CHAPTER 6

GRAPHS

6.1 THE GRAPH ABSTRACT DATA TYPE

6.1.1 Introduction

The first recorded evidence of the use of graphs dates back to 1736, when Leonhard Euler used them to solve the now classical Königsberg bridge problem. In the town of Königsberg (now Kaliningrad) the river Pregel (Pregolya) flows around the island Kneiphof and then divides into two. There are, therefore, four land areas that have this river on its borders (see Figure 6.1(a)). These land areas are interconnected by seven bridges labeled a–g. The land areas themselves are labeled A–D. The Königsberg bridge problem is to determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to the starting land area. One possible walk is

- start from land area B
- walk across bridge a to island A
- take bridge e to area D

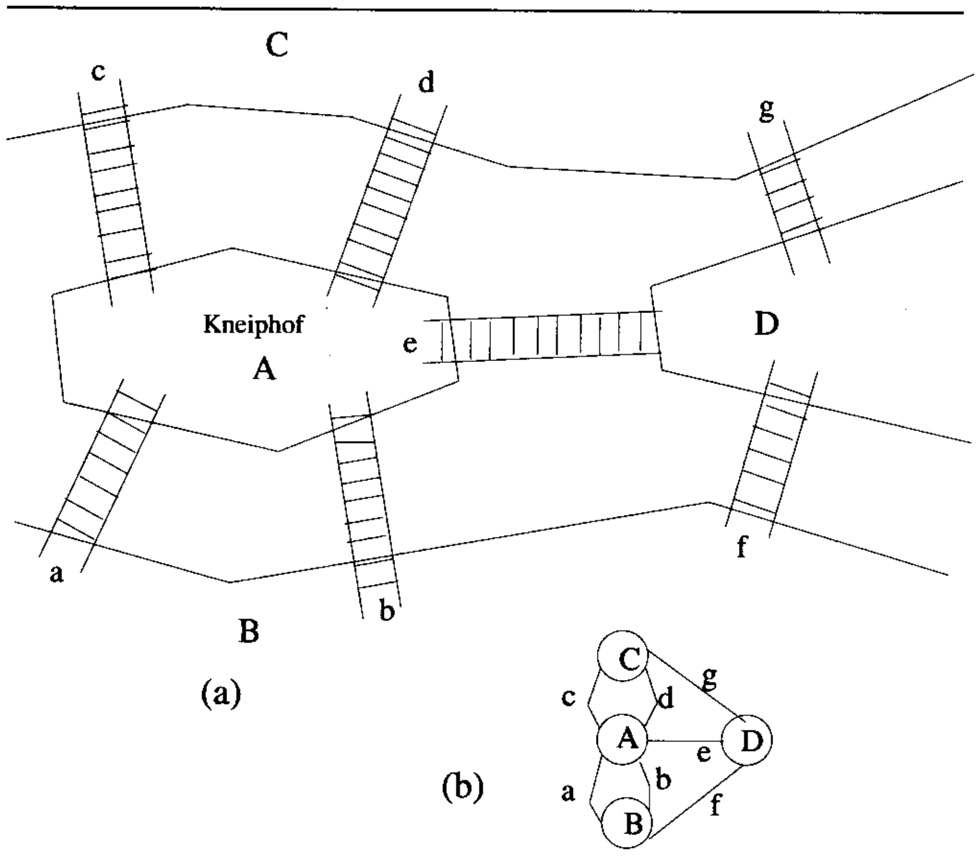


Figure 6.1: (a) Section of the river Pregel in Königsberg; (b) Euler's graph

- take bridge *g* to *C*
- take bridge *d* to *A*
- take bridge *b* to *B*
- take bridge *f* to *D*

This walk does not go across all bridges exactly once, nor does it return to the starting land area *B*. Euler answered the Königsberg bridge problem in the negative: The people of Königsberg will not be able to walk across each bridge exactly once and return to the starting point. He solved the problem by representing the land areas as vertices and the bridges as edges in a graph (actually a multigraph) as in Figure 6.1(b). His solution is elegant and applies to all graphs. Defining the *degree* of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each vertex is even. A walk that does this is called *Eulerian*. There is no Eulerian walk for the Königsberg bridge problem, as all four vertices are of odd degree.

Since this first application, graphs have been used in a wide variety of applications. Some of these applications are: analysis of electrical circuits, finding shortest routes, project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, and so on. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

6.1.2 Definitions

A graph, G, consists of two sets, V and E. V is a finite, nonempty set of *vertices*. E is a set of pairs of vertices; these pairs are called *edges*. $V(G)$ and $E(G)$ will represent the sets of vertices and edges, respectively, of graph G. We will also write $G = (V,E)$ to represent a graph. In an *undirected graph* the pair of vertices representing any edge is unordered. Thus, the pairs (u,v) and (v,u) represent the same edge. In a *directed graph* each edge is represented by a directed pair $<u,v>$; u is the *tail* and v the *head* of the edge[+]. Therefore, $<v,u>$ and $<u,v>$ represent two different edges. Figure 6.2 shows three graphs: G_1, G_2, and G_3. The graphs G_1 and G_2 are undirected. G_3 is a directed graph.

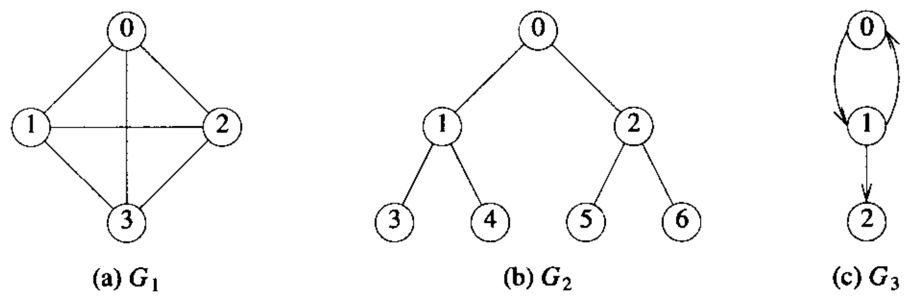


Figure 6.2: Three sample graphs

[+]Often, both the undirected edge (i,j) and the directed edge $<i,j>$ are written as (i,j). Which is meant is deduced from the context. In this book, we refrain from this practice.

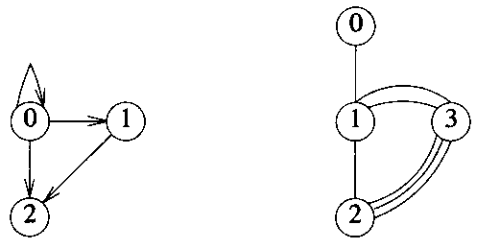
The set representation of each of these graphs is

$V(G_1) = \{0,1,2,3\}$; $E(G_1) = \{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$V(G_2) = \{0,1,2,3,4,5,6\}$; $E(G_2) = \{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$V(G_3) = \{0,1,2\}$; $E(G_3) = \{<0,1>,<1,0>,<1,2>\}$.

Notice that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is a tree; the graphs G_1 and G_3 are not.

Since we define the edges and vertices of a graph as sets, we impose the following restrictions on graphs:

(1) A graph may not have an edge from a vertex, v, back to itself. That is, edges of the form (v, v) and $<v, v>$ are not legal. Such edges are known as *self edges* or *self loops*. If we permit self edges, we obtain a data object referred to as a *graph with self edges*. An example is shown in Figure 6.3(a).



(a) Graph with a self edge (b) Multigraph

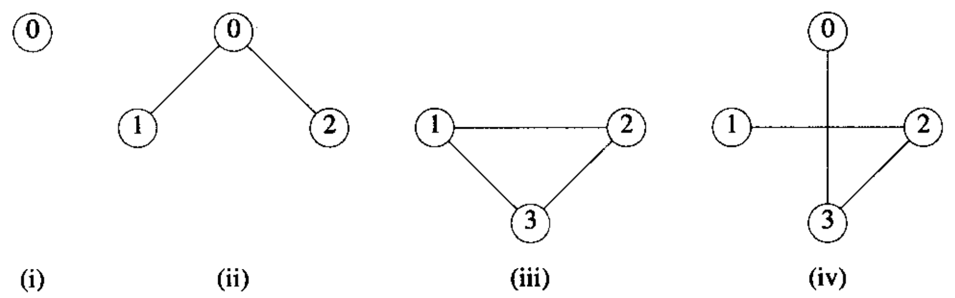
Figure 6.3: Examples of graphlike structures

(2) A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as a *multigraph* (see Figure 6.3(b)).

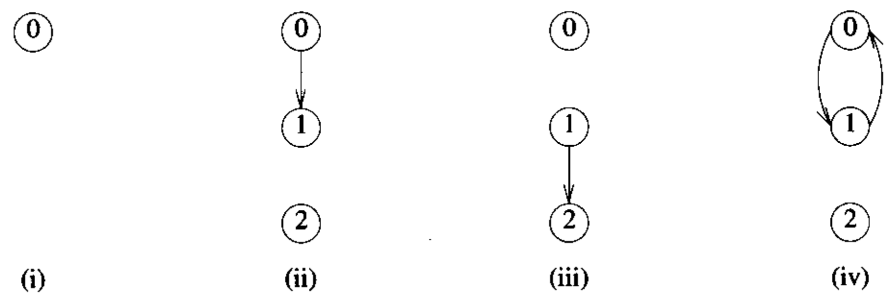
The number of distinct unordered pairs (u,v) with $u \neq v$ in a graph with n vertices is $n(n-1)/2$. This is the maximum number of edges in any n-vertex, undirected graph. An n-vertex, undirected graph with exactly $n(n-1)/2$ edges is said to be *complete*. The graph G_1 of Figure 6.2(a) is the complete graph on four vertices, whereas G_2 and G_3 are not complete graphs. In the case of a directed graph on n vertices, the maximum number of edges is $n(n-1)$.

If (u,v) is an edge in $E(G)$, then we shall say the vertices u and v are *adjacent* and that the edge (u,v) is *incident* on vertices u and v. The vertices adjacent to vertex 1 in G_2 are 3, 4, and 0. The edges incident on vertex 2 in G_2 are (0,2), (2,5), and (2,6). If $<u,v>$ is a directed edge, then vertex u is *adjacent to* v, and v is *adjacent from* u. The edge $<u,v>$ is incident to u and v. In G_3, the edges incident to vertex 1 are $<0,1>$, $<1,0>$, and $<1,2>$.

A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure 6.4 shows some of the subgraphs of G_1 and G_3.



(a) Some of the subgraphs of G_1



(b) Some of the subgraphs of G_3

Figure 6.4: Some subgraphs

A *path* from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \cdots, i_k, v$ such that $(u, i_1), (i_1, i_2), \cdots, (i_k, v)$ are edges in $E(G)$. If G' is directed, then the path consists of $<u, i_1>, <i_1, i_2>, \cdots, <i_k, v>$ edges in $E(G')$. The *length* of a path is the number of edges on it. A *simple path* is a path in which all vertices except possibly the first and last are distinct. A path such as (0,1), (1,3), (3,2), is also written as 0,1,3,2. Paths 0,1,3,2 and 0,1,3,1 of G_1 are both of length 3. The first is a simple path; the second is not. 0,1,2 is a simple directed path in G_3. 0,1,2,1 is not a path in G_3, as the edge $<2,1>$ is not in $E(G_3)$.

A *cycle* is a simple path in which the first and last vertices are the same. 0,1,2,0 is a cycle in G_1. 0,1,0 is a cycle in G_3. For the case of directed graphs we normally add the prefix "directed" to the terms cycle and path.

In an undirected graph, G, two vertices u and v are said to be *connected* iff there is a path in G from u to v (since G is undirected, this means there must also be a path from v to u). An undirected graph is said to be connected iff for every pair of distinct vertices u and v in $V(G)$ there is a path from u to v in G. Graphs G_1 and G_2 are connected, whereas G_4 of Figure 6.5 is not. A *connected component* (or simply a component), H, of an undirected graph is a *maximal* connected subgraph. By maximal, we mean that G contains no other subgraph that is both connected and properly contains H. G_4 has two components, H_1 and H_2 (see Figure 6.5).

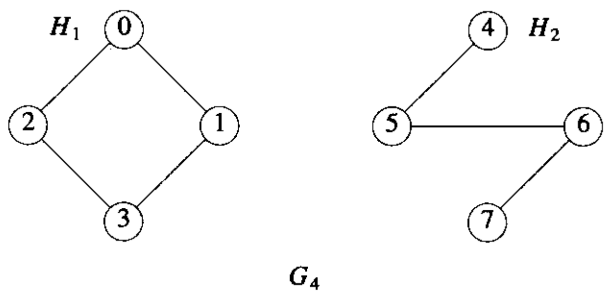


Figure 6.5: A graph with two connected components

A *tree* is a connected acyclic (i.e., has no cycles) graph.

