EXERCISES

1. Find an optimal binary merge pattern for ten files whose lengths are $28, 32, 12, 5, 84, 53, 91, 35, 3,$ and 11.

2. (a) Show that if all internal nodes in a tree have degree k, then the number n of external nodes is such that $n \bmod (k-1) = 1$.
   (b) Show that for every n such that $n \bmod (k-1) = 1$, there exists a k-ary tree T with n external nodes (in a k-ary tree all nodes have degree at most k). Also show that all internal nodes of T have degree k.

3. (a) Show that if $n \bmod (k-1) = 1$, then the greedy rule described following Theorem 4.10 generates an optimal k-ary merge tree for all $(q_1, q_2, \ldots, q_n)$.
   (b) Draw the optimal three-way merge tree obtained using this rule when $(q_1, q_2, \ldots, q_{11}) = (3, 7, 8, 9, 15, 16, 18, 20, 23, 25, 28)$.

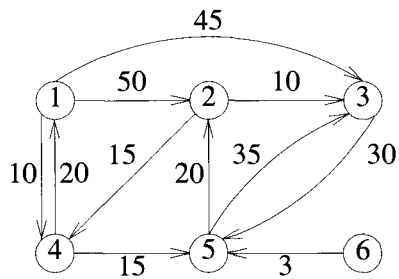
4. Obtain a set of optimal Huffman codes for the messages $(M_1, \ldots, M_7)$ with relative frequencies $(q_1, \ldots, q_7) = (4, 5, 7, 8, 10, 12, 20)$. Draw the decode tree for this set of codes.

5. Let T be a decode tree. An optimal decode tree minimizes $\sum q_i d_i$. For a given set of q's, let D denote all the optimal decode trees. For any tree $T \in D$, let $L(T) = \max\{d_i\}$ and let $\mathrm{SL}(T) = \sum d_i$. Schwartz has shown that there exists a tree $T^* \in D$ such that $L(T^*) = \min_{T \in D}\{L(T)\}$ and $\mathrm{SL}(T^*) = \min_{T \in D}\{SL(T)\}$.

   (a) For $(q_1, \ldots, q_8) = (1, 1, 2, 2, 4, 4, 4, 4)$ obtain trees $T1$ and $T2$ such that $L(T1) > L(T2)$.
   (b) Using the data of a, obtain $T1$ and $T2 \in D$ such that $L(T1) = L(T2)$ but $SL(T1) > SL(T2)$.
   (c) Show that if the subalgorithm Least used in algorithm Tree is such that in case of a tie it returns the tree with least depth, then Tree generates a tree with the properties of T^*.

4.8 SINGLE-SOURCE SHORTEST PATHS

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:



(a) Graph

Path	*Length*
1) 1, 4	10
2) 1, 4, 5	25
3) 1, 4, 5, 2	45
4) 1, 3	45

(b) Shortest paths from 1

Figure 4.15 Graph and shortest paths from vertex 1 to all destinations

- Is there a path from A to B?
- If there is more than one path from A to B, which is the shortest path?

The problems defined by these questions are special cases of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the *source*, and the last vertex the *destination*. The graphs are digraphs to allow for one-way streets. In the problem we consider, we are given a directed graph $G = (V, E)$, a weighting function *cost* for the edges of G, and a source vertex v_0. The problem is to determine the shortest paths from v_0 to *all* the remaining vertices of G. It is assumed that all the weights are positive. The shortest path between v_0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example 4.11 Consider the directed graph of Figure 4.15(a). The numbers on the edges are the weights. If node 1 is the source vertex, then the shortest path from 1 to 2 is $1, 4, 5, 2$. The length of this path is $10 + 15 + 20 = 45$. Even though there are three edges on this path, it is shorter than the path $1, 2$ which is of length 50. There is no path from 1 to 6. Figure 4.15(b) lists the shortest paths from node 1 to nodes $4, 5, 2$, and 3, respectively. The paths have been listed in nondecreasing order of path length. □

To formulate a greedy-based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by

one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way (and also a systematic way) to generate the shortest paths from v_0 to the remaining vertices is to generate these paths in nondecreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on. For the graph of Figure 4.15(a) the nearest vertex to $v_0 = 1$ is 4 ($cost[1,4] = 10$). The path $1,4$ is the first path generated. The second nearest vertex to node 1 is 5 and the distance between 1 and 5 is 25. The path $1,4,5$ is the next path generated. In order to generate the shortest paths in this order, we need to be able to determine (1) the next vertex to which a shortest path must be generated and (2) a shortest path to this vertex. Let S denote the set of vertices (including v_0) to which the shortest paths have already been generated. For w not in S, let $dist[w]$ be the length of the shortest path starting from v_0, going through only those vertices that are in S, and ending at w. We observe that:

1. If the next shortest path is to vertex u, then the path begins at v_0, ends at u, and goes through only those vertices that are in S. To prove this, we must show that all the intermediate vertices on the shortest path to u are in S. Assume there is a vertex w on this path that is not in S. Then, the v_0 to u path also contains a path from v_0 to w that is of length less than the v_0 to u path. By assumption the shortest paths are being generated in nondecreasing order of path length, and so the shorter path v_0 to w must already have been generated. Hence, there can be no intermediate vertex that is not in S.

