each stage *HuffmanCode* finds two trees T_1 and T_2 whose roots R_1 and R_2 have the smallest and second-smallest frequencies over all the trees in the current forest. A new (internal) node R is then added to the forest, together with two edges joining R to R_1 and R to R_2, so that a new tree is created with R as the root and T_1 and T_2 as the left and right subtrees of R. The frequency of the new root vertex R

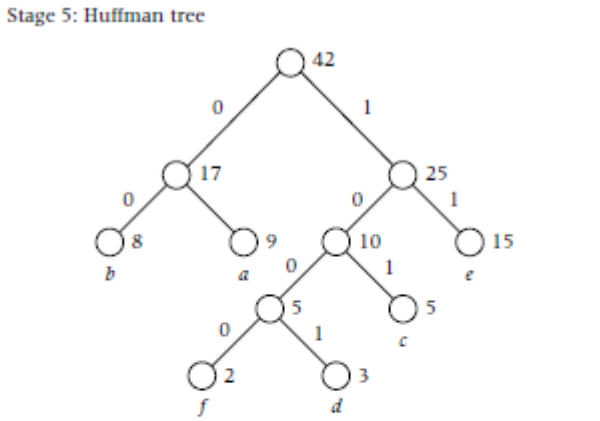
is taken to be the sum of the frequencies of the old root vertices R_1 and R_2. The Huffman tree is constructed after $n - 1$ stages, involving the addition of $n - 1$ internal nodes.

The procedure *HuffmanCode* calls the procedure *GenerateCode* described earlier. We utilize high-level parameter descriptions in our calls to procedures implementing the various priority queue operations. For example, a call to *CreatePriorityQueue* with parameters *Leaf*[0:$n - 1$] and *Q* creates a priority queue *Q* of pointers to the leaf nodes. A call to *RemovePriorityQueue* with parameters *Q* and *L* removes the element at the front of the queue *Q* and assigns it to *L*.

```
Procedure HuffmanCode(A[0:n – 1],Freq[0:n – 1],Code[0:n – 1])
Input: Freq[0:n – 1]     (an array of nonnegative frequencies: Freq[i] = f_i)
Output:    Code[0:n – 1]  (an array of binary strings for Huffman code: Code[i] is the
                                binary string encoding symbol a_i, i = 0,...,n – 1)
    for i ← 0 to n – 1 do       //initialize leaf nodes
        AllocateHuffmanNode(P)
        P → SymbolIndex ← i
        P → Frequency ← Freq[i]
        P → LeftChild ← null
        P → RightChild ← null
        Leaf[i] ← P
    endfor
    CreatePriorityQueue(Leaf[0:n – 1],Q)     //create priority queue of
                                //pointers to leaf nodes with Frequency as the key
    for i ← 1 to n – 1 do
        RemovePriorityQueue(Q,L)   //L,R point to root nodes of //smallest and
        RemovePriorityQueue(Q,R)   //second-smallest frequency, //respectively
        AllocateHuffmanNode(Root)
        Root → LeftChild ← L
        Root → RightChild ← R
        Root → Frequency ← (L → Frequency) + (R → Frequency)
        InsertPriorityQueue(Q,Root)
    endfor
    Root → BinaryString ← ''    //BinaryString of root initialized to null //string
    GenerateCode(Root,Code[0:n – 1])
end HuffmanCode
```

The action of *HuffmanCode* is illustrated in Figure 6.6 for the same sample alphabet and set of frequencies as in Figure 6.5.


Initial forest
9
a
8
b
5
c
3
d
15
e
2
f
Stage 1
5
2
f
3
d
9
a
8
b
5
c
15
e
Stage 2
10
5
5
c
2
f
3
d
9
a
8
b
15
e
Stage 3
10
5
5
c
2
f
3
d
17
8
b
9
a
15
e
Stage 4
25
10
15
e
5
f
5
c
2
f
3
d
17
8
b
9
a



Symbol	Frequency	Encoding
a	9	01
b	8	00
c	5	101
d	3	1001
e	15	11
f	2	1000

$WLPL(T) = (2)(9) + (2)(8) + (3)(5) + (4)(3) + (2)(15) + (4)(2) = 99$

Action of HuffmanCode for the alphabet $A = \{a,b,c,d,e,f\}$ with the same set of frequencies as in Figure 6.5

Figure 6.6

An efficient way to implement the priority queue *Q* is to use a min-heap (see Chapter 4). Then *CreatePriorityQueue* (*CreateMinHeap*), *RemovePriorityQueue* (*RemoveMinHeap*), and *InsertPriorityQueue* (*InsertMinHeap*) have worst-case complexities belonging to $O(n)$, $O(\log n)$, and $O(\log n)$, respectively. Hence, the worst-case complexity of *HuffmanCode* belongs to $O(n \log n)$.

The correctness of the algorithm *HuffmanCode* depends on the following proposition, whose proof is left to the exercises.

Proposition 6.4.1

Given a set of frequencies f_i, $i = 0,...,n-1$, the Huffman tree *T* constructed by *HuffmanCode* has minimal weighted $WLPL(T)$ over all 2-trees whose edges are weighted by these frequencies. □

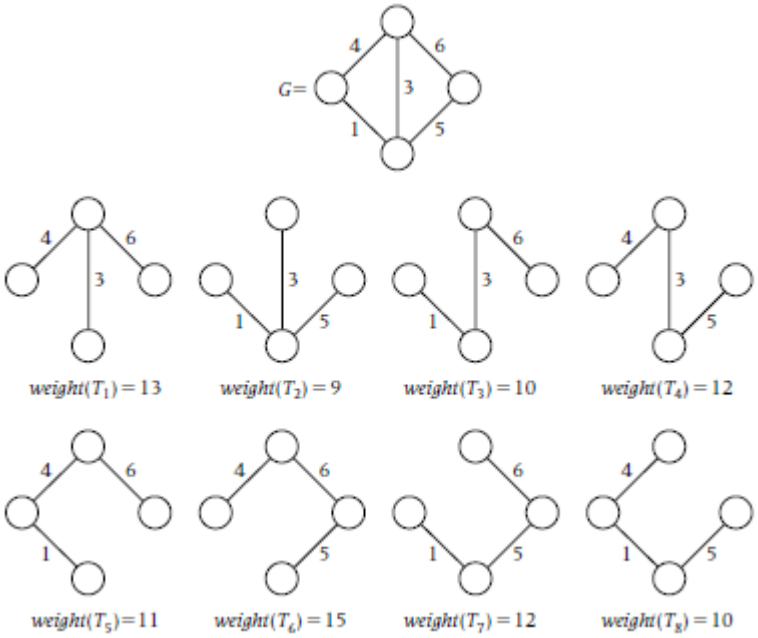
Finding a binary tree of minimum weighted leaf path length has other important applications besides Huffman codes. For example, the problem of finding optimal merge patterns is solved by constructing a binary tree having minimum weighted leaf path length, where the leaf nodes of the tree correspond to files or sublists and the weight of a leaf is the length of its associated file or sublist.