A directed graph G is said to be *strongly connected* iff for every pair of distinct vertices u and v in $V(G)$, there is a directed path from u to v and also from v to u. The graph G_3 is not strongly connected, as there is no path from vertex 2 to 1. A *strongly connected component* is a maximal subgraph that is strongly connected. G_3 has two strongly connected components (see Figure 6.6).

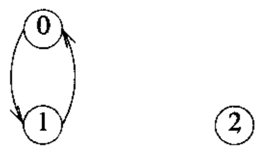


Figure 6.6: Strongly connected components of G_3

The degree of a vertex is the number of edges incident to that vertex. The degree of vertex 0 in G_1 is 3. If G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 1 of G_3 has in-degree 1, out-degree 2, and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then the number of edges is

$$e = \left(\sum_{i=0}^{n-1} d_i\right)/2$$

In the remainder of this chapter, we shall refer to a directed graph as a *digraph*. When we use the term *graph*, we assume that it is an undirected graph. Now that we have defined all the terminology we will need, let us consider the graph as an ADT. The resulting specification is given in ADT 6.1.

ADT *Graph* is
objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.
functions:
for all *graph* ∈ *Graph*, v, v_1, and v_2 ∈ *Vertices*

Graph Create()	::=	**return** an empty graph.
Graph InsertVertex(*graph*, v)	::=	**return** a graph with v inserted. v has no incident edges.
Graph InsertEdge(*graph*, v_1, v_2)	::=	**return** a graph with a new edge between v_1 and v_2.
Graph DeleteVertex(*graph*, v)	::=	**return** a graph in which v and all edges incident to it are removed.
Graph DeleteEdge(*graph*, v_1, v_2)	::=	**return** a graph in which the edge (v_1, v_2) is removed. Leave the incident nodes in the graph.
Boolean IsEmpty(*graph*)	::=	**if** (*graph* == empty graph) **return** *TRUE* **else return** *FALSE*.
List Adjacent(*graph*, v)	::=	**return** a list of all vertices that are adjacent to v.

ADT 6.1: Abstract data type *Graph*

The operations in ADT 6.1 are a basic set in that they allow us to create any arbitrary graph and do some elementary tests. In the later sections of this chapter we shall see functions that traverse a graph (depth first or breadth first search) and that determine if a graph has special properties (connected, biconnected, planar).

6.1.3 Graph Representations

Although several representations for graphs are possible, we shall study only the three most commonly used: adjacency matrices, adjacency lists, and adjacency multilists. Once again, the choice of a particular representation will depend upon the application one has in mind and the functions one expects to perform on the graph.

6.1.3.1 Adjacency Matrix

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional $n \times n$ array, say a, with the property that $a[i][j] = 1$ iff the edge (i, j) ($<i, j>$ for a directed graph) is in $E(G)$. $a[i][j] = 0$ if there is no such edge in G. The adjacency matrices for the graphs G_1, G_3, and G_4 are shown in Figure 6.7. The adjacency matrix for an undirected graph is symmetric, as the edge (i, j) is in $E(G)$ iff the edge (j, i) is also in $E(G)$. The adjacency matrix for a directed graph may not be symmetric (as is the case for G_3). The space needed to represent a graph using its adjacency matrix is n^2 bits. About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.

$$\begin{array}{c|cccc|} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(a) G_1

$$\begin{array}{c|ccc|} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \\ \hline \end{array}$$

(b) G_3

$$\begin{array}{c|cccccccc|} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$

(c) G_4

Figure 6.7: Adjacency matrices

From the adjacency matrix, one may readily determine if there is an edge connecting any two vertices i and j. For an undirected graph the degree of any vertex i is its row sum:

$$\sum_{j=0}^{n-1} a[i][j]$$

For a directed graph the row sum is the out-degree, and the column sum is the in-degree.

Suppose we want to answer a nontrivial question about graphs, such as, How many edges are there in G? or, Is G connected? Adjacency matrices will require at least $O(n^2)$ time, as $n^2 - n$ entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse (i.e., most of the terms in the adjacency matrix are zero) one would expect that the former question could be answered in significantly less time, say $O(e + n)$, where e is the number of edges in G, and $e << n^2/2$. Such a speed-up can be made possible through the use of a representation in which only the edges that are in G are explicitly stored. This leads to the next representation for graphs, adjacency lists.

6.1.3.2 Adjacency Lists

In this representation of graphs, the n rows of the adjacency matrix are represented as n chains (though sequential lists could be used just as well). There is one chain for each vertex in G. The nodes in chain i represent the vertices that are adjacent from vertex i. The *data* field of a chain node stores the index of an adjacent vertex. The adjacency lists for G_1, G_3, and G_4 are shown in Figure 6.8. Notice that the vertices in each chain are not required to be ordered. An array *adjLists* is used so that we can access the adjacency list for any vertex in $O(1)$ time. *adjLists*$[i]$ is a pointer to the first node in the adjacency list for vertex i.

For an undirected graph with n vertices and e edges, the linked adjacency lists representation requires an array of size n and $2e$ chain nodes. Each chain node has two fields. In terms of the number of bits of storage needed, the node count should be multiplied by $\log n$ for the array positions and $\log n + \log e$ for the chain nodes, as it takes $O(\log m)$ bits to represent a number of value m. If instead of chains, we use sequential lists, the adjacency lists may be packed into an integer array *node*$[n + 2e + 1]$. In one possible sequential mapping , *node*$[i]$ gives the starting point of the list for vertex i, $0 \le i < n$, and *node*$[n]$ is set to $n + 2e + 1$. The vertices adjacent from vertex i are stored in *node*$[i]$, $\cdots$, *node*$[i + 1] - 1$, $0 \le i < n$. Figure 6.9 shows the representation for the graph G_4 of Figure 6.5.

The degree of any vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list.

For a digraph, the number of list nodes is only e. The out-degree of any vertex may be determined by counting the number of nodes on its adjacency list. Determining the in-degree of a vertex is a little more complex. If there is a need to access repeatedly all vertices adjacent to another vertex, then it may be worth the effort to keep another set

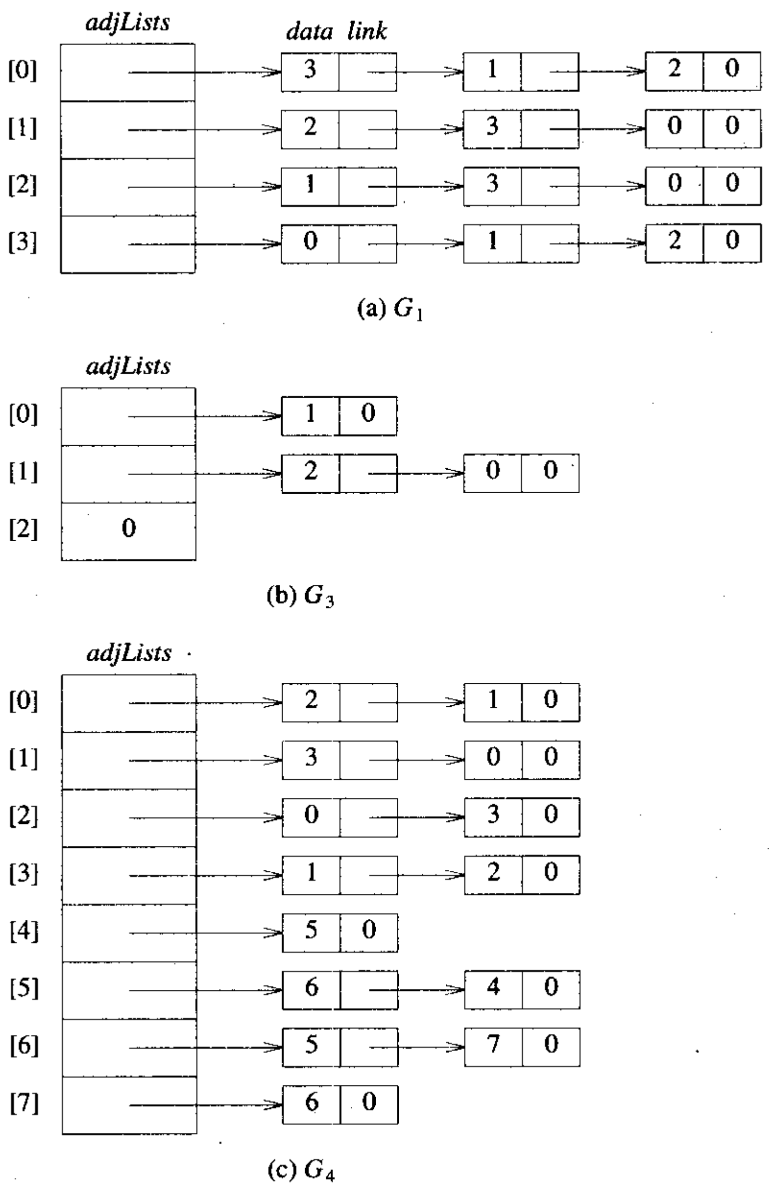


Figure 6.8: Adjacency lists

int *nodes* [*n* + 2**e* + 1];

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

Figure 6.9: Sequential representation of graph G_4

of lists in addition to the adjacency lists. This set of lists, called *inverse adjacency lists*, will contain one list for each vertex. Each list will contain a node for each vertex adjacent to the vertex it represents (see Figure 6.10).

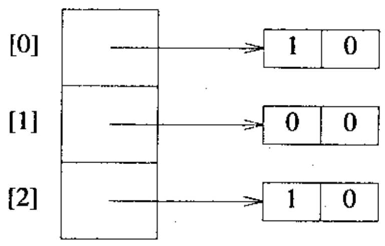


Figure 6.10: Inverse adjacency lists for G_3 (Figure 6.2(c))

Alternatively, one can adopt a simplified version of the list structure used for sparse matrix representation in Chapter 4. Figure 6.11 shows the resulting structure for the graph G_3 of Figure 6.2(c). The header nodes are stored sequentially. The first two fields in each node give the head and tail of the edge represented by the node, the remaining two fields are links for row and column chains.

6.1.3.3 Adjacency Multilists

In the adjacency-list representation of an undirected graph, each edge (u,v) is represented by two entries, one on the list for u and the other on the list for v. As we shall see, in some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes

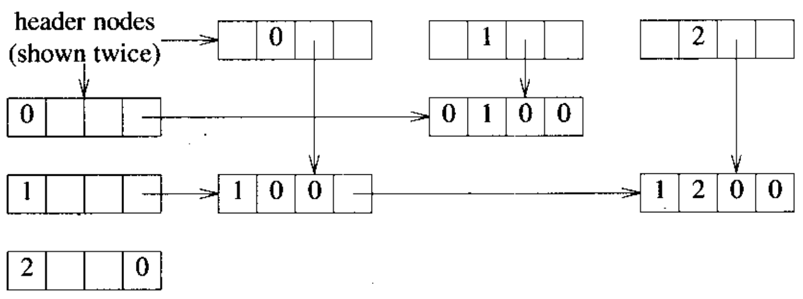


Figure 6.11: Orthogonal list representation for G_3 of Figure 6.2(c)

may be shared among several lists). For each edge there will be exactly one node, but this node will be in two lists (i.e., the adjacency lists for each of the two nodes to which it is incident). The new node structure is

m	*vertex*1	*vertex*2	*link*1	*link*2

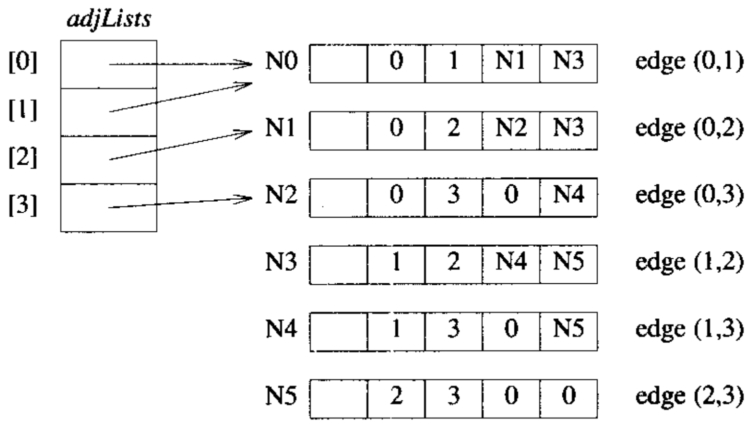
where *m* is a Boolean mark field that may be used to indicate whether or not the edge has been examined. The storage requirements are the same as for normal adjacency lists, except for the addition of the mark bit *m*. Figure 6.12 shows the adjacency multilists for G_1 of Figure 6.2(a).

6.1.3.4 Weighted Edges

In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. In these applications, the adjacency matrix entries $a[i][j]$ would keep this information too. When adjacency lists are used, the weight information may be kept in the list nodes by including an additional field, *weight*. A graph with weighted edges is called a *network*.