2. The destination of the next path generated must be that of vertex u which has the minimum distance, $dist[u]$, among all vertices not in S. This follows from the definition of $dist$ and observation 1. In case there are several vertices not in S with the same $dist$, then any of these may be selected.

3. Having selected a vertex u as in observation 2 and generated the shortest v_0 to u path, vertex u becomes a member of S. At this point the length of the shortest paths starting at v_0, going though vertices only in S, and ending at a vertex w not in S may decrease; that is, the value of $dist[w]$ may change. If it does change, then it must be due to a shorter path starting at v_0 and going to u and then to w. The intermediate vertices on the v_0 to u path and the u to w path must all be in S. Further, the v_0 to u path must be the shortest such path; otherwise $dist[w]$ is not defined properly. Also, the u to w path can be chosen so as not to contain any intermediate vertices. Therefore,

we can conclude that if $dist[w]$ is to change (i.e., decrease), then it is because of a path from v_0 to u to w, where the path from v_0 to u is the shortest such path and the path from u to w is the edge $\langle u, w \rangle$. The length of this path is $dist[u] + cost[u, w]$.

The above observations lead to a simple Algorithm 4.14 for the single-source shortest path problem. This algorithm (known as Dijkstra's algorithm) only determines the lengths of the shortest paths from v_0 to all other vertices in G. The generation of the paths requires a minor extension to this algorithm and is left as an exercise. In the function ShortestPaths (Algorithm 4.14) it is assumed that the n vertices of G are numbered 1 through n. The set S is maintained as a bit array with $S[i] = 0$ if vertex i is not in S and $S[i] = 1$ if it is. It is assumed that the graph itself is represented by its cost adjacency matrix with $cost[i, j]$'s being the weight of the edge $\langle i, j \rangle$. The weight $cost[i, j]$ is set to some large number, ∞, in case the edge $\langle i, j \rangle$ is not in $E(G)$. For $i = j$, $cost[i, j]$ can be set to any nonnegative number without affecting the outcome of the algorithm.

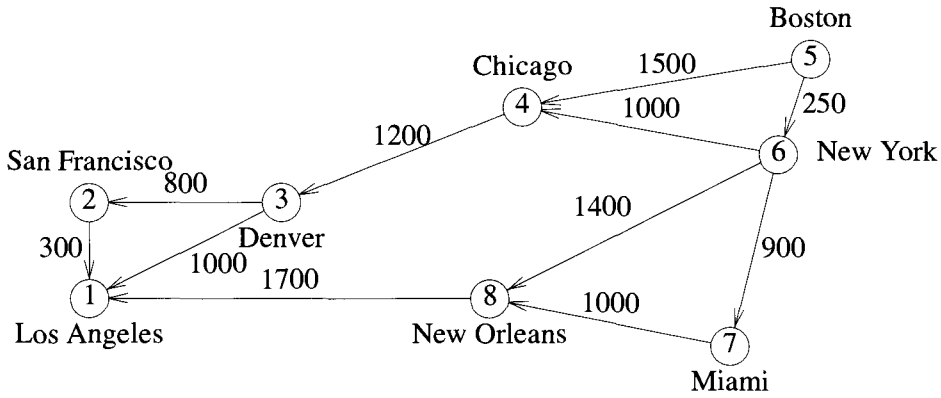
From our earlier discussion, it is easy to see that the algorithm is correct. The time taken by the algorithm on a graph with n vertices is $O(n^2)$. To see this, note that the **for** loop of line 7 in Algorithm 4.14 takes $\Theta(n)$ time. The **for** loop of line 12 is executed $n - 2$ times. Each execution of this loop requires $O(n)$ time at lines 15 and 16 to select the next vertex and again at the **for** loop of line 18 to update $dist$. So the total time for this loop is $O(n^2)$. In case a list t of vertices currently not in s is maintained, then the number of nodes on this list would at any time be $n - num$. This would speed up lines 15 and 16 and the **for** loop of line 18, but the asymptotic time would remain $O(n^2)$. This and other variations of the algorithm are explored in the exercises.