6.5 Minimum Spanning Tree

The problem of finding a minimum spanning tree in a weighted graph is particularly important in network applications. For example, the problem arises when designing any physical network connecting n nodes, where the connections between nodes are subject to feasibility and weight constraints. Examples of such physical networks include communication networks, transportation networks, energy pipelines, VLSI chips, and so forth. In all these examples, the weight of a minimum spanning tree provides a lower bound on the cost of building the network.

Given a spanning tree T in a graph G with a weighting w of the edges E of G, the *weight* of T, denoted *weight*(T), is the sum of the w-weights of its edges. If T has minimum weight over all spanning trees of G, then we call T a *minimum spanning tree*.

In Figure 6.7, all the spanning trees of a weighted graph on four vertices are enumerated. The minimum spanning tree is then obtained by inspection.



Enumeration of the spanning trees of G. Minimum spanning tree is T_2.

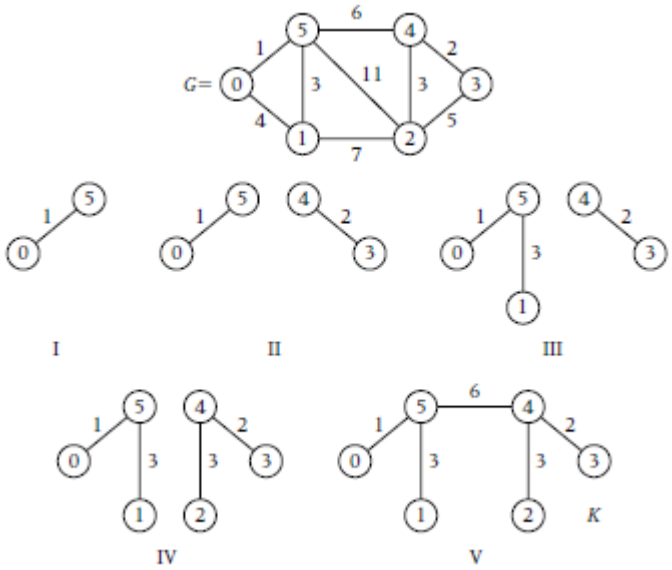
Figure 6.7

The number of spanning trees, even for relatively small graphs, is usually enormous, making an enumerative brute-force search infeasible. Indeed, Cayley proved that the complete graph K_n on n vertices contains n^{n-2} spanning trees!

Fortunately, efficient algorithms based on the greedy method do exist for finding minimum spanning trees. We discuss two of the more famous ones, namely, Kruskal's and Prim's algorithms. These algorithms are similar in the sense that their selection functions always choose an edge of minimum weight among the remaining edges. However, Prim's algorithm selection function only considers edges incident to vertices already included in the tree. In particular, the partial solutions built by Prim's algorithm are trees, whereas the partial solutions built by Kruskal's algorithm are forests.

6.5.1 Kruskal's Algorithm

Given a connected graph $G = (V,E)$, with $|V| = n$ and $|E| = m$, and a weight function w defined on the edge set E, Kruskal's algorithm finds a minimum spanning tree T of G by constructing a sequence of n forests
$F_0,F_1,...,F_{n-1}$, where F_0 is the empty forest and F_i is obtained from F_{i-1} by adding a single edge e. The edge e is chosen so that it has minimum weight among all the edges not belonging to F_{i-1} and doesn't form a cycle when added to F_{i-1}. Kruskal's algorithm is illustrated for a sample graph in Figure 6.8.



Stages in Kruskal's Algorithm

Figure 6.8

Remark

For convenience, we assume that the input graph G to Kruskal's algorithm is connected. However, Kruskal's algorithm is easily modified to accept *any* graph G (connected or disconnected) and to output a minimum (weight) forest, each of whose trees spans a (connected) component of G.

Using a proof by contradiction, we now show that the spanning tree K generated by Kruskal's algorithm is a minimum spanning tree. For convenience, we will assume that the edge weights are distinct (the proof can be slightly modified to hold for nondistinct edge weights, see Exercise 12.4). Let the edges of K be denoted by $x_1,x_2,...,x_{n-1}$, listed in increasing order of their weights. Assume that K is not a minimum spanning tree, and let T be a minimum spanning tree with edges $y_1,y_2,...,y_{n-1}$, listed in increasing order of their weights. Let j denote the first index i such that $x_i \neq y_i$, so that $x_i = y_i$, $1 \leq i \leq j - 1$. Since x_j was chosen by the greedy strategy, we have $w(x_j) < w(y_j)$.

We now construct a spanning tree T' whose weight is smaller than T. Consider the subgraph H obtained by adding the edge x_j to T. It is easily verified that H contains a unique cycle C and that this cycle contains x_j. Since $y_i = x_i$, $i = 1,2,...,j - 1$, and the tree K does not contain C, it follows that C must contain the edge y_k for some index $k \geq j$. Let T'

be the spanning tree obtained from H by deleting the edge y_k. Since $w(y_k) \geq w(y_j) > w(x_j)$, we have

$$weight(T') = weight(T) + w(x_j) - w(y_k) < weight(T),$$

contradicting our assumption that T is a minimum spanning tree.

The key implementation detail in Kruskal's algorithm is the detection of whether the addition of a given edge e to the existing forest F creates a cycle. Observe that a cycle is formed by the edge $e = uv$ if, and only if, u and v belong to the same component (tree) in F. Checking whether u and v belong to the same component can be done efficiently using the union and find algorithms for disjoint sets discussed in Chapter 4, where the collection of disjoint sets in the current context is the collection of vertex sets of each tree in forest F.

Key Fact
When implementing Kruskal's algorithm, note that an edge uv can be added to the current forest if, and only if, u and v belong to different trees in the forest. To efficiently test this condition, we use the disjoint set ADT, where the disjoint sets correspond to vertex sets of the trees in the forest, and the forest is maintained using the procedures *Union* and *Find2* discussed in Chapter 4.