EXERCISES

1. Does the multigraph of Figure 6.13 have an Eulerian walk? If so, find one.


adjLists
[0]
[1]
[2]
[3]
N0 | 0 | 1 | N1 | N3 | edge (0,1)
N1 | 0 | 2 | N2 | N3 | edge (0,2)
N2 | 0 | 3 | 0 | N4 | edge (0,3)
N3 | 1 | 2 | N4 | N5 | edge (1,2)
N4 | 1 | 3 | 0 | N5 | edge (1,3)
N5 | 2 | 3 | 0 | 0 | edge (2,3)

The lists are	vertex 0:	N0 → N1 → N2
	vertex 1:	N0 → N3 → N4
	vertex 2:	N1 → N3 → N5
	vertex 3:	N2 → N4 → N5

Figure 6.12: Adjacency multilists for G_1 of Figure 6.2(a)

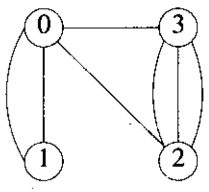


Figure 6.13: A multigraph

2. For the digraph of Figure 6.14 obtain
   (a) the in-degree and out-degree of each vertex
   (b) its adjacency-matrix
   (c) its adjacency-list representation

(d) its adjacency-multilist representation

(e) its strongly connected components

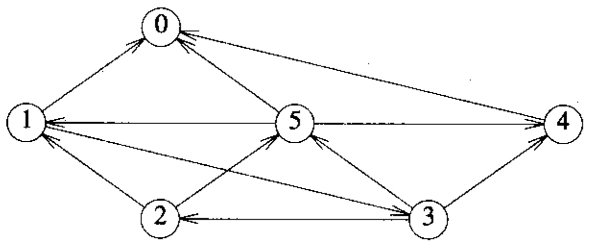


Figure 6.14: A digraph

3. Draw the complete undirected graphs on one, two, three, four, and five vertices. Prove that the number of edges in an n-vertex complete graph is $n(n-1)/2$.
4. Is the directed graph of Figure 6.15 strongly connected? List all the simple paths.

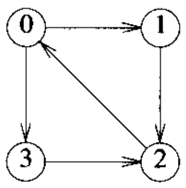


Figure 6.15: A directed graph

5. Obtain the adjacency-matrix, adjacency-list, and adjacency-multilist representations of the graph of Figure 6.15.
6. Show that the sum of the degrees of the vertices of an undirected graph is twice the number of edges.
7. (a) Let G be a connected, undirected graph on n vertices. Show that G must have at least $n-1$ edges and that all connected, undirected graphs with $n-1$ edges are trees.

   (b) What is the minimum number of edges in a strongly connected digraph on n

vertices? What form do such digraphs have?

8. For an undirected graph G with n vertices, prove that the following are equivalent:
   (a) G is a tree
   (b) G is connected, but if any edge is removed the resulting graph is not connected
   (c) For any two distinct vertices $u \in V(G)$ and $v \in V(G)$, there is exactly one simple path from u to v
   (d) G contains no cycles and has $n - 1$ edges
9. Write a C function to input the number of vertices and edges in an undirected graph. Next, input the edges one by one and to set up the linked adjacency-list representation of the graph. You may assume that no edge is input twice. What is the run time of your function as a function of the number of vertices and the number of edges?
10. Do the preceding exercise but this time set up the multilist representation.
11. Let G be an undirected, connected graph with at least one vertex of odd degree. Show that G contains no Eulerian walk.

6.2 ELEMENTARY GRAPH OPERATIONS

When we discussed binary trees in Chapter 5, we indicated that tree traversals were among the most frequently used tree operations. Thus, we defined and implemented preorder, inorder, postorder, and level order tree traversals. An analogous situation occurs in the case of graphs. Given an undirected graph, $G = (V, E)$, and a vertex, v, in $V(G)$ we wish to visit all vertices in G that are reachable from v, that is, all vertices that are connected to v. We shall look at two ways of doing this: *depth first search* and *breadth first search*. Depth first search is similar to a preorder tree traversal, while breadth first search resembles a level order tree traversal. In our discussion of depth first search and breadth first search, we shall assume that the linked adjacency list representation for graphs is used. The excercises explore the use of other representations.

6.2.1 Depth First Search

We begin the search by visiting the start vertex, v. In this simple application, visiting consists of printing the node's vertex field. Next, we select an unvisited vertex, w, from v's adjacency list and carry out a depth first search on w. We preserve our current position in v's adjacency list by placing it on a stack. Eventually our search reaches a vertex, u, that has no unvisited vertices on its adjacency list. At this point, we remove a vertex from the stack and continue processing its adjacency list. Previously visited vertices are discarded; unvisited vertices are visited and placed on the stack. The search terminates when the stack is empty. Although this sounds like a complicated function, it is easy to implement recursively. As indicated previously, it is similar to a preorder tree traversal since we visit a vertex and then continue with the next unvisited descendant. The recursive implementation of depth first search is presented in *dfs* (Program 6.1). This function uses a global array, *visited[MAX_VERTICES]*, that is initialized to *FALSE.* When we visit a vertex, *i*, we change *visited[i]* to *TRUE.* The declarations are:

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{/* depth first search of a graph beginning at v */
  nodePointer w;
  visited[v] = TRUE;
  printf("%5d",v);
  for (w = graph[v]; w; w = w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Program 6.1: Depth first search

Example 6.1: We wish to carry out a depth first search of graph G of Figure 6.16(a). Figure 6.16(b) shows the adjacency lists for this graph. If we initiate this search from vertex v_0, then the vertices of G are visited in the following order: v_0, v_1, v_3, v_7, v_4, v_5, v_2, v_6.

By examining Figures 6.16(a) and (b), we can verify that *dfs* (v_0) visits all vertices connected to v_0. This means that all the vertices visited, together with all edges in G incident to these vertices, form a connected component of G. □

Analysis of *dfs*: If we represent G by its adjacency lists, then we can determine the vertices adjacent to v by following a chain of links. Since *dfs* examines each node in the adjacency lists at most once, the time to complete the search is O(e). If we represent G by its adjacency matrix, then determining all vertices adjacent to v requires O(n) time. Since we visit at most n vertices, the total time is O(n^2). □

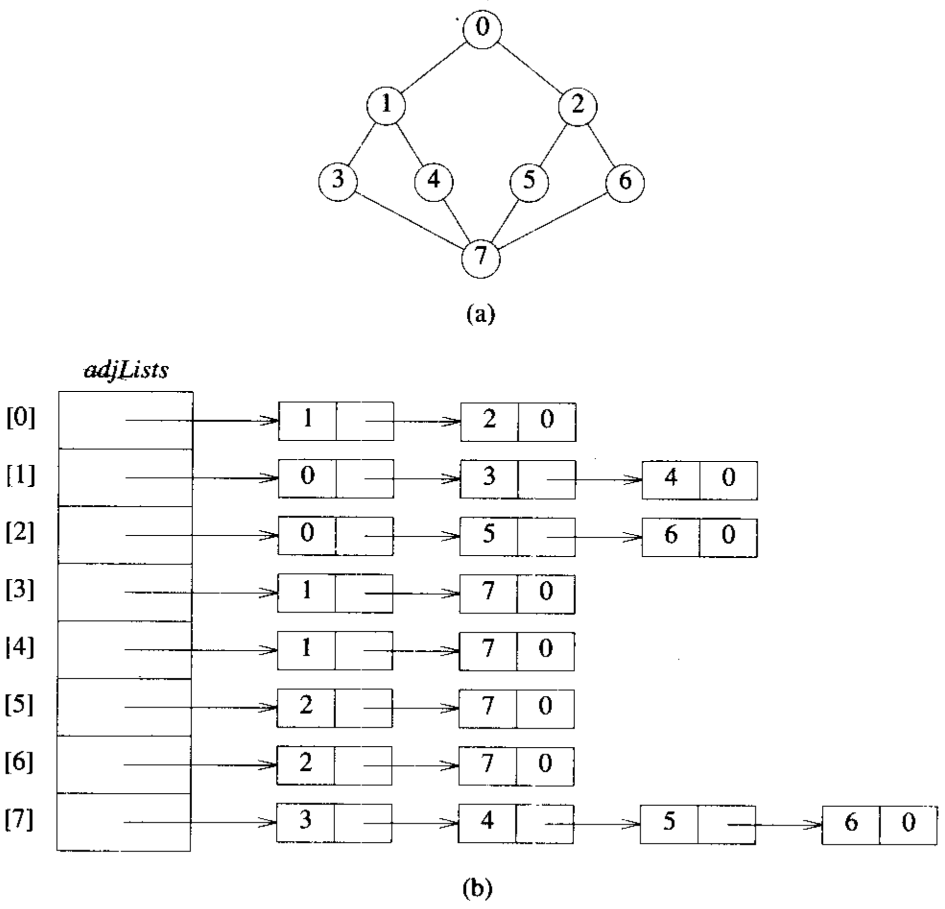


Figure 6.16: Graph G and its adjacency lists

6.2.2 Breadth First Search

Breadth first search starts at vertex v and marks it as visited. It then visits each of the vertices on v's adjacency list. When we have visited all the vertices on v's adjacency list, we visit all the unvisited vertices that are adjacent to the first vertex on v's adjacency list. To implement this scheme, as we visit each vertex we place the vertex in a queue. When we have exhausted an adjacency list, we remove a vertex from the queue and proceed by examining each of the vertices on its adjacency list. Unvisited vertices are visited and then placed on the queue; visited vertices are ignored. We have finished the search when the queue is empty.

To implement breadth first search, we use a dynamically linked queue as described in Chapter 4. Each queue node contains vertex and link fields. The *addq* and *deleteq* functions of Chapter 4 (Programs 4.7 and 4.8) will work correctly if we replace all references to *element* with **int**. The function *bfs* (Program 6.2) contains the C code to implement the breadth first search.

```
void bfs(int v)
{/* breadth first traversal of a graph, starting at v
    the global array visited is initialized to 0, the queue
    operations are similar to those described in
    Chapter 4, front and rear are global */
  nodePointer w;
  front = rear = NULL; /* initialize queue */
  printf("%5d",v);
  visited[v] = TRUE;
  addq(v);
  while (front) {
    v = deleteq();
    for (w = graph[v]; w; w = w->link)
      if (!visited[w->vertex]) {
        printf("%5d", w->vertex);
        addq(w->vertex);
        visited[w->vertex] = TRUE;
      }
    }
}
```

Program 6.2: Breadth first search of a graph

The queue definition and the function prototypes used by *bfs* are:

```
typedef struct queue *queuePointer;
typedef struct {
        int vertex;
        queuePointer link;
        } queue;
queuePointer front, rear;
void addq(int);
int deleteq();
```

Analysis of *bfs*: Since each vertex is placed on the queue exactly once, the **while** loop is iterated at most n times. For the adjacency list representation, this loop has a total cost of $d_0 + \cdots + d_{n-1} = O(e)$, where $d_i = degree\,(v_i)$. For the adjacency matrix representation, the **while** loop takes $O(n)$ time for each vertex visited. Therefore, the total time is $O(n^2)$. As was true of *dfs*, all vertices visited, together with all edges incident to them, form a connected component of G. □

6.2.3 Connected Components

We can use the two elementary graph searches to create additional, more interesting, graph operations. For illustrative purposes, let us look at the problem of determining whether or not an undirected graph is connected. We can implement this operation by simply calling either *dfs* (0) or *bfs* (0) and then determining if there are any unvisited vertices. For example, the call *dfs* (0) applied to graph G_4 of Figure 6.5 terminates without visiting vertices 4, 5, 6, and 7. Therefore, we can conclude that graph G_4 is not connected. The computing time for this operation is $O(n + e)$ if adjacency lists are used.

A closely related problem is that of listing the connected components of a graph. This is easily accomplished by making repeated calls to either *dfs* (v) or *bfs* (v) where v is an unvisited vertex. The function *connected* (Program 6.3) carries out this operation. Although we have used *dfs*, *bfs* may be used with no change in the time complexity.

```
void connected(void)
{/* determine the connected components of a graph */
int i;
for (i = 0; i < n; i++)
   if(!visited[i]) {
      dfs(i);
      printf("\n");
   }
}
```

Program 6.3: Connected components

Analysis of *connected*: If G is represented by its adjacency lists, then the total time taken by *dfs* is $O(e)$. Since the **for** loop takes $O(n)$ time, the total time needed to generate all the connected components is $O(n + e)$.

If G is represented by its adjacency matrix, then the time needed to determine the connected components is $O(n^2)$. □

6.2.4 Spanning Trees

When graph G is connected, a depth first or breadth first search starting at any vertex visits all the vertices in G. The search implicitly partitions the edges in G into two sets: T (for tree edges) and N (for nontree edges). T is the set of edges used or traversed during the search and N is the set of remaining edges. We can determine the set of tree edges by adding a statement to the **if** clause of either *dfs* or *bfs* that inserts the edge (v, w) into a linked list of edges. (T represents the head of this linked list.) The edges in T form a tree that includes all vertices of G. A *spanning tree* is any tree that consists solely of edges in G and that includes all the vertices in G. Figure 6.17 shows a graph and three of its spanning trees.