Any shortest path algorithm must examine each edge in the graph at least once since any of the edges could be in a shortest path. Hence, the minimum possible time for such an algorithm would be $\Omega(|E|)$. Since cost adjacency matrices were used to represent the graph, it takes $O(n^2)$ time just to determine which edges are in G, and so any shortest path algorithm using this representation must take $\Omega(n^2)$ time. For this representation then, algorithm ShortestPaths is optimal to within a constant factor. If a change to adjacency lists is made, the overall frequency of the **for** loop of line 18 can be brought down to $O(|E|)$ (since $dist$ can change only for vertices adjacent from u). If $V - S$ is maintained as a red-black tree (see Section 2.4.2), each execution of lines 15 and 16 takes $O(\log n)$ time. Note that a red-black tree supports the following operations in $O(\log n)$ time: insert, delete (an arbitrary element), find-min, and search (for an arbitrary element). Each update in line 21 takes $O(\log n)$ time as well (since an update can be done using a delete and an insertion into the red-black tree). Thus the overall run time is $O((n + |E|) \log n)$.

```
1   Algorithm ShortestPaths(v, cost, dist, n)
2   // dist[j], 1 ≤ j ≤ n, is set to the length of the shortest
3   // path from vertex v to vertex j in a digraph G with n
4   // vertices. dist[v] is set to zero. G is represented by its
5   // cost adjacency matrix cost[1 : n, 1 : n].
6   {
7       for i := 1 to n do
8       { // Initialize S.
9           S[i] := false; dist[i] := cost[v, i];
10      }
11      S[v] := true; dist[v] := 0.0; // Put v in S.
12      for num := 2 to n − 1 do
13      {
14          // Determine n − 1 paths from v.
15          Choose u from among those vertices not
16          in S such that dist[u] is minimum;
17          S[u] := true; // Put u in S.
18          for (each w adjacent to u with S[w] = false) do
19              // Update distances.
20              if (dist[w] > dist[u] + cost[u, w])) then
21                      dist[w] := dist[u] + cost[u, w];
22      }
23  }
```

Algorithm 4.14 Greedy algorithm to generate shortest paths

Example 4.12 Consider the eight vertex digraph of Figure 4.16(a) with cost adjacency matrix as in Figure 4.16(b). The values of *dist* and the vertices selected at each iteration of the **for** loop of line 12 in Algorithm 4.14 for finding all the shortest paths from Boston are shown in Figure 4.17. To begin with, S contains only Boston. In the first iteration of the **for** loop (that is, for $num = 2$), the city u that is not in S and whose $dist[u]$ is minimum is identified to be New York. New York enters the set S. Also the $dist[\,]$ values of Chicago, Miami, and New Orleans get altered since there are shorter paths to these cities via New York. In the next iteration of the **for** loop, the city that enters S is Miami since it has the smallest $dist[\,]$ value from among all the nodes not in S. None of the $dist[\,]$ values are altered. The algorithm continues in a similar fashion and terminates when only seven of the eight vertices are in S. By the definition of *dist*, the distance of the last vertex, in this case Los Angeles, is correct as the shortest path from Boston to Los Angeles can go through only the remaining six vertices. □



(a) Digraph

	1	2	3	4	5	6	7	8
1	0							
2	300	0						
3	100	800	0					
4			1200	0				
5				1500	0	250		
6				1000		0	900	1400
7							0	1000
8	1700							0

(b) Length-adjacency matrix

Figure 4.16 Figures for Example 4.12

One can easily verify that the edges on the shortest paths from a vertex v to all remaining vertices in a connected undirected graph G form a spanning tree of G. This spanning tree is called a *shortest-path spanning tree*. Clearly, this spanning tree may be different for different root vertices v. Figure 4.18 shows a graph G, its minimum-cost spanning tree, and a shortest-path spanning tree from vertex 1.

Iteration	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
2	{5,6}	7	+∞	+∞	+∞	1250	0	250	1150	1650
3	{5,6,7}	4	+∞	+∞	2450	1250	0	250	1150	1650
4	{5,6,7,4}	8	3350	+∞	2450	1250	0	250	1150	1650
5	{5,6,7,4,8}	3	3350	3250	2450	1250	0	250	1150	1650
6	{5,6,7,4,8,3}	2	3350	3250	2450	1250	0	250	1150	1650
	{5,6,7,4,8,3,2}									

Figure 4.17 Action of ShortestPaths

EXERCISES

1. Use algorithm ShortestPaths to obtain in nondecreasing order the lengths of the shortest paths from vertex 1 to all remaining vertices in the digraph of Figure 4.19.

2. Using the directed graph of Figure 4.20 explain why ShortestPaths will not work properly. What is the shortest path between vertices v_1 and v_7 ?

3. Rewrite algorithm ShortestPaths under the following assumptions:

   (a) G is represented by its adjacency lists. The head nodes are HEAD(1),..., HEAD(n) and each list node has three fields: VERTEX, COST, and LINK. COST is the length of the corresponding edge and n the number of vertices in G.

   (b) Instead of representing S, the set of vertices to which the shortest paths have already been found, the set $T = V(G) - S$ is represented using a linked list. What can you say about the computing time of your new algorithm relative to that of ShortestPaths?

4. Modify algorithm ShortestPaths so that it obtains the shortest paths in addition to the lengths of these paths. What is the computing time of your algorithm?