In the following pseudocode for procedure *Kruskal*, the forest F is maintained using its set of edges.

```
procedure Kruskal(G,w,MCSTree)
Input: G   (a connected graph with vertex set V of cardinality n
                and edge set E = {e_i = u_i v_i | i ∈ {1,...,m}})
           w   (a weight function on E)
Output:    MSTree (a minimum spanning tree)
    Sort the edges in nondecreasing order of weights w(e_1) ≤ ... ≤ w(e_m)
    Forest ← ∅
    Size ← 0
    for i ← 0 to n – 1 do  //initialize a disjoint collection of single vertices
        Parent[i] ← – 1
    endfor
    j ← 0
    while Size ≤ n – 1 .and. j < m do
        j ← j + 1
        Find2(Parent[0:n – 1],u_j,r)        //e_j = u_j v_j
        Find2(Parent[0:n – 1],v_j,s)
        if r ≠ s then  //add edge e_j to Forest
            Forest ← Forest ∪{e_j}
            Size ← Size + 1
            Union(Parent[0:n – 1],r,s) //combine sets containing u_j and v_j
        endif
    endwhile
    MSTree ← Forest
end Kruskal
```

Since the procedure *Kruskal* does $n - 1$ unions and at most m finds, it follows for the results given in Chapter 4 that checking for cycles has a worst-case complexity that is almost linear in m. In procedure *Kruskal*, we assumed as a precondition that the edges were sorted in increasing order of their weights. Since presorting the edges can be done in time $O(m\log m) = O(m\log n)$, the overall order of the worst-case complexity is $\Theta(m\log n)$.

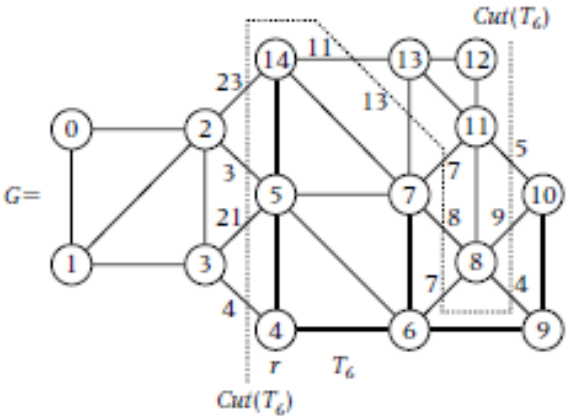
Instead of sorting the edges, we could more efficiently select the next minimum weight edge by utilizing a priority queue of edges of the graph, implemented, for example, with a min-heap. *CreatePriorityQueue* (*MakeMinHeap*) and *RemovePriorityQueue* (*RemoveMinHeap*) then have worst-case complexities belonging to $O(m)$ and $O(\log m) = O(\log n)$, respectively. Hence, the selection process remains $O(m\log n)$ in the worst case. However, improvement over procedure *Kruskal* occurs for inputs where the tree is completed early (that is, not many of the selected edges form cycles). For example, in the best case, the first $n - 1$ edges selected form a tree, yielding best-case complexity belonging to $O(m + n\log n)$.

6.5.2 Prim's Algorithm

Prim's algorithm for generating a minimum spanning tree differs from Kruskal's algorithm in that it maintains a tree at each stage instead of a forest. The initial tree T_0 may be taken to be any single vertex r of the graph G. Prim's algorithm builds a sequence of n trees (rooted at r) $T_0, T_1, \ldots, T_{n-1}$, where T_{i+1} is obtained from T_i by adding a single edge e_{i+1}, $i = 0,\ldots,n-2$. The edge e_{i+1} is selected (greedily) to be a minimum weight edge among all edges having *exactly* one vertex in T_i. The set of all edges in G having exactly one vertex in T_i is denoted by $Cut(T_i)$. With this notation, at the i^{th} step Prim's algorithm chooses an edge e_i of minimum weight among all the edges in $Cut(T_i)$ (see Figure 6.9). That we can restrict attention to edges in $Cut(T_i)$ is a consequence of the following key fact.

Key Fact

Given any tree T and edge e in a graph G, $T \cup e$ is a tree if, and only if, $e \in Cut(T)$.

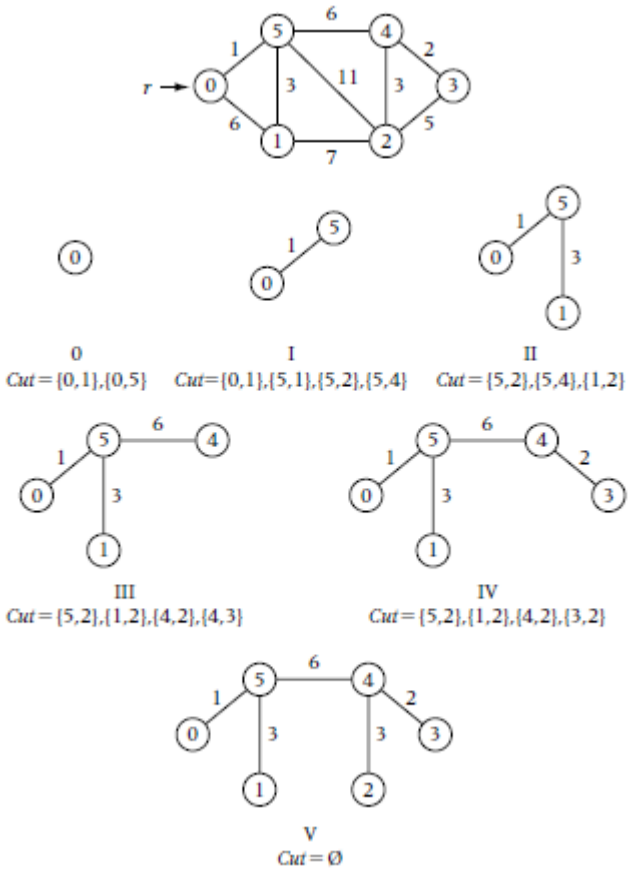


Sample tree T_6 of Prim's algorithm with $V(T_6) = \{4,5,6,7,9,10,14\}$ and $r = 4$.
$Cut(T) = \{\{3,4\},\{3,5\},\{2,5\},\{2,14\},\{13,14\},\{7,13\},\{7,11\},$
$\{7,8\},\{6,8\},\{8,9\},\{8,10\},\{10,11\}\}$.
At stage 6 Prim's algorithm adds edge $e_6 = \{2,5\}$ to T_6,
since $\{2,5\}$ has minimum weight 3 in $Cut(T_6)$.

Sample tree T_6 of Prim's algorithms

Figure 6.9

A proof that Prim's algorithm yields the minimum spanning tree is left to the exercises. Prim's algorithm is illustrated in Figure 6.10 using the same sample graph as in Figure 6.8.