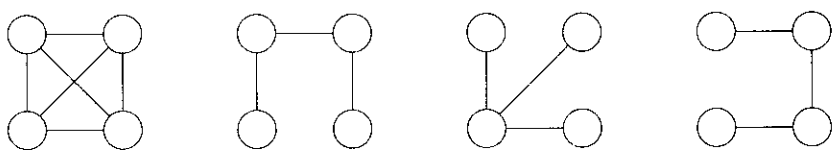


Figure 6.17: A complete graph and three of its spanning trees

As we just indicated, we may use either *dfs* or *bfs* to create a spanning tree. When *dfs* is used, the resulting spanning tree is known as a *depth first spanning tree*. When *bfs* is used, the resulting spanning tree is called a *breadth first spanning tree*. Figure 6.18 shows the spanning trees that result from a depth first and breadth first search starting at vertex v_0 in the graph of Figure 6.16.

Now suppose we add a nontree edge, (v, w), into any spanning tree, T. The result is a cycle that consists of the edge (v, w) and all the edges on the path from w to v in T. For example, if we add the nontree edge (7, 6) to the *dfs* spanning tree of Figure 6.18(a), the resulting cycle is 7, 6, 2, 5, 7. We can use this property of spanning trees to obtain an independent set of circuit equations for an electrical network.

Example 6.2 [***Creation of circuit equations***]: To obtain the circuit equations, we must first obtain a spanning tree for the electrical network. Then we introduce the nontree edges into the spanning tree one at a time. The introduction of each such edge produces a cycle. Next we use Kirchoff's second law on this cycle to obtain a circuit equation. The cycles obtained in this way are independent (we cannot obtain any of these cycles by taking a linear combination of the remaining cycles) since each contains a nontree edge that is not contained in any other cycle. Thus, the circuit equations are also independent. In fact, we can show that the cycles obtained by introducing the nontree edges one at a time into the spanning tree form a cycle basis. This means that we can

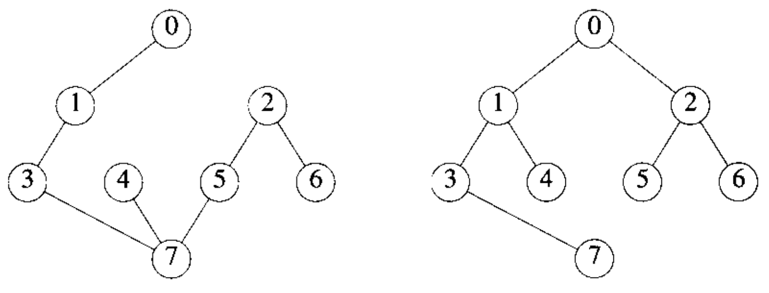


Figure 6.18: Depth-first and breadth-first spanning trees for graph of Figure 6.16

construct all other cycles in the graph by taking a linear combination of the cycles in the basis. (For further details, see the Harary text cited in the References and Selected Readings.) □

Let us examine a second property of spanning trees. A spanning tree is a *minimal subgraph*, G', of G such that $V(G') = V(G)$ and G' is connected. We define a minimal subgraph as one with the fewest number of edges. Any connected graph with n vertices must have at least $n - 1$ edges, and all connected graphs with $n - 1$ edges are trees. Therefore, we conclude that a spanning tree has $n - 1$ edges. (The exercises explore this property more fully.)

Constructing minimal subgraphs finds frequent application in the design of communication networks. Suppose that the vertices of a graph, G, represent cities and the edges represent communication links between cities. The minimum number of links needed to connect n cities is $n - 1$. Constructing the spanning trees of G gives us all feasible choices. However, we know that the cost of constructing communication links between cities is rarely the same. Therefore, in practical applications, we assign weights to the edges. These weights might represent the cost of constructing the communication link or the length of the link. Given such a weighted graph, we would like to select the spanning tree that represents either the lowest total cost or the lowest overall length. We assume that the cost of a spanning tree is the sum of the costs of the edges of that tree. Algorithms to obtain minimum cost spanning trees are studied in a later section.

6.2.5 Biconnected Components

The operations that we have implemented thus far are simple extensions of depth first and breadth first search. The next operation we implement is more complex and requires the introduction of additional terminology. We begin by assuming that G is an undirected connected graph.

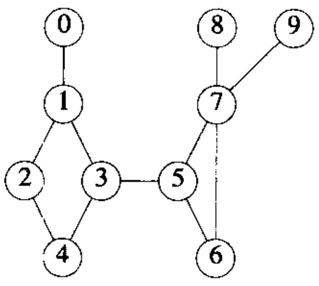
An *articulation point* is a vertex v of G such that the deletion of v, together with all edges incident on v, produces a graph, G', that has at least two connected components. For example, the connected graph of Figure 6.19 has four articulation points, vertices 1, 3, 5, and 7.

A *biconnected graph* is a connected graph that has no articulation points. For example, the graph of Figure 6.16 is biconnected, while the graph of Figure 6.19 obviously is not. In many graph applications, articulation points are undesirable. For instance, suppose that the graph of Figure 6.19(a) represents a communication network. In such graphs, the vertices represent communication stations and the edges represent communication links. Now suppose that one of the stations that is an articulation point fails. The result is a loss of communication not just to and from that single station, but also between certain other pairs of stations.

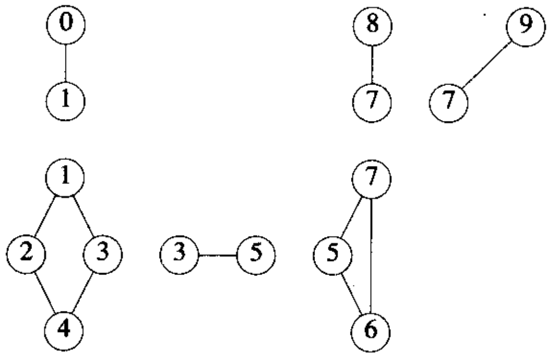
A *biconnected component* of a connected undirected graph is a *maximal biconnected subgraph, H,* of G. By maximal, we mean that G contains no other subgraph that is both biconnected and properly contains H. For example, the graph of Figure 6.19(a) contains the six biconnected components shown in Figure 6.19(b). The biconnected graph of Figure 6.16, however, contains just one biconnected component: the whole graph. It is easy to verify that two biconnected components of the same graph have no more than one vertex in common. This means that no edge can be in two or more biconnected components of a graph. Hence, the biconnected components of G partition the edges of G.

We can find the biconnected components of a connected undirected graph, G, by using any depth first spanning tree of C. For example, the function call *dfs* (3) applied to the graph of Figure 6.19(a) produces the spanning tree of Figure 6.20(a). We have redrawn the tree in Figure 6.20(b) to better reveal its tree structure. The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search. We call this number the *depth first number*, or *dfn*, of the vertex. For example, *dfn* (3) = 0, *dfn* (0) = 4, and *dfn* (9) = 8. Notice that vertex 3, which is an ancestor of both vertices 0 and 9, has a lower *dfn* than either of these vertices. Generally, if u and v are two vertices, and u is an ancestor of v in the depth first spanning tree, then *dfn* $(u) <$ *dfn* (v).

The broken lines in Figure 6.20(b) represent nontree edges. A nontree edge (u, v) is a *back edge iff* either u is an ancestor of v or v is an ancestor of u. From the definition of depth first search, it follows that all nontree edges are back edges. This means that the root of a depth first spanning tree is an articulation point *iff* it has at least two children. In addition, any other vertex u is an articulation point *iff* it has at least one child w such



(a) Connected graph



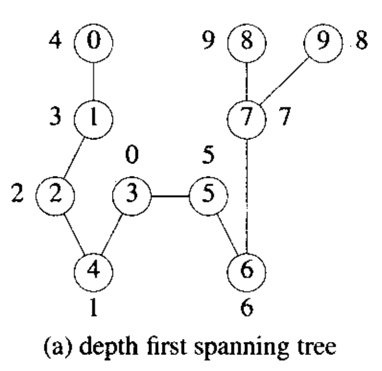
(b) Biconnected components

Figure 6.19: A connected graph and its biconnected components

that we cannot reach an ancestor of u using a path that consists of only w, descendants of w, and a single back edge. These observations lead us to define a value, *low*, for each vertex of G such that $low(u)$ is the lowest depth first number that we can reach from u using a path of descendants followed by at most one back edge:

$$\begin{aligned} low(u) = \min\{&dfn(u), \min\{low(w) \mid w \text{ is a child of } u\}, \\ &\min\{dfn(w) \mid (u, w) \text{ is a back edge}\}\} \end{aligned}$$

Therefore, we can say that u is an articulation point *iff* u is either the root of the spanning tree and has two or more children, or u is not the root and u has a child w such that $low(w) \geq dfn(u)$. Figure 6.21 shows the *dfn* and low values for each vertex of the spanning tree of Figure 6.20(b). From this table we can conclude that vertex 1 is an



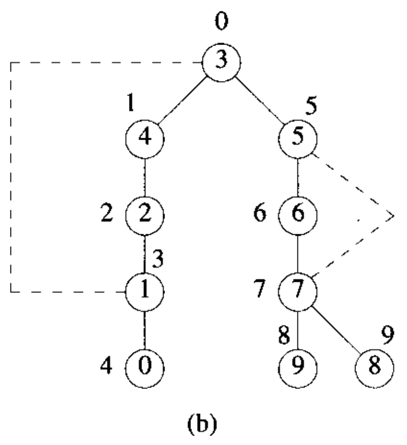


Figure 6.20: Depth first spanning tree of Figure 6.19(a)

articulation point since it has a child 0 such that *low*(0) = 4 ≥ *dfn*(1) = 3. Vertex 7 is also an articulation point since *low*(8) = 9 ≥ *dfn*(7) = 7, as is vertex 5 since *low*(6) = 5 ≥ *dfn*(5) = 5. Finally, we note that the root, vertex 3, is an articulation point because it has more than one child.

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	3	0	0	0	5	5	7	9	8

Figure 6.21: *dfn* and *low* values for *dfs* spanning tree with *root* = 3

We can easily modify *dfs* to compute *dfn* and *low* for each vertex of a connected undirected graph. The result is *dfnlow* (Program 6.4).

We invoke the function with the call *dfnlow(x, −1)*, where *x* is the starting vertex for the depth first search. The function uses a *MIN*2 macro that returns the smaller of its two parameters. The results are returned as two global variables, *dfn* and *low*. We also use a global variable, *num*, to increment *dfn* and *low*. The function *init* (Program 6.5) contains the code to correctly initialize *dfn*, *low*, and *num*. The global declarations are:

```
void dfnlow(int u, int v)
{/* compute dfn and low while performing a dfs search
    beginning at vertex u, v is the parent of u (if any) */
   nodePointer ptr;
   int w;
   dfn[u] = low[u] = num++;
   for (ptr = graph[u]; ptr; ptr = ptr→link) {
      w = ptr→vertex;
      if (dfn[w] < 0) { /* w is an unvisited vertex */
         dfnlow(w,u);
         low[u] = MIN2(low[u],low[w]);
      }
      else if (w != v)
         low[u] = MIN2(low[u],dfn[w]);
   }
}
```

Program 6.4: Determining *dfn* and *low*

```
#define MIN2(x,y) ((x) < (y) ? (x) : (y))
short int dfn[MAX_VERTICES];
short int low[MAX_VERTICES];
int num;
```

```
void init(void)
{
   int i;
   for (i = 0; i < n; i++) {
      visited[i] = FALSE;
      dfn[i] = low[i] = -1;
   }
   num = 0;
}
```

Program 6.5: Initialization of ~~*dfn* and~~ *low*

We can partition the edges of the connected graph into their biconnected components by adding some code to *dfnlow*. We know that *low[w]* has been computed following the return from the function call *dfnlow* (*w*, *u*). If *low* [*w*] ≥ *dfn* [*u*], then we have identified a new biconnected component. We can output all edges in a biconnected component if we use a stack to save the edges when we first encounter them. The function *bicon* (Program 6.6) contains the code. The same initialization function (Program 6.5) is used. The function call is *bicon* (*x*, −1), where *x* is the root of the spanning tree. Note that the parameters for the stack operations *push* and *pop* are slightly different from those used in Chapter 3.

```
void bicon(int u, int v)
{/* compute dfn and low, and output the edges of G by their
   biconnected components, v is the parent (if any) of u
   in the resulting spanning tree. It is assumed that all
   entries of dfn[] have been initialized to -1, num is
   initially to 0, and the stack is initially empty */
  nodePointer ptr;
  int w,x,y;
  dfn[u] = low[u] = num++;
  for (ptr = graph[u]; ptr; ptr = ptr→link) {
    w = ptr→vertex;
    if (v != w && dfn[w] < dfn[u])
      push(u,w); /* add edge to stack */
      if (dfn[w] <0) { /* w has not been visited */
        bicon(w,u);
        low[u] = MIN2(low[u],low[w]);
        if (low[w] >= dfn[u]) {
          printf("New biconnected component: ");
          do { /* delete edge from stack */
            pop(&x, &y);
            printf(" <%d,%d>",x,y);
          } while (!((x == u) && (y == w)));
          printf("\n");
        }
      }
      else if (w != v) low[u] = MIN2(low[u],dfn[w]);
  }
}
```