Tree T and $Cut(T)$ at each stage in Prim's algorithm, with $r = 0$

Figure 6.10

As with Kruskal's algorithm, the complexity of Prim's algorithm is dependent on specific implementation details. A straightforward implementation of Prim's algorithm based on explicitly implementing $Cut(T)$ at each stage is inefficient. A more efficient implementation of Prim's algorithm is based on the following observations. For convenience, we set $w(uv) = \infty$ (or some sufficiently large number) whenever u and v are nonadjacent vertices in G. For each vertex $v \in V(G) - V(T)$, we let $\eta_T(v)$ denote a vertex u of T that is "nearest" to v; that is, $w(uv)$ equals the minimum of $w(xv)$ over all $x \in V(T)$. For example, in Figure 6.9 with $T = T_6$, $V(G) - V(T)$ consists of the vertices $v =$ 3,4,9,12,14, with $\eta_T(v) = 6,5,10,11,15$, respectively. Clearly, the following equality holds:

$$\min\{w(\{v, \eta_T(v)\}) \mid v \in V(G) - V(T)\} = \min\{w(e) \mid e \in Cut(T)\}. \qquad (6.5.1)$$

Thus, when implementing Prim's algorithm, we can maintain η_T at each stage rather than explicitly implementing $Cut(T)$.

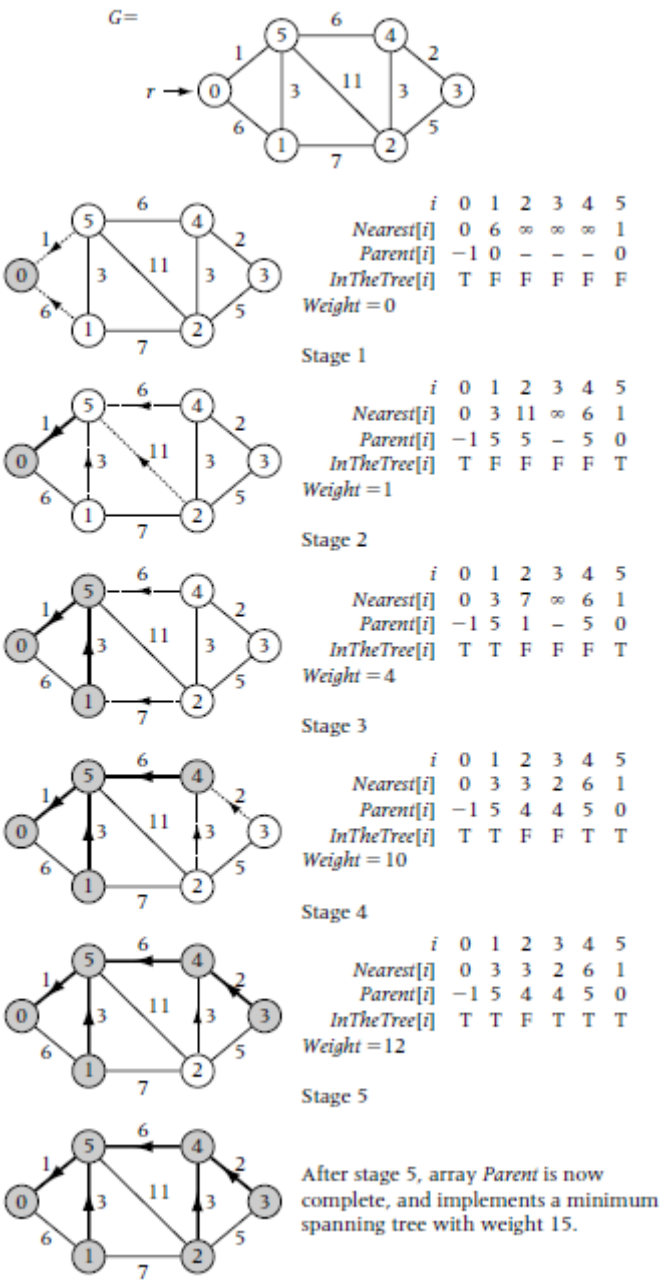
When implementing Prim's algorithm, the tree T is dynamically maintained using an array $Parent[0{:}n-1]$. The final state of $Parent[0{:}n-1]$ yields the parent array implementation of a minimum spanning tree T. Guided by (6.5.1), at any stage of the algorithm a subarray of $Parent[0{:}n-1]$ gives the parent array of the current tree T, whereas another subarray of $Parent[0{:}n-1]$ maintains the nearest weight vertices $\eta_T(v)$.

More precisely, *Parent*[*v*] has the following dynamic interpretation. If $v \in V(T)$, then *Parent*[*v*] is the parent of vertex *v* in *T*. If $v \notin V(T)$ but is adjacent to at least one vertex of *T*, then *Parent*[*v*] = $\eta_T(v)$. Otherwise, *Parent*[*v*] is undefined. Initially, *Parent*[*r*] = 0.

In the following pseudocode for procedure *Prim*, we also maintain a local array *Nearest*[0:*n* – 1], which is initialized to ∞'s, except for *Nearest*[*r*] which is set to 0. At each stage, if *Parent*[*v*] is defined, then *Nearest*[*v*] = *w*({*v*,*Parent*[*v*]}), otherwise *Nearest*[*v*] = ∞. The next vertex *u* to be added to the tree *T* is the one such that *Nearest*[*u*] is minimized over all $u \notin V(T)$. The arrays *Parent*[0:*n* – 1] and *Nearest*[0:*n* – 1] are examined for possible alteration at each stage for each vertex *v* not belonging to *T* that is adjacent to *u*. If *w*(*uv*) < *Nearest*[*v*], then we set *Parent*[*v*] = *u* and *Nearest*[*v*] = *w*(*uv*). The procedure *Prim* also maintains a Boolean array *InTheTree* to keep track of the vertices not in *T*.

```
procedure Prim(G,w,Parent[0:n – 1])
Input: G (a connected graph with vertex set V and edge set E)
          w (a weight function on E)
Output:   Parent[0:n – 1] (parent array of a minimum spanning tree}
    for v ← 0 to n – 1 do
        Nearest[v] ← ∞
        InTheTree[v] ← .false.
    endfor
    Parent[r] ← 0                        // root the tree at an arbitrary vertex r
    Nearest[r] ← 0
    for Stage ← 1 to n – 1 do
        Select vertex u that minimizes Nearest[u] over all u such that
                                                        InTheTree[u] = .false.
        InTheTree[u] ← .true.                    // add u to T
        for each vertex v such that uv ∈ E do    // update Nearest[v] and
            if .not. InTheTree[v] then           // Parent[v] for all v ∉ V(T)
                if w(uv) < Nearest[v] then       //that are adjacent to u
                    Nearest[v] ← w(uv)
                    Parent[v] ← u
                endif
            endif
        endfor
    endfor
end Prim
```

The action of procedure *Prim* is illustrated in Figure 6.11 for the same graph given in Figure 6.10.



Action of procedure *Prim* for the same graph as in Figure 6.10

Figure 6.11

Remark

Note that the procedure *Prim* terminates after only $n - 1$ stages, even though there are n vertices in the final minimum spanning tree. The reason is simple: After stage $n - 1$ has been completed, *InTheTree*[0:$n - 1$] is **.false.** for only one vertex w. Thus, another iteration of the **for** loop controlled by *Stage* would result in no change to the arrays

Nearest[0:$n-1$] and *Parent*[0:$n-1$]. In other words, the last vertex and edge in the minimum spanning tree come in for free.

Clearly, the procedure *Prim* has $\Theta(n^2)$ complexity, where we choose the comparison used in updating *Nearest*[0:$n-1$] in the inner **for** loop as the basic operation. The question arises as to whether procedure *Prim* is more or less efficient than the procedure *Kruskal*, which has $\Theta(m \log m)$ worst-case complexity. The answer depends on how many edges G has in comparison to the number of vertices. When the ratio m/n is large, then procedure *Prim* is more efficient, whereas when m/n is small, procedure *Kruskal* is better. For example, when the number of edges has the same order as the number of vertices (such as for graphs whose vertex degrees are bounded above by some fixed constant), procedure *Kruskal* has $\Theta(n\log n)$ worst-case complexity, which is significantly better than the $\Theta(n^2)$ worst-case complexity of procedure *Prim*. At the other extreme, when the number of edges is quadratic in the number of vertices (such as for the complete graph K_n, so that $m = n(n-1)/2$), procedure *Kruskal* has worst-case complexity $\Theta(n^2\log n)$, as compared to the $\Theta(n^2)$ worst-case complexity of procedure *Prim*.

Note that in procedure *Prim*, the operations performed on the array *Nearest*[u] are essentially those associated with a priority queue. Thus, procedure *Prim* could be rewritten (procedure *Prim2*) using a priority queue Q of the vertices not in the current tree T. The priority of a vertex u is the weight of the edge joining u to its nearest neighbor $\eta_T(u)$ in T. After removing a vertex u from the priority queue and placing it in T, then we must reduce the priorities of each vertex v adjacent to u that is now "nearer" to the tree. If the priority queue Q is implemented as a min-heap, then both the operation of removing an element from Q and the operation of changing the priority value of an element in Q have $O(\log n)$ complexity. Since there are n removals of an element from Q and the number of times that the priority of an vertex v is lowered is no more than its degree, it follows that the worst-case complexity of *Prim2* belongs to $O(n\log n + 2m\log n) = O(m\log n)$. We leave the pseudocode for procedure *Prim2* as an exercise.

6.6 Shortest Paths in Weighted Graphs and Digraphs

In the previous chapter we saw that breadth first search solves the problem of finding shortest paths from a given vertex r in a graph or digraph to every vertex v reachable from r, where the weight of a path is simply the number of edges in a path. In a network setting this is often referred to as minimizing the number of *hops* from r to v. Often in applications, it is not just the weight of a path as measured by number of hops, but its total weight obtained by summing the weights of its edges, where each edge e has been assigned a nonnegative real weight $w(e)$. For example, a graph might model a network of cities, and the weight might be the distance or driving time between adjacent cities. As another example, a digraph might model a network of (direct) airplane flights, and the weight of a flight might be its cost or its flying time.

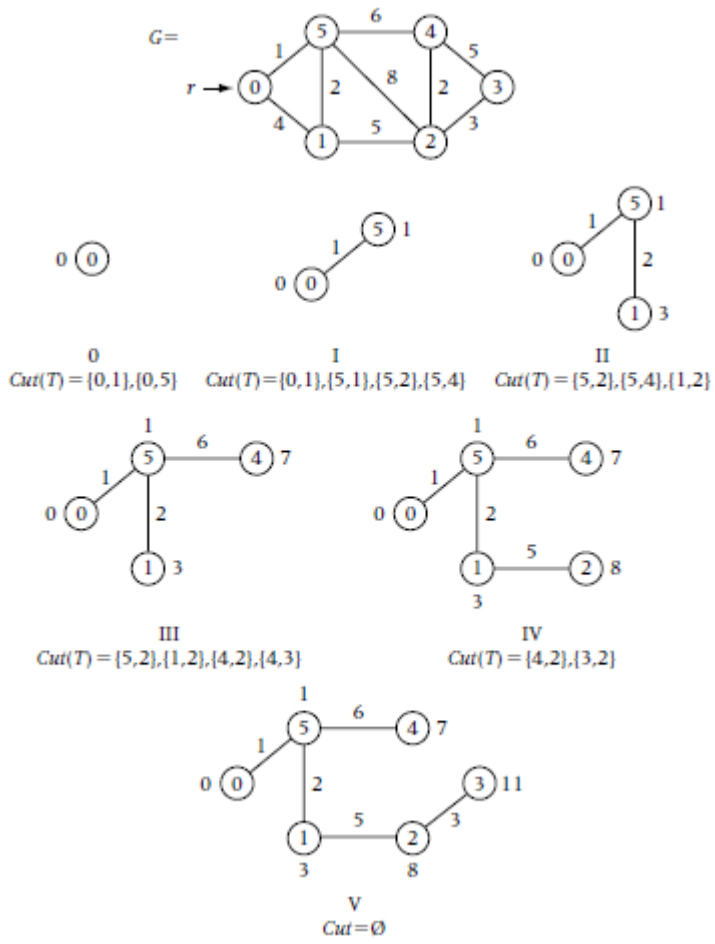
We now discuss an algorithm due to Dijkstra that grows a shortest path tree using the greedy method. For undirected graphs Dijkstra's algorithm follows a similar strategy to Prim's algorithm, differing only in the way the next edge is selected. At each stage in Dijkstra's algorithm, instead of selecting an edge of minimum weight in $Cut(T)$, we select an edge $uv \in Cut(T)$, $u \in V(T)$, so that the path from r to v in the augmented tree $T \cup$

{*uv*} is shortest; that is, such that *Dist*[*u*] + *w*(*uv*) is minimized over all edges $uv \in Cut(T)$, where *Dist*[*u*] is the weight of a path from *r* to *u* in *T*. The following pseudocode is a high-level description of Dijkstra's algorithm.

```
procedure Dijkstra(G,a,w,T)
Input: G (a graph with vertex set V = {0,...,n – 1} and edge set E)
           r (a vertex of G)
           w (a nonnegative weighting on E)
Output:    T (a shortest path tree rooted at r)
dcl Dist[0:n – 1] (an array initialized to ∞)
    T is initialized to be the tree consisting of the single vertex r
    Dist[r] ← 0

    while Cut(T) ≠ ∅ do
        select an edge uv ∈ Cut(T) such that Dist[u] + w(uv) is minimized
        add vertex v and edge uv to T
    endwhile
end Dijkstra
```

Figure 6.12 shows the tree *T* at each stage of Dijkstra's algorithm for a sample weighted graph *G*. In this figure the label on the outside of each vertex *v* already included in the growing tree *T* is the weight of the path from *v* to *a* in *T*.