Program 6.6: Biconnected components of a graph

Analysis of *bicon*: The function *bicon* assumes that the connected graph has at least two vertices. Technically, a graph with one vertex and no edges is biconnected, but, our implementation does not handle this special case. The complexity of *bicon* is $O(n + e)$. We leave the proof of its correctness as an exercise. □

EXERCISES

1. Rewrite *dfs* so that it uses an adjacency matrix representation of graphs.
2. Rewrite *bfs* so that it uses an adjacency matrix representation.
3. Let G be a connected undirected graph. Show that no edge of G can be in two or more biconnected components of G. Can a vertex of G be in more than one biconnected component?
4. Let G be a connected graph and let T be any of its depth first spanning trees. Show that every edge of G that is not in T is a back edge relative to T.
5. Write the stack operations necessary to fully implement the *bicon* function. Use a dynamically linked representation for the stack.
6. Prove that function *bicon* correctly partitions the edges of a connected graph into the biconnected components of the graph.
7. A *bipartite graph, $G = (V, E)$,* is an undirected graph whose vertices can be partitioned into two disjoint sets V_1 and $V_2 = V - V_1$ with the properties:

   - no two vertices in V_1 are adjacent in G
   - no two vertices in V_2 are adjacent in G

   The graph G_4 of Figure 6.5 is bipartite. A possible partitioning of V is $V_1 = \{0, 3, 4, 6\}$ and $V_2 = \{1, 2, 5, 7\}$. Write a function to determine whether a graph is bipartite. If the graph is bipartite your function should obtain a partitioning of the vertices into two disjoint sets, V_1 and V_2, satisfying the two properties listed. Show that if G is represented by its adjacency lists, then this function has a computing time of $O(n + e)$, where $n = | V(G) |$ and $e = | E(G) |$ ($| \, |$ is the cardinality of the set, that is, the number of elements in it).
8. Show that every tree is a bipartite graph.
9. Prove that a graph is bipartite *iff* it contains no cycles of odd length.
10. Apply depth first and breadth first searches to the complete graph on four vertices. List the vertices in the order that they are visited.
11. Show how to modify *dfs* as it is used in *connected* to produce a list of all newly visited vertices.

12. Prove that when *dfs* is applied to a connected graph the edges of T form a tree.
13. Prove that when *bfs* is applied to a connected graph the edges of T form a tree.
14. An edge, (u, v), of a connected graph, G, is a *bridge iff* its deletion from G produces a graph that is no longer connected. In the graph of Figure 6.19, the edges (0, 1), (3, 5), (7, 8), and (7, 9) are bridges. Write a function that finds the bridges in a graph. Your function should have a time complexity of $O(n + e)$. (Hint: use *bicon* as a starting point.)
15. Using a complete graph with n vertices, show that the number of spanning trees is at least $2^{n-1} - 1$.

6.3 MINIMUM COST SPANNING TREES

The *cost* of a spanning tree of a weighted undirected graph is the sum of the costs (weights) of the edges in the spanning tree. A *minimum cost spanning tree* is a spanning tree of least cost. Three different algorithms can be used to obtain a minimum cost spanning tree of a connected undirected graph. All three use an algorithm design strategy called the *greedy method.* We shall refer to the three algorithms as Kruskal's, Prim's, and Sollin's algorithms, respectively.

In the greedy method, we construct an optimal solution in stages. At each stage, we make a decision that is the best decision (using some criterion) at this time. Since we cannot change this decision later, we make sure that the decision will result in a feasible solution. The greedy method can be applied to a wide variety of programming problems. Typically, the selection of an item at each stage is based on either a least cost or a highest profit criterion. A feasible solution is one which works within the constraints specified by the problem.

For spanning trees, we use a least cost criterion. Our solution must satisfy the following constraints:

(1) we must use only edges within the graph
(2) we must use exactly $n - 1$ edges
(3) we may not use edges that would produce a cycle.

6.3.1 Kruskal's Algorithm

Kruskal's algorithm builds a minimum cost spanning tree T by adding edges to T one at a time. The algorithm selects the edges for inclusion in T in nondecreasing order of their cost. An edge is added to T if it does not form a cycle with the edges that are already in T. Since G is connected and has $n > 0$ vertices, exactly $n - 1$ edges will be selected for inclusion in T.

Example 6.3: We will construct a minimum cost spanning tree of the graph of Figure 6.22(a). Figure 6.23 shows the order in which the edges are considered for inclusion, as well as the result and the changes (if any) in the spanning tree. For example, edge (0, 5) is the first considered for inclusion. Since it obviously cannot create a cycle, it is added to the tree. The result is the tree of Figure 6.22(c). Similarly, edge (2, 3) is considered next. It is also added to the tree, and the result is shown in Figure 6.22(d). This process continues until the spanning tree has $n-1$ edges (Figure 6.22(h)). The cost of the spanning tree is 99. □

Program 6.7 presents a formal description of Kruskal's algorithm. (We leave writing the C function as an exercise.) We assume that initially E is the set of all edges in G. To implement Kruskal's algorithm, we must be able to determine an edge with minimum cost and delete that edge. We can handle both of these operations efficiently if we maintain the edges in E as a sorted sequential list. As we shall see in Chapter 7, we can sort the edges in E in $O(e \log e)$ time. Actually, it is not necessary to sort the edges in E as long as we are able to find the next least cost edge quickly. Obviously a min heap is ideally suited for this task since we can determine and delete the next least cost edge in $O(\log e)$ time. Construction of the heap itself requires $O(e)$ time.

To check that the new edge, (v, w), does not form a cycle in T and to add such an edge to T, we may use the union-find operations discussed in Section 5.9. This means that we view each connected component in T as a set containing the vertices in that component. Initially, T is empty and each vertex of G is in a different set (see Figure 6.22(b)). Before we add an edge, (v, w), we use the find operation to determine if v and w are in the same set. If they are, the two vertices are already connected and adding the edge (v, w) would cause a cycle. For example, when we consider the edge (3, 2), the sets would be $\{0\}$, $\{1, 2, 3\}$, $\{5\}$, $\{6\}$. Since vertices 3 and 2 are already in the same set, the edge (3, 2) is rejected. The next edge examined is (1, 5). Since vertices 1 and 5 are in different sets, the edge is accepted. This edge connects the two components $\{1, 2, 3\}$ and $\{5\}$. Therefore, we perform a union on these sets to obtain the set $\{1, 2, 3, 5\}$.

Since the union-find operations require less time than choosing and deleting an edge (lines 3 and 4), the latter operations determine the total computing time of Kruskal's algorithm. Thus, the total computing time is $O(e \log e)$. Theorem 6.1 proves that Program 6.7 produces a minimum spanning tree of G.

Theorem 6.1: Let G be an undirected connected graph. Kruskal's algorithm generates a minimum cost spanning tree.

Proof: We shall show that:

(a) Kruskal's method produces a spanning tree whenever a spanning tree exists.

(b) The spanning tree generated is of minimum cost.

For (a), we note that Kruskal's algorithm only discards edges that produce cycles. We know that the deletion of a single edge from a cycle in a connected graph produces a

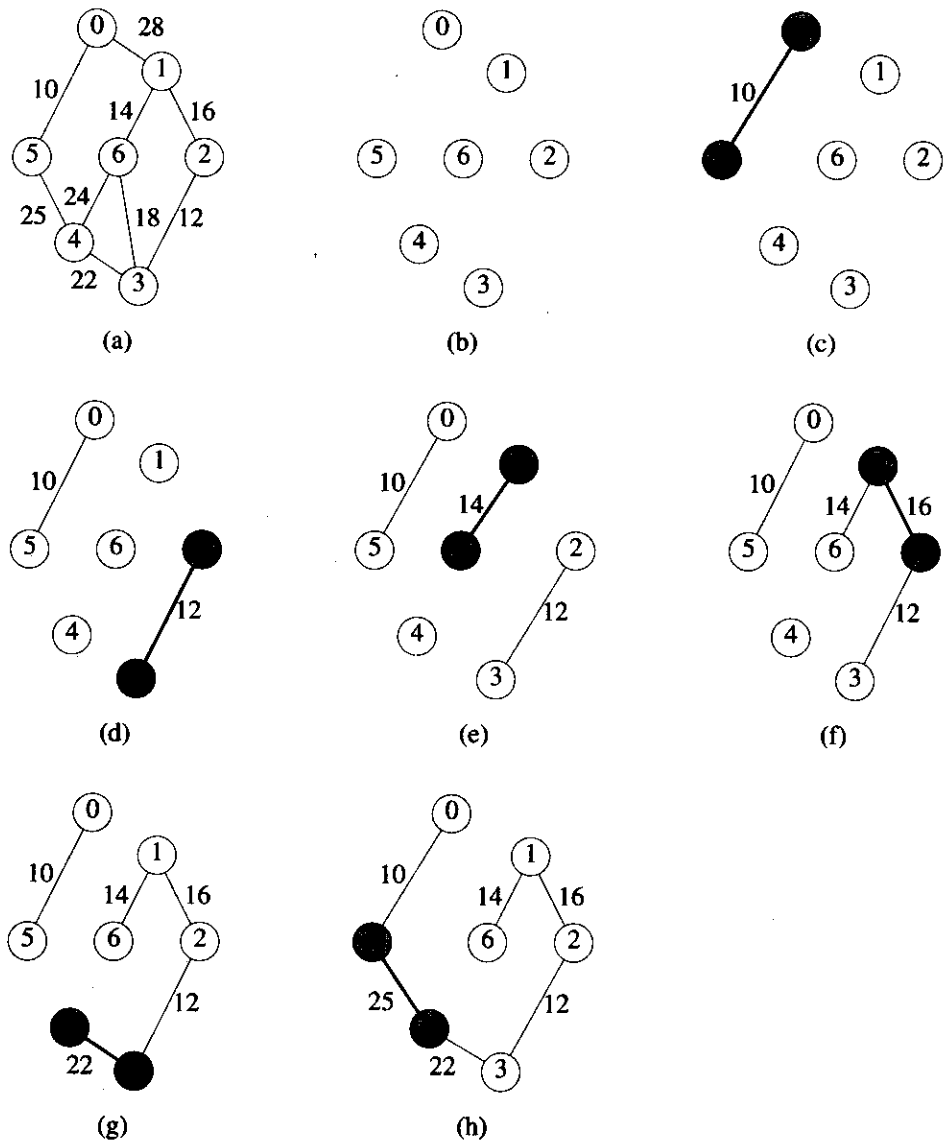


Figure 6.22: Stages in Kruskal's algorithm

Edge	Weight	Result	Figure
----	---	initial	Figure 6.22(b)
(0,5)	10	added to tree	Figure 6.22(c)
(2,3)	12	added	Figure 6.22(d)
(1,6)	14	added	Figure 6.22(e)
(1,2)	16	added	Figure 6.22(f)
(3,6)	18	discarded	
(3,4)	22	added	Figure 6.22(g)
(4,6)	24	discarded	
(4,5)	25	added	Figure 6.22(h)
(0,1)	28	not considered	

Figure 6.23: Summary of Kruskal's algorithm applied to Figure 6.22(a)

```
T = {};
while (T contains less than n-1 edges && E is not empty) {
   choose a least cost edge (v,w) from E;
   delete (v,w) from E;
   if ((v,w) does not create a cycle in T)
      add (v,w) to T;
   else
      discard (v,w);
}
if (T contains fewer than n-1 edges)
   printf("No spanning tree\n");
```

Program 6.7: Kruskal's algorithm

graph that is also connected. Therefore, if G is initially connected, the set of edges in T and E always form a connected graph. Consequently, if G is initially connected, the algorithm cannot terminate with $E = \{\}$ and $|T| < n - 1$.

Now let us show that the constructed spanning tree, T, is of minimum cost. Since G has a finite number of spanning trees, it must have at least one that is of minimum cost. Let U be such a tree. Both T and U have exactly $n - 1$ edges. If $T = U$, then T is of minimum cost and we have nothing to prove. So, assume that $T \neq U$. Let k, $k > 0$, be the number of edges in T that are not in U (k is also the number of edges in U that are not in

T).

We shall show that *T* and *U* have the same cost by transforming *U* into *T*. This transformation is done in *k* steps. At each step, the number of edges in *T* that are not in *U* is reduced by exactly 1. Furthermore, the cost of *U* is not changed as a result of the transformation. As a result, *U* after *k* transformation steps has the same cost as the initial *U* and contains exactly those edges that are in *T*. This implies that *T* is of minimum cost.

For each transformation step, we add one edge, *e*, from *T* to *U* and remove one edge, *f*, from *U*. We select the edges *e* and *f* in the following way:

(1) Let *e* be the least cost edge in *T* that is not in *U*. Such an edge must exist because $k > 0$.