Tree T and $Cut(T)$ at each stage of Dijkstra's algorithms

Figure 6.12

Let T_i be the tree generated by the procedure *Dijkstra* after i iterations of the **while** loop, $i = 0,1,...,t$, where t is the number of iterations of the loop ($t \leq n - 1$). For $v \in V(T_i)$, let $P_i^{(v)}$ denote the path from r to v in T_i, $i = 0,1,...,t$. We prove correctness of procedure *Dijkstra* by establishing the following lemma.

Lemma 6.6.1 For each vertex $v \in V(T_i)$, $P_i^{(v)}$ is a shortest path in G from r to v, $i = 0,1,...,t$.

Proof. We prove the lemma using mathematical induction on i.

Basis step: Clearly, the lemma is true for $i = 0$, since T_0 consists of the single vertex r.

Induction step: Assume that the lemma holds for T_i. The tree T_{i+1} is obtained from T_i by adding a single edge uv that minimizes $Dist[u] + w(uv)$ over all edges $uv \in Cut(T_i)$, where $Dist[u]$ is the weight of $P_u^{(i)}$. By induction assumption, for all $x \in V(T_i)$, $P_x^{(i+1)} = P_x^{(i)}$ is a shortest path in G from r to x. It remains to show that $P_v^{(i+1)}$ is a shortest path in

G from r to v. Consider any path Q in G from r to v. Since the deletion of the edges in $Cut(T_i)$ disconnects G with vertices r and v lying in different components, Q must contain edge xy from $Cut(T_i)$, $x \in V(T_i)$, such that the subpath Q_x of Q from r to x lies entirely in T_i. Then we have

$$\begin{aligned} weight(Q) &\geq weight(Q_x) + w(xy) && \text{(since the weight function is nonnegative)} \\ &\geq weight(P_x^{(i)}) + w(xy) && \text{(since } P_x^{(i)} \text{ is a shortest path in } G \text{ from } r \text{ to } x \text{ in } G\text{)} \\ &\geq weight(P_u^{(i)}) + w(uv) && \text{(by the greedy choice } Dijkstra \text{ makes)} \\ &= weight(P_u^{(i+1)}) \end{aligned}$$

Since Q was chosen to be an arbitrary path in G from r to v, it follows that $P_u^{(i+1)}$ is a shortest path G from r to v. This completes the induction step and the correctness proof of procedure *Dijkstra*. ■

As with Kruskal's and Prim's algorithms, the complexity of Dijkstra's algorithm is dependent on specific implementation details. We now implement a procedure *Dijkstra* that is similar to the procedure *Prim* in that the tree T is dynamically grown using an array *Parent*[0:n – 1]. We also maintain a Boolean array *InTheTree* to keep track of the vertices not in T. Again, rather than having to explicitly maintain the sets $Cut(T_i)$, we can efficiently select the next edge uv to be added to the tree by maintaining an array *Dist*[0:n – 1], where *Dist*[v] is the distance from r to v in T and *Dist*[v] = ∞ if v is not in T. Upon completion of *Dijkstra*, the distance from the root r to all the vertices of G is contained in the array *Dist*[0:n – 1]. At each intermediate stage *Dist*[v] has the following dynamic interpretation. If $v \in V(T)$, then *Dist*[v] is the weight of a path in T from r to v. If $v \notin V(T)$ and v is adjacent to a vertex of T, then *Dist*[v] is the minimum value of *Dist*[u] + $w(uv)$ over all $uv \in Cut(T)$. If $v \notin V(T)$ and v is not adjacent to any vertex of T, then *Dist*[v] = ∞. *Dist*[v] is initialized to ∞ for each vertex $v \neq r$ and *Dist*[r] is initialized to 0. Each time a vertex u is added to the tree T, we need only update *Parent*[v] and *Dist*[v] for those vertices v that are adjacent to u and do not belong to the tree T. If *Dist*[v] > *Dist*[u] + $w(uv)$, then we set *Parent*[v] = u and *Dist*[v] = *Dist*[u] + $w(uv)$.