(2) When we add *e* to *U*, we create a unique cycle. Let *f* be any edge on this cycle that is not in *T*. We know that at least one of the edges on this cycle is not in *T* because *T* contains no cycles.

Given the way *e* and *f* are selected, it follows that $V = U + \{e\} - \{f\}$ is a spanning tree and that *T* has exactly $k - 1$ edges that are not in *V*. We need to show that the cost of *V* is the same as the cost of *U*. Clearly, the cost of *V* is the cost of *U* plus the cost of the edge *e* minus the cost of the edge *f*. The cost of *e* cannot be less than the cost of *f* since this would mean that the spanning tree *V* has a lower cost than the tree *U*. This is impossible. If *e* has a higher cost than *f*, then *f* is considered before *e* by Kruskal's algorithm. Since it is not in *T*, Kruskal's algorithm must have discarded this edge at this time. Therefore, *f* together with the edges in *T* having a cost less than or equal to the cost of *f* must form a cycle. By the choice of *e*, all these edges are also in *U*. Thus, *U* must contain a cycle. However, since *U* is a spanning tree it cannot contain a cycle. So the assumption that *e* is of higher cost than *f* leads to a contradiction. This means that *e* and *f* must have the same cost. Hence, *V* has the same cost as *U*. □

6.3.2 Prim's Algorithm

Prim's algorithm, like Kruskal's, constructs the minimum cost spanning tree one edge at a time. However, at each stage of the algorithm, the set of selected edges forms a tree. By contrast, the set of selected edges in Kruskal's algorithm forms a forest at each stage. Prim's algorithm begins with a tree, *T*, that contains a single vertex. This may be any of the vertices in the original graph. Next, we add a least cost edge (u, v) to *T* such that $T \cup \{(u, v)\}$ is also a tree. We repeat this edge addition step until *T* contains $n - 1$ edges. To make sure that the added edge does not form a cycle, at each step we choose the edge (u, v) such that exactly one of *u* or *v* is in *T*. Program 6.8 contains a formal description of Prim's algorithm. *T* is the set of tree edges, and *TV* is the set of tree vertices, that is, vertices that are currently in the tree. Figure 6.24 shows the progress of Prim's algorithm on the graph of Figure 6.22(a).

```
T = {};
TV = {0}; /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
   let (u, v) be a least cost edge such that u ∈ TV and
   v ∉ TV;
   if (there is no such edge)
      break;
   add v to TV;
   add (u, v) to T;
}
if (T contains fewer than n-1 edges)
   printf("No spanning tree\n");
```

Program 6.8: Prim's algorithm

To implement Prim's algorithm, we assume that each vertex v that is not in TV has a companion vertex, $near(v)$, such that $near(v) \in TV$ and $cost(near(v), v)$ is minimum over all such choices for $near(v)$. (We assume that $cost(v, w) = \infty$ if $(v, w) \notin E$). At each stage we select v so that $cost(near(v), v)$ is minimum and $v \notin TV$. Using this strategy we can implement Prim's algorithm in $O(n^2)$, where n is the number of vertices in G. Asymptotically faster implementations are also possible. One of these results from the use of Fibonacci heaps which we examine in Chapter 9.

6.3.3 Sollin's Algorithm

Unlike Kruskal's and Prim's algorithms, Sollin's algorithm selects several edges for inclusion in T at each stage. At the start of a stage, the selected edges, together with all n graph vertices, form a spanning forest. During a stage we select one edge for each tree in the forest. This edge is a minimum cost edge that has exactly one vertex in the tree. Since two trees in the forest could select the same edge, we need to eliminate multiple copies of edges. At the start of the first stage the set of selected edges is empty. The algorithm terminates when there is only one tree at the end of a stage or no edges remain for selection.

Figure 6.25 shows Sollin's algorithm applied to the graph of Figure 6.22(a). The initial configuration of zero selected edges is the same as that shown in Figure 6.22(b). Each tree in this forest is a a single vertex. At the next stage, we select edges for each of the vertices. The edges selected are (0, 5), (1, 6), (2, 3), (3, 2), (4, 3), (5, 0), (6, 1). After eliminating the duplicate edges, we are left with edges (0, 5), (1, 6), (2, 3), and (4, 3). We add these edges to the set of selected edges, thereby producing the forest of Figure

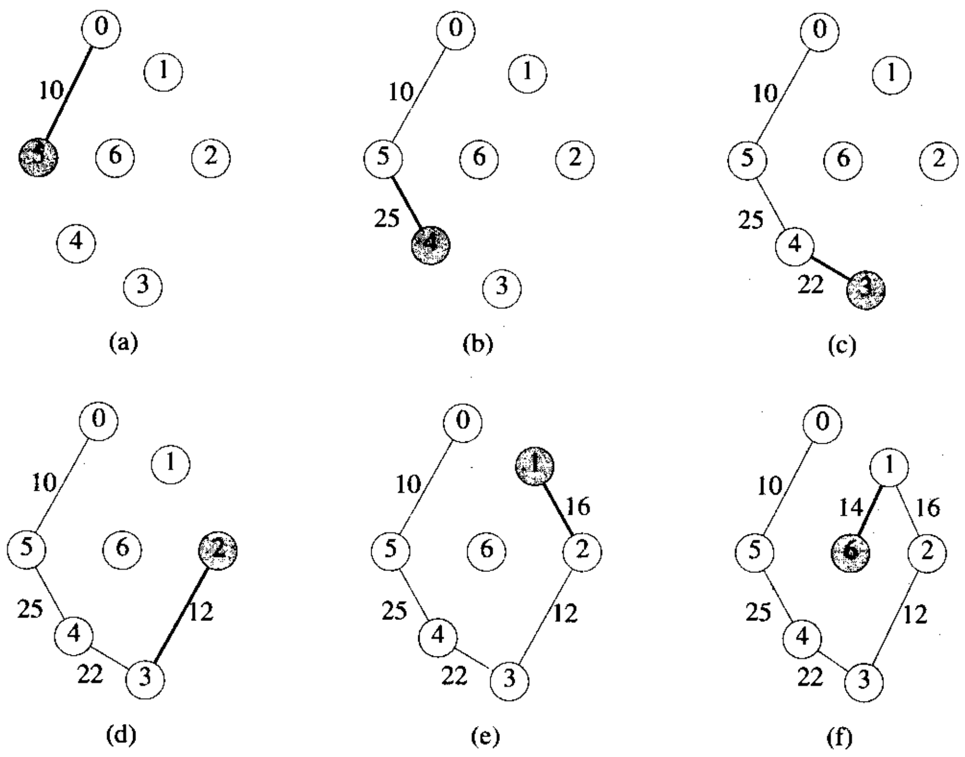


Figure 6.24: Stages in Prim's algorithm

6.25(a). In the next stage, the tree with vertex set {0, 5} selects edge (5, 4), and the two remaining trees select edge (1, 2). After these two edges are added, the spanning tree is complete, as shown in Figure 6.25(b). We leave the development of Sollin's algorithm into a C function and its correctness proof as exercises.

EXERCISES

1. Prove that Prim's algorithm finds a minimum cost spanning tree for every undirected connected graph.
2. Refine Prim's algorithm (Program 6.8) into a C function that finds a minimum cost spanning tree. The complexity of your function should be $O(n^2)$, where n is the number of vertices in the graph. Show that this is the case.

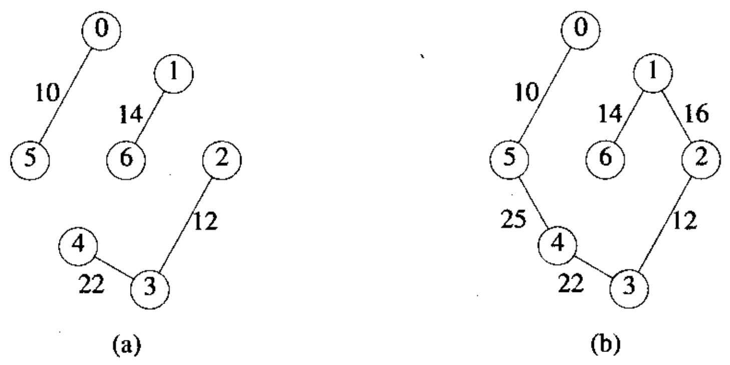


Figure 6.25: Stages in Sollin's algorithm

3. Prove that Sollin's algorithm finds a minimum cost spanning tree for every connected undirected graph.
4. What is the maximum number of stages in Sollin's algorithm? Give this as a function of the number of vertices, *n*, in the graph.
5. Write a C function that finds a minimum cost spanning tree using Sollin's algorithm. What is the complexity of your function?
6. Write a C function that finds a minimum cost spanning tree using Kruskal's algorithm. You may use the *union* and *find* functions from Chapter 5 and the *sort* function from Chapter 1 or the min heap functions from Chapter 5.
7. Show that if T is a spanning tree for an undirected graph G, then the addition of an edge $e, e \notin E(T)$ and $e \in E(G)$, to T creates a unique cycle.

6.4 SHORTEST PATHS AND TRANSITIVE CLOSURE

MapQuest, Google Maps, Yahoo! Maps, and MapNation are a few of the many Web systems that find a path between any two specified locations in the country. Path finding systemes generally use a graph to represent the highway system of a state or a country. In this graph, the vertices represent cities and the edges represent sections of the highway. Each edge has a weight representing the distance between the two cities connected by the edge. Alternatively, the weight could be an estimate of the time it takes to travel between the two cities. A motorist wishing to drive from city A to city B would be interested in answers to the following questions:

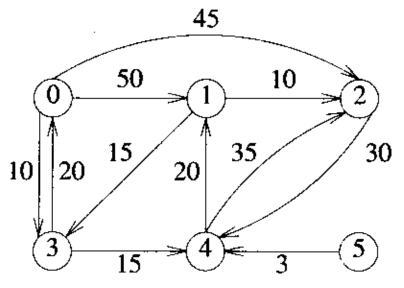
(1) Is there a path from A to B?

(2) If there is more than one path from A to B, which path is the shortest?

The problems defined by (1) and (2) above are special cases of the path problems we shall be studying in this section. An edge weight is also referred to as an edge length or edge cost. We shall use the terms weight, cost, and length interchangeably. The length (cost, weight) of a path is now defined to be the sum of the lengths (costs, weights) of the edges on that path, rather than the number of edges. The starting vertex of the path will be referred to as the *source* and the last vertex the *destination*. The graphs will be digraphs to allow for one-way streets.

6.4.1 Single Source/All Destinations: Nonnegative Edge Costs

In this problem we are given a directed graph, $G = (V, E)$, a weighting function, $w(e)$, $w(e) > 0$, for the edges of G, and a source vertex, v_0. We wish to determine a shortest path from v_0 to each of the remaining vertices of G. As an example, consider the graph of Figure 6.26(a). If v_0 is the source vertex, then the shortest path from v_0 to v_1 is v_0, v_2, v_3, v_1. The length of this path is $10 + 15 + 20 = 45$. Although there are three edges on this path, it is shorter than the path $v_0\ v_1$, which has a length of 50. Figure 6.26(b) lists the shortest paths from v_0 to v_1, v_2, v_3, and v_4 in nondecreasing order of path length. There is no path from v_0 to v_5.



(a) Graph

	Path	*Length*
1)	0, 3	10
2)	0, 3, 4	25
3)	0, 3, 4, 1	45
4)	0, 2	45

(b) Shortest paths from 0

Figure 6.26: Graph and shortest paths from vertex 0 to all destinations

We may use a greedy algorithm to generate the shortest paths in the order indicated in Figure 6.26(b). Let S denote the set of vertices, including v_0, whose shortest paths have been found. For w not in S, let *distance*$[w]$ be the length of the shortest path starting from v_0, going through vertices only in S, and ending in w. Generating the paths in nondecreasing order of length leads to the following observations:

(1) If the next shortest path is to vertex u, then the path from v_0 to u goes through only those vertices that are in S. To prove this we must show that all intermediate vertices on the shortest path from v_0 to u are already in S. Assume that there is a vertex w on this path that is not in S. Then, the path from v_0 to u also contains a path from v_0 to w which has a length that is less than the length of the path from v_0 to u. Since we assume that the shortest paths are generated in nondecreasing order of path length, we must have previously generated the path from v_0 to w. This is obviously a contradiction. Therefore, there cannot be any intermediate vertex that is not in S.

(2) Vertex u is chosen so that it has the minimum distance, *distance*[u], among all the vertices not in S. This follows from the definition of *distance* and observation (1). If there are several vertices not in S with the same distance, then we may select any one of them.

(3) Once we have selected u and generated the shortest path from v_0 to u, u becomes a member of S. Adding u to S can change the distance of shortest paths starting at v_0, going through vertices only in S, and ending at a vertex, w, that is not currently in S. If the distance changes, we have found a shorter such path from v_0 to w. This path goes through u. The intermediate vertices on this path are in S and its subpath from u to w can be chosen so as to have no intermediate vertices. The length of the shorter path is *distance* [u] + *length* (<u, w>).

We attribute these observations, along with the algorithm to determine the shortest paths from v_0 to all other vertices in G to Edsger Dijkstra. To implement Dijkstra's algorithm, we assume that the n vertices are numbered from 0 to $n - 1$. We maintain the set S as an array, *found*, with *found* [i] = *FALSE* if vertex i is not in S and *found* [i] = *TRUE* if vertex i is in S. We represent the graph by its cost adjacency matrix, with *cost*[i][j] being the weight of edge <i, j>. If the edge <i, j> is not in G, we set *cost*[i][j] to some large number. The choice of this number is arbitrary, although we make two stipulations regarding its value:

(1) The number must be larger than any of the values in the cost matrix.

(2) The number must be chosen so that the statement *distance*[u] + *cost*[u][w] does not produce an overflow into the sign bit.

Restriction (2) makes *INT_MAX* (defined in <*limits.h*>) a poor choice for nonexistent edges. For $i = j$, we may set *cost*[i][j] to any nonnegative number without affecting the outcome. For the digraph of Figure 6.26(a), we may set the cost of a nonexistent edge with $i \neq j$ to 1000, for example. The function *shortestPath* (Program 6.9) contains our implementation of Dijkstra's algorithm. This function uses *choose* (Program 6.10) to return a vertex, u, such that u has the minimum distance from the start vertex, v.

```
void shortestPath(int v, int cost[][MAX_VERTICES],
                  int distance[], int n, short int found[])
{/* distance[i] represents the shortest path from vertex v
    to i, found[i] is 0 if the shortest path from i
    has not been found and a 1 if it has, cost is the
    adjacency matrix */
  int i,u,w;
  for (i = 0; i < n; i++) {
    found[i] = FALSE;
    distance[i] = cost[v][i];
  }
  found[v] = TRUE;
  distance[v] = 0;
  for (i = 0; i < n-2; i++) {
    u = choose(distance,n,found);
    found[u] = TRUE;
    for (w = 0; w < n; w++)
      if (!found[w])
        if (distance[u] + cost[u][w] < distance[w])
          distance[w] = distance[u] + cost[u][w];
  }
}
```

Program 6.9: Single source shortest paths

Analysis of *shortestpath*: The time taken by the algorithm on a graph with n vertices is $O(n^2)$. To see this, note that the first **for** loop takes $O(n)$ time. The second **for** loop is executed $n - 2$ times. Each execution of this loop requires $O(n)$ time to select the next vertex and also to update *dist*. So the total time for this loop is $O(n^2)$. 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 is $O(e)$. Since we represented the graph as a cost adjacency matrix, it takes $O(n^2)$ time just to determine the edges that are in G. Therefore, any shortest path algorithm using this representation has a time complexity of $O(n^2)$. The exercises explore several variations that speed up the algorithm, but the asymptotic time complexity remains $O(n^2)$. For the case of graphs with few edges, the use of Fibonacci heaps together with an adjacency list representation produces a more efficient implementation of the greedy algorithm for the single-source all-destinations problem. We discuss this in Chapter 9. □

Example 6.4: Consider the eight-vertex digraph of Figure 6.27(a) with length-

```
int choose(int distance[], int n, short int found[])
{/* find the smallest distance not yet checked */
   int i, min, minpos;
   min = INT_MAX;
   minpos = -1;
   for (i = 0; i < n; i++)
      if (distance[i] < min && !found[i]) {
         min = distance[i];
         minpos = i;
      }
   return minpos;
}
```

Program 6.10: Choosing the least cost edge

adjacency matrix as in Figure 6.27(b). Suppose that the source vertex is Boston. The values of *dist* and the vertex *u* selected in each iteration of the outer **for** loop of Program 6.9 are shown in Figure 6.28. We use ∞ to denote the value *LARGE*. Note that the algorithm terminates after only 6 iterations of the **for** loop. 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. □

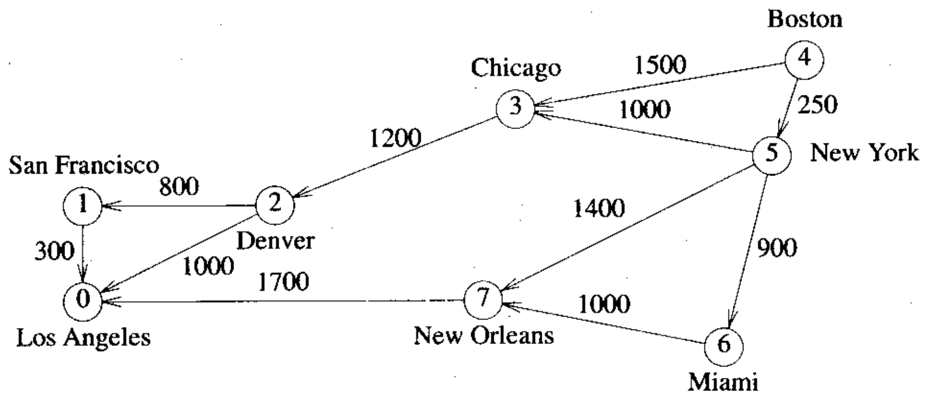
6.4.2 Single Source/All Destinations: General Weights

We now consider the general case when some or all of the edges of the directed graph *G* may have negative length. To see that function *shortestpath* (Program 6.9) does not necessarily give the correct results on such graphs, consider the graph of Figure 6.29. Let $v = 0$ be the source vertex. Since $n = 3$, the loop of lines 7 to 14 is iterated just once; $u = 2$ in line 8, and no changes are made to *dist*. The function terminates with *dist* [1] = 7 and *dist* [2] = 5. The shortest path from 0 to 2 is 0, 1, 2. This path has length 2, which is less than the computed value of *dist* [2].

When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary so as to ensure that shortest paths consist of a finite number of edges. For example, consider the graph of Figure 6.30. The length of the shortest path from vertex 0 to vertex 2 is $-\infty$, as the length of the path

$$0, 1, 0, 1, 0, 1, \cdots, 0, 1, 2$$

can be made arbitrarily small. This is so because of the presence of the cycle 0, 1, 0, which has a length of -1.



(a) Digraph

$$
\begin{array}{c|cccccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
0 & 0 & & & & & & & \\
1 & 300 & 0 & & & & & & \\
2 & 1000 & 800 & 0 & & & & & \\
3 & & & 1200 & 0 & & & & \\
4 & & & & 1500 & 0 & 250 & & \\
5 & & & & 1000 & & 0 & 900 & 1400 \\
6 & & & & & & & 0 & 1000 \\
7 & 1700 & & & & & & & 0
\end{array}
$$

(b) Length-adjacency matrix

Figure 6.27: Digraph for Example 6.4

When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most $n - 1$ edges on it. To see this, observe that a path that has more than $n - 1$ edges must repeat at least one vertex and hence must contain a cycle. Elimination of the cycles from the path results in another path with the same source and destination. This path is cycle-free and has a length that is no more than that of the original path, as the length of the eliminated cycles was at least zero. We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph. As in the case of function *shortestPath* (Program 6.9), we shall compute only the length, *dist* [u], of the shortest path from the source vertex v to u. An exercise examines the extension needed to construct the shortest paths.

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

Figure 6.28: Action of *shortestPath* on digraph of Figure 6.27

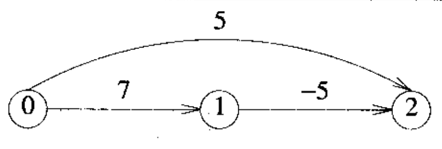


Figure 6.29: Directed graph with a negative-length edge

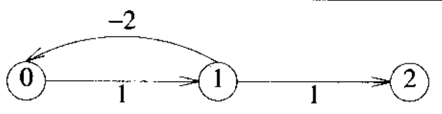


Figure 6.30: Directed graph with a cycle of negative length

Let $dist^l[u]$ be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most l edges. Then, $dist^1[u] = length[v][u]$, $0 \le u < n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most $n-1$ edges. Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u.

Our goal then is to compute $dist^{n-1}[u]$ for all u. This can be done using the dynamic programming methodology. First, we make the following observations:

(1) If the shortest path from v to u with at most k, $k > 1$, edges has no more than $k-1$ edges, then $dist^k[u] = dist^{k-1}[u]$.

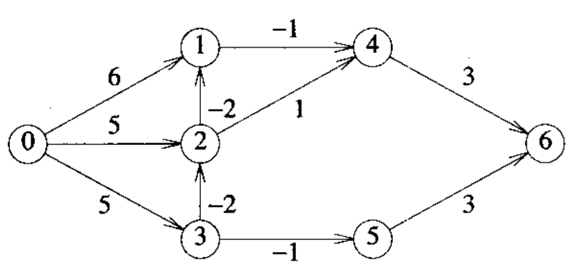
(2) If the shortest path from v to u with at most k, $k > 1$, edges has exactly k edges, then it is comprised of a shortest path from v to some vertex j followed by the edge $<j,u>$. The path from v to j has $k-1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $<i,u>$ is in the graph are candidates for j. Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + length[i][u]$ is the correct value for j.

These observations result in the following recurrence for *dist*:

$$dist^k[u] = \min\{dist^{k-1}[u], \min_i\{dist^{k-1}[i] + length[i][u]\}\}$$

This recurrence may be used to compute $dist^k$ from $dist^{k-1}$, for $k = 2, 3, \cdots, n-1$.

Example 6.5: Figure 6.31 gives a seven-vertex graph, together with the arrays $dist^k$, $k = 1, \cdots, 6$. These arrays were computed using the equation just given. □



(a) A directed graph

	$dist^k[7]$						
k	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Figure 6.31: Shortest paths with negative edge lengths

An exercise shows that if we use the same memory location $dist[u]$ for $dist^k[u]$, $k = 1, \cdots, n-1$, then the final value of $dist[u]$ is still $dist^{n-1}[u]$. Using this fact and the recurrence for *dist* shown above, we arrive at the algorithm of Program 6.11 to compute the length of the shortest path from vertex v to each other vertex of the graph. This algorithm is referred to as the Bellman and Ford algorithm.

```
 1 void BellmanFord(int n, int v)
 2 {/* Single source all destination shortest paths
 3     with negative edge lengths. */
 4    for (int i = 0; i < n; i++)
 5       dist[i] = length[v][i]; /* initialize dist */

 6    for (int k = 2; k <= n-1; k++)
 7       for (each u such that u != v and u
 8            has at least one incoming edge)
 9          for (each <i, u> in the graph)
10             if (dist[u] > dist[i] + length[i][u])
11                dist[u] = dist[i] + length[i][u];
12 }
```

Program 6.11: Bellman and Ford algorithm to compute shortest paths

Analysis of *BellmanFord:* Each iteration of the **for** loop of lines 6 to 11 takes $O(n^2)$ time if adjacency matrices are used and $O(e)$ time if adjacency lists are used. The overall complexity is $O(n^3)$ when adjacency matrices are used and $O(ne)$ when adjacency lists are used. The observed complexity of the shortest-path algorithm can be reduced by noting that if none of the *dist* values change on one iteration of the **for** loop of lines 6 to 11, then none will change on successive iterations. So, this loop may be rewritten to terminate either after $n-1$ iterations or after the first iteration in which no *dist* values are changed, whichever occurs first. Another possibility is to maintain a queue of vertices i whose *dist* value changed on the previous iteration of the **for** loop. These are the only values for i that need to be considered in line 9 during the next iteration. When a queue of these values is maintained, we can rewrite the loop of lines 6 to 11 so that on each iteration, a vertex i is removed from the queue, and the *dist* values of all vertices adjacent from i are updated as in lines 10 and 11. Vertices whose *dist* value decreases as a result of this are added to the end of the queue unless they are already on it. The loop terminates when the queue becomes empty. □

6.4.3 All Pairs Shortest Paths

In the all-pairs-shortest-path problem we must find the shortest paths between all pairs of vertices, v_i, v_j, $i \neq j$. We could solve this problem using *shortestpath* with each of the vertices in $V(G)$ as the source. Since G has n vertices and *shortestpath* has a time complexity of $O(n^2)$, the total time required would be $O(n^3)$. However, we can obtain a conceptually simpler algorithm that works correctly even if some edges in G have negative weights. (We do require that G has no cycles with a negative length.) Although this algorithm still has a computing time of $O(n^3)$, it has a smaller constant factor. This new algorithm uses the dynamic programming method.

We represent the graph G by its cost adjacency matrix with *cost*$[i][j] = 0$, $i = j$. If the edge $<i, j>$, $i \neq j$ is not in G, we set *cost*$[i][j]$ to some sufficiently large number using the same restrictions discussed in the single source problem. Let $A^k[i][j]$ be the cost of the shortest path from i to j, using only those intermediate vertices with an index $\leq k$. The cost of the shortest path from i to j is $A^{n-1}[i][j]$ as no vertex in G has an index greater than $n-1$. Further, $A^{-1}[i][j] = cost[i][j]$ since the only i to j paths allowed have no intermediate vertices on them.