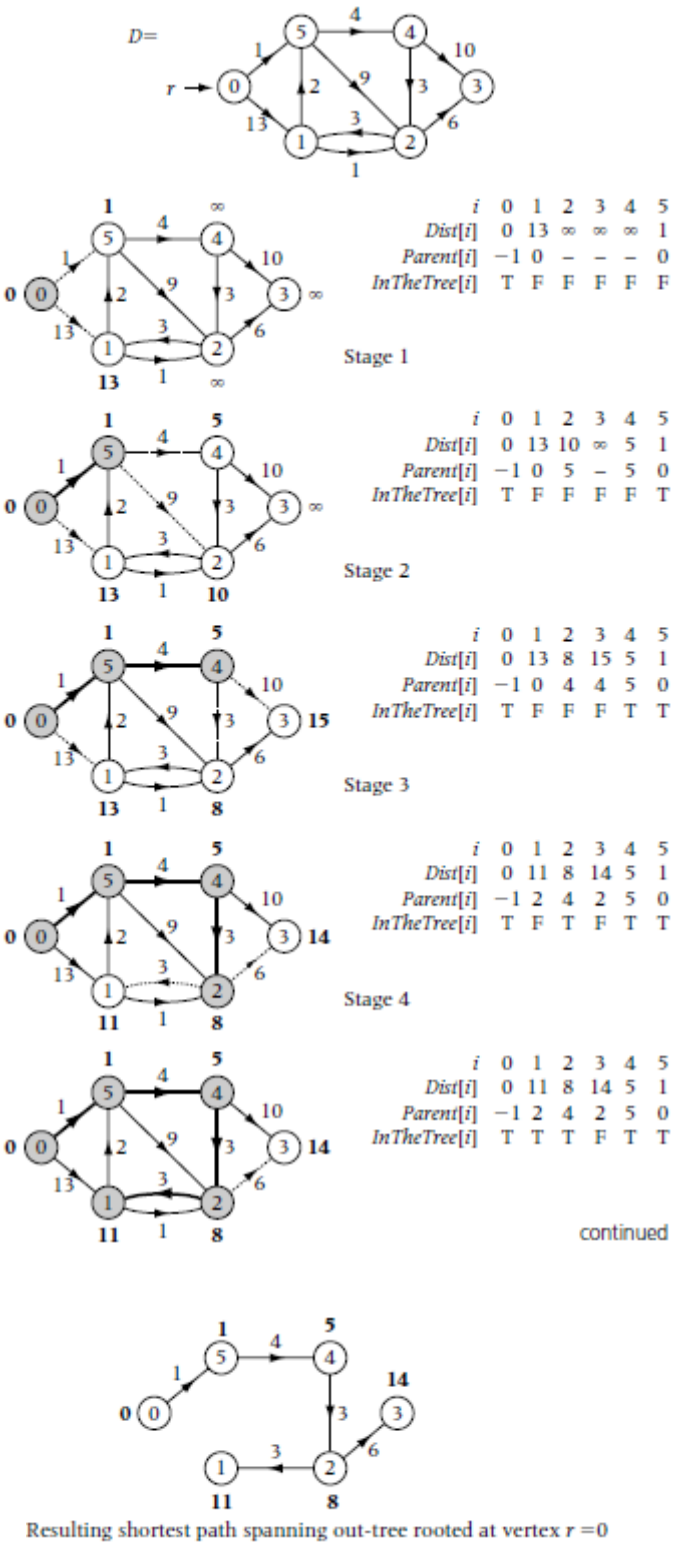
```
procedure Dijkstra(G,w,r,Parent[0:n – 1],Dist)
Input: G (a connected graph with vertex set V and edge set E)
           r (a vertex of G)
           w (a weight function on E)
Output:  Parent[0:n – 1] (parent array of a shortest path spanning tree)
           Dist[0:n – 1] (array of weights of shortest paths from a)
    for v ← 1 to n do       // initialize Dist[0:n – 1] and InTheTree[0:n – 1]
        Dist[v] ← ∞
        InTheTree[v] ← .false.
    endfor
    Dist[r] ← 0
    Parent[r] ← 0
    for Stage ← 1 to n – 1 do
        Select vertex u that minimizes Dist[u] over all u such that
```

```
                                                    InTheTree[u] = .false.
        InTheTree[u] ← .true.
        for each vertex v such that uv ∈ E do// update Dist[v] and Parent[v] arrays}
            if .not. InTheTree[v] then
                if Dist[u] + w(uv) < Dist[v] then
                    Dist[v] ← Dist[u] + w(uv)
                    Parent[v] ← u
                endif
            endif
        endfor
    endfor
end Dijkstra
```

Remarks

1. As with procedure Prim, procedure Dijkstra terminates after only $n - 1$ stages, even though there are n vertices in the final shortest path spanning tree. Another iteration of the for loop controlled by *Stage* would result in no change to the arrays Dist[0:$n - 1$] and Parent[0:$n - 1$].
2. In spite of the similarity between Prim's algorithm and Dijkstra's algorithm and the fact that they both generate spanning trees, they solve different problems. Simple examples show that a shortest path spanning tree can fail to be a minimum spanning tree (see Exercise 6.x).

The shortest path algorithms presented in this section for (undirected) graphs generalize naturally to digraphs. In fact, the code for the procedure *Dijkstra* applies essentially unchanged to digraphs. We merely have to be careful in the case of digraphs to note that uv corresponds to a directed edge $uv = (u,v)$ from u to v (as opposed to the undirected edge $uv = \{u,v\}$). We illustrate the action of Dijkstra's algorithm for a sample digraph in Figure 6.13, where the current value of *Dist*[v] is shown outside each vertex v.



i	0	1	2	3	4	5
Dist[i]	0	13	∞	∞	∞	1
Parent[i]	−1	0	–	–	–	0
InTheTree[i]	T	F	F	F	F	F

i	0	1	2	3	4	5
Dist[i]	0	13	10	∞	5	1
Parent[i]	−1	0	5	–	5	0
InTheTree[i]	T	F	F	F	F	T

i	0	1	2	3	4	5
Dist[i]	0	13	8	15	5	1
Parent[i]	−1	0	4	4	5	0
InTheTree[i]	T	F	F	F	T	T

i	0	1	2	3	4	5
Dist[i]	0	11	8	14	5	1
Parent[i]	−1	2	4	2	5	0
InTheTree[i]	T	F	T	F	T	T

i	0	1	2	3	4	5
Dist[i]	0	11	8	14	5	1
Parent[i]	−1	2	4	2	5	0
InTheTree[i]	T	T	T	F	T	T

Action of procedure *Dijkstra* for a sample digraph *D*

Figure 6.13

As with procedure *Prim*, the worst-case complexity of procedure *Dijkstra* belongs to $\Theta(n^2)$.

6.5 Closing Remarks

The greedy method is a powerful tool to use, when it applies, since solutions generated by this method are often elegant and very efficient. However, as we have seen, potential solutions generated by the greedy method must always be verified for correctness. Often, greedy decisions that are locally optimal do not lead in the end to solutions that are globally optimal.

The area of data compression is a very active area of research today, Because of limited bandwidth and the vast amount of visual and audio text files that are being transported over the Internet, data compression is absolutely necessary in order to handle these transmissions efficiently. Some of these data compression algorithms such as jpeg not only use the Huffman code algorithm, but also use transformations closely related to the Fast Fourier Transform that we discuss in Chapter 7.

In this chapter we discussed Dijkstra's algorithm for computing shortest paths based on the greedy method. In Chapter 8 we will discuss two other algorithms for computing shortest paths, the Floyd-Warshall and Bellman-Ford algorithms, based on the design strategy Dynamic Programming.

Exercises

Section 6.1 General Description

6.1 Given the standard U.S. coins, pennies (1¢), nickels (5¢), dimes (10¢), quarters (25¢), and half-dollars (50¢), show that the greedy method of making change always yields a solution using the fewest coins.

6.2 For a given integer base $b > 2$, suppose you have (an unlimited number of) coins of each of the denominations $b^0, b^1, ..., b^{n-1}$. Show that the greedy method of making change for these denominations always yields a solution using the fewest coins.

Section 6.2 Optimal Sequential Storage Order

6.3 Design and analyze a greedy algorithm for the optimal sequential storage problem when m tapes are available.

6.4 For the sequential storage order problem, consider the general case of minimizing $AveCost(\pi)$ as given by (6.2.1), where p_i denotes the probability that the object to be processed is b_i, $i = 0,...n-1$.

a. Show that the greedy strategy used when all the probabilities are equal (that is, sorting the objects by increasing order of costs c_i) fails to give an optimal solution for a general set of probabilities p_i.

b. Show that the greedy strategy of arranging the objects in decreasing order of probabilities also fails to give optimal solutions in general.

c. Show that the greedy strategy of arranging the objects in increasing order of the ratios c_i/p_i yields an optimal solution.

Section 6.3 The Knapsack Problem

6.5 Solve the following instance of the knapsack problem for capacity $C = 30$:

i	0	1	2	3	4	5	6	7
v_i	60	50	40	30	20	10	5	1
w_i	30	100	10	10	8	8	1	1

6.6 For the Knapsack problem with weights $w_0,w_1,...,w_{n-1}$ and values $v_0,v_1,...,v_{n-1}$, give a single example where both the greedy solution based on placing the objects in the knapsack in the order $w_0 \leq w_1 \leq ... \leq w_{n-1}$ as well as the greedy solution based on placing the objects in the knapsack in the order $v_0 \geq v_1 \geq ... \geq v_{n-1}$ yield suboptimal solutions.

6.7 a. Give an example where the greedy method described in Section 6.3 for the 0/1 Knapsack problem yields a suboptimal solution.

b. Show that the greedy method yields arbitrarily bad solutions; that is, given any $\varepsilon > 0$, find an example where the ratio of the greedy solution to the optimal solution is less than ε.

Section 6.4 Huffman Codes

6.8 Prove Proposition 6.4.1

6.9 Trace the action of *HuffmanCode* for the letters *a,b,c,d,e,f,g,h* occurring with frequencies 10,7,3,5,9,2,3,2.

6.10 Given the Huffman Code Tree in Figure 6.6, decode the string
10011110000110111100 1

6.11 Suppose the Huffman Tree has nodes with the structure

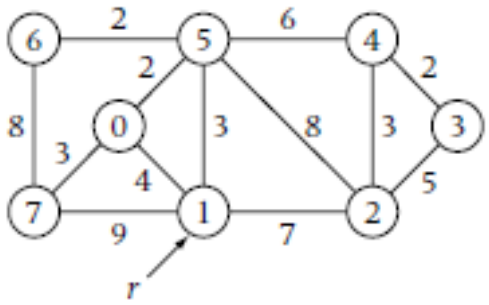
HuffmanTreeNode = **record**
Symbol: character
LeftChild: → *HuffmanTreeNode*
RightChld: → *Huffman TreeNode*
end *HuffmanTreeNode*

6.12 Given an arbitrary 2-tree T having n leaf nodes, find a set of frequencies such T is a Huffman tree for these frequencies.

6.13 Using a greedy algorithm to solve the *optimal merge pattern* problem. In this problem, we have n sorted files of lengths $f_0, f_1, \ldots, f_{n-1}$, and we wish to merge them into a single file by a sequence of merges of pairs of files. To merge two files of lengths m_1 and m_2 takes $m_1 + m_2$ operations.

6.5 Minimum Spanning Tree

6.14 Consider the following weighted graph *G*.



a. Trace the action of procedure *Kruskal* for *G*.

b. Trace the action of procedure *Prim* for *G*, with $r = 2$.

6.15 Give pseudocode for a version of the algorithm *Kruskal2* that maintains the weighted edges of the graph in a priority queue.

6.16 Prove the following two propositions used in the proof of the correctness of Kruskal's algorithm.

a. Adding an edge to a tree creates a unique cycle.

b. Removing an edge from a cycle in a connected graph still leaves the resulting graph connected.

6.17 a. Show how the problem of finding a minimum spanning tree in a weighted graph with repeated edge weights can be transformed into an equivalent problem with distinct edge weights

b. Show that the minimum spanning tree is unique when the edge weights are distinct.

6.18 Modify procedure *Kruskal* to accept *any* graph *G* (connected or disconnected) and to output a minimum (weight) forest, each of whose trees is a minimum spanning tree for a component of *G*.

6.19 Prove that Prim's algorithm yields a minimum spanning tree.

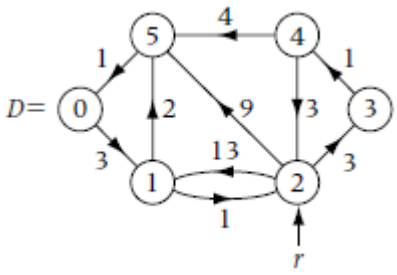
6.20 Analyze the complexity of the variant of procedure *Prim* based on an explicit implementing of *Cut*(*T*) at each stage of the algorithm. Compare its complexity to that of procedure *Prim*.

6.21 Give pseudocode for procedure *Prim2* using a priority queue.

6.22 The problem of finding a minimum spanning tree in an undirected graph *G* generalizes naturally to the problem of finding a minimum spanning in-tree (or out-tree) rooted at a given vertex *r* in a strongly connected digraph *D*. Prim's algorithm directly generalizes to digraphs, with the modification that *Cut*(*T*) is replaced by *CutIn*(*T*) (or *CutOut*(*T*)) consisting of all edges of *Cut*(*T*) having head in *T* (or having tail in *T*). Show that this greedy method doesn't always yield a spanning in-tree (or out-tree) rooted at *r*.

6.6 Shortest Paths in Weighted Graphs and Digraphs

6.23 a) Trace the action of procedure *Dijkstra* for the following digraph with initial vertex $r = 2$.



b) repeat for r = 3.

6.24 a. Trace the action of procedure *Dijkstra* for the graph *G* of Figure 6.x with $r = 1$.

b. Note that the shortest path tree generated in part (a) happens to coincide with the minimum spanning tree. Give an example of a weighted graph *G* with distinct weights and a vertex *r* such that the shortest spanning tree *T* generated by procedure *Dijkstra* is not a minimum spanning tree. In fact, find an example of a graph on *n* vertices where the shortest path tree has only one edge in common with the minimum spanning tree.

6.25 Given a weighted graph *G* and vertices *a*,*b*, consider the following greedy strategy that attempts to grow a shortest path from *a* to *b*. Choose an edge *e* having smallest weight among the remaining edges incident with the terminal vertex *u* of the path *P* previously generated, such that *e* contains no vertex of *P* other than *u*. These greedy choices continue until either *b* is reached or no choice for *e* exists. Give an example where the path so grown

a. never reaches *b*,

b. reaches *b*, but is not a shortest path from *a* to *b*.

6.26 Prove that the tree generated by procedure *Dijkstra* spans all vertices in the component of the graph *G* containing *a*.

6.27 Give pseudocode for *Dijkstra2* using a priority queue.

6.28 Show that Dijkstra's algorithm can fail in the case where some of the weights are negative, even though no negative cycles exist. (In particular, why doesn't adding a sufficiently large positive constant to each edge always work?)