The basic idea in the all pairs algorithm is to begin with the matrix A^{-1} and successively generate the matrices $A^0, A^1, A^2, \cdots, A^{n-1}$. If we have already generated A^{k-1}, then we may generate A^k by realizing that for any pair of vertices i, j one of the two rules below applies.

(1) The shortest path from i to j going through no vertex with index greater than k does not go through the vertex with index k and so its cost is $A^{k-1}[i][j]$.

(2) The shortest such path does go through vertex k. Such a path consists of a path from i to k followed by one from k to j. Neither of these goes through a vertex with index greater than $k-1$. Hence, their costs are $A^{k-1}[i][k]$ and $A^{k-1}[k][j]$.

These rules yield the following formulas for $A^k[i][j]$:

$$A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$$

and

$$A^{-1}[i][j] = cost[i][j]$$

Example 6.6: Figure 6.32 shows a digraph together with its A^{-1} matrix. For this graph $A^1[0][2] \neq \min\{A^1[0][2], A^0[0][1] + A^0[1][2]\} = 2$. Instead, $A^1[0][2] = -\infty$ because the length of the path:

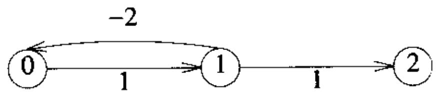
$$0, 1, 0, 1, 0, 1, \cdots, 0, 1, 2$$

can be made arbitrarily small. This situation occurs because we have a cycle, 0, 1, 0, that has a negative length (-1). □

The function *allCosts* (Program 6.12) computes $A^{n-1}[i][j]$. The computations are done in place using the array *distance*, which we define as:

```
int distance[MAX_VERTICES][MAX_VERTICES];
```

The reason this computation can be carried out in place is that $A^k[i,k] = A^{k-1}[i,k]$ and $A^k[k,j] = A^{k-1}[k,j]$ and so the in place computation does not alter the outcome.



(a) Directed graph

$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

(b) A^{-1}

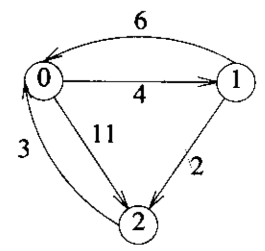
Figure 6.32: Graph with negative cycle

```
void allCosts(int cost[][MAX_VERTICES],
                  int distance[][MAX_VERTICES], int n)
{/* compute the shortest distance from each vertex
    to every other, cost is the adjacency matrix,
    distance is the matrix of computed  distances */
   int i,j,k;
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         distance[i][j] = cost[i][j];
   for (k = 0; k < n; k++)
      for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
            if (distance[i][k] + distance[k][j] <
                                          distance[i][j])
               distance[i][j] =
               distance[i][k] + distance[k][j];
}
```

Program 6.12: All pairs, shortest paths function

Analysis of *allCosts*: This algorithm is especially easy to analyze because the looping is independent of the data in the distance matrix. The total time for *allCosts* is $O(n^3)$. An exercise examines the extensions needed to generate the $<i, j>$ paths with these lengths. We can speed up the algorithm by using our knowledge of the fact that the innermost **for** loop is executed only when *distance*[i][k] and *distance*[k][j] are not equal to ∞. □

Example 6.7: For the digraph of Figure 6.33(a), the initial a matrix, A^{-1}, plus its value after each of three iterations, A^0, A^1, and A^2, is also given in Figure 6.33. □



(a) Example digraph

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(b) A^{-1}

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

(c) A^{0}

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

(d) A^{1}

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

(e) A^{2}

Figure 6.33: Example for all-pairs shortest-paths problem

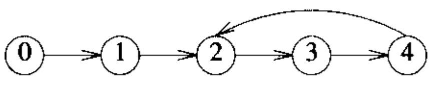
6.4.4 Transitive Closure

We end this section by studying a problem that is closely related to the all pairs, shortest path problem. Assume that we have a directed graph G with unweighted edges. We want to determine if there is a path from i to j for all values of i and j. Two cases are of interest. The first case requires positive path lengths, while the second requires only nonnegative path lengths. These cases are known as the *transitive closure* and *reflexive transitive closure* of a graph, respectively. We define them as follows:

Definition: The *transitive closure matrix*, denoted A^+, of a directed graph, G, is a matrix such that $A^+[i][j] = 1$ if there is a path of length > 0 from i to j; otherwise, $A^+[i][j] = 0$. □

Definition: The *reflexive transitive closure matrix*, denoted A^*, of a directed graph, G, is a matrix such that $A^*[i][j] = 1$ if there is a path of length ≥ 0 from i to j; otherwise, $A^*[i][j] = 0$. □

Figure 6.34 shows A^+ and A^* for a digraph. Clearly, the only difference between A^* and A^+ is in the terms on the diagonal. $A^+[i][i] = 1$ iff there is a cycle of length >1 containing vertex i, whereas $A^*[i][i]$ is always one, as there is a path of length 0 from i to i.



(a) Digraph G

$$\begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 & 0 \end{array}$$

(b) Adjacency matrix A

$$\begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 & 1 \\ 3 & 0 & 0 & 1 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 & 1 \end{array}$$

(c) A^+

$$\begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 & 1 \\ 3 & 0 & 0 & 1 & 1 & 1 \\ 4 & 0 & 0 & 1 & 1 & 1 \end{array}$$

(d) A^*

Figure 6.34: Graph G and its adjacency matrix A, A^+, and A^*

We can use *allCosts* to compute A^+. We begin with $cost[i][j] = 1$ if $<i, j>$ is an edge in G and $cost[i][j] = +\infty$ if $<i, j>$ is not in G. When *allCosts* terminates, we obtain A^+ from *distance* by letting $A^+[i][j] = 1$ *iff distance[i][j]* $< +\infty$. We then obtain A^* by setting all the diagonal elements in A^+ to 1. The total time is $O(n^3)$. We can simplify the algorithm by changing the **if** statement in the nested **for** loops to:

```
distance[i][j] = distance[i][j] || distance[i][k] &&
                    distance[k][j]
```

and initializing *distance* to be the adjacency matrix of the graph. With this modification, *distance* will be equivalent to A^+ when *allCosts* terminates.

The transitive closure of an undirected graph G can be found more easily from its connected components. From the definition of a connected component, it follows that there is a path between every pair of vertices in the component and there is no path in G between two vertices that are in different components. Hence, if A is the adjacency matrix of an undirected graph (i.e., A is symmetric) then its transitive closure A^+ may be determined in $O(n^2)$ time by first determining the connected components of the graph. $A^+[i][j] = 1$ iff there is a path from vertex i to j. For every pair of distinct vertices in the same component, $A^+[i][j] = 1$. On the diagonal, $A^+[i][i] = 1$ iff the component containing i has at least two vertices.

EXERCISES

1. Let T be a tree with root v. The edges of T are undirected. Each edge in T has a nonnegative length. Write a C function to determine the length of the shortest paths from v to the remaining vertices of T. Your function should have complexity $O(n)$, where n is the number of vertices in T. Show that this is the case.
2. Let G be a directed, acyclic graph with n vertices. Assume that the vertices are numbered 0 through $n-1$ and that all edges are of the form $<i,j>$, where $i < j$. Assume that the graph is available as a set of adjacency lists and that each edge has a length (which may be negative) associated with it. Write a C++ function to determine the length of the shortest paths from vertex 0 to the remaining vertices. The complexity of your algorithm should be $O(n + e)$, where e is the number of edges in the graph. Show that this is the case.
3. (a) Do the previous exercise, but this time find the length of the longest paths instead of the shortest paths.

   (b) Extend your algorithm of (a) to determine a longest path from vertex 0 to each of the remaining vertices.
4. What is a suitable value for *LARGE* in the context of function *shortestpath* (Program 6.9)? Provide this as a function of the largest edge length *maxL* and the number of vertices n.
5. Using the idea of *shortestpath* (Program 6.9), write a C++ function to find a minimum-cost spanning tree whose worst-case time is $O(n^2)$.
6. Use *shortestpath* (Program 6.9) to obtain, in nondecreasing order, the lengths of the shortest paths from vertex 0 to all remaining vertices in the digraph of Figure 6.35.
7. Rewrite *shortestpath* (Program 6.9) under the following assumptions:

   (a) G is represented by its adjacency lists, where each node has three fields: *vertex*, *length*, and *link*. *length* is the length of the corresponding edge and n the number of vertices in G.

   (b) Instead of 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.

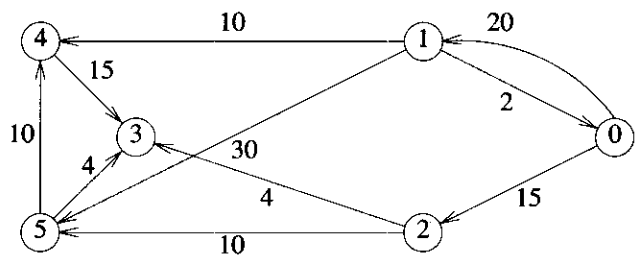


Figure 6.35: A digraph

What can you say about the computing time of your new function relative to that of *shortestpath*?

8. Modify *shortestpath* (Program 6.9) so that it obtains the shortest paths, in addition to the lengths of these paths. What is the computing time of your modified function?

9. Using the directed graph of Figure 6.36, explain why *shortestpath* will not work properly. What is the shortest path between vertices 0 and 6?

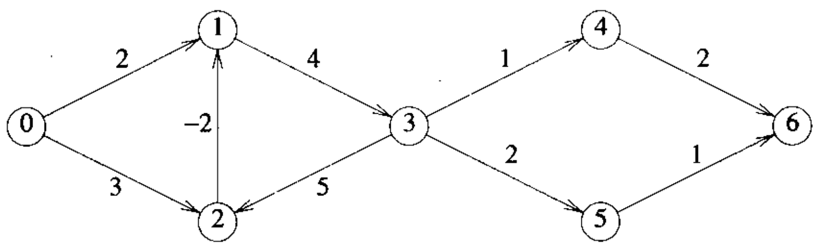


Figure 6.36: Directed graph on which *ShortestPath* does not work properly

10. Prove the correctness of function *BellmanFord* (Program 6.11). Note that this function does not faithfully implement the computation of the recurrence for $dist^k$. In fact, for $k < n - 1$, the *dist* values following iteration k of the **for** loop of lines 4 to 7 may not be $dist^k$.

11. Transform function *BellmanFord* into a complete C function. Assume that graphs are represented using adjacency lists in which each node has an additional field called *length* that gives the length of the edge represented by that node. As a result of this, there is no length-adjacency matrix. Generate some test graphs and test the correctness of your function.

12. Rewrite function *BellmanFord* so that the loop of lines 4 to 7 terminates either after $n - 1$ iterations or after the first iteration in which no *dist* values are changed, whichever occurs first.

13. Rewrite function *BellmanFord* by replacing the loop of lines 4 to 7 with code that uses a queue of vertices that may potentially result in a reduction of other *dist* vertices. This queue initially contains all vertices that are adjacent from the source vertex v. On each successive iteration of the new loop, a vertex i is removed from the queue (unless the queue is empty), and the *dist* values to vertices adjacent from i are updated as in line 7 of Program 6.11. When the *dist* value of a vertex is reduced because of this, it is added to the queue unless it is already on the queue.

    (a) Prove that the new function produces the same results as the original one.

    (b) Show that the complexity of the new function is no more than that of the original one.

14. Compare the run-time performance of the Bellman and Ford functions of the preceding two exercises and that of Program 6.11. For this, generate test graphs that will expose the relative performance of the three functions.

15. Modify function *BellmanFord* so that it obtains the shortest paths, in addition to the lengths of these paths. What is the computing time of your function?

16. What is a suitable value for *LARGE* in the context of function *allCosts* (Program 6.12)? Provide this as a function of the largest edge length *maxL* and the number of vertices n.

17. Modify function *allCosts* (Program 6.12) so that it obtains a shortest path for all pairs of vertices. What is the computing time of your new function?

18. Use function *allCosts* to obtain the lengths of the shortest paths between all pairs of vertices in the graph of Figure 6.35. Does *allCosts* give the right answers? Why?

19. By considering the complete graph with n vertices, show that the maximum number of simple paths between two vertices is $O((n-1)!)$.

20. Show that $A^+ = A^* \times A$, where matrix multiplication of the two matrices is defined as $a_{ij}^+ = \vee_{k=1}^{n} a_{ik}^* \wedge a_{kj}$. $\vee$ is the logical **or** operation, and $\wedge$ is the logical **and** operation.

21. Obtain the matrices A^+ and A^* for the digraph of Figure 6.15.

22. What is a suitable value for *LARGE* when *allCosts* (Program 6.12) is used to compute the transitive closure of a directed graph? Provide this as a function of the number of vertices *n*.

6.5 ACTIVITY NETWORKS

6.5.1 Activity-on-Vertex (AOV) Networks

All but the simplest of projects can be subdivided into several subprojects called activities. The successful completion of these activities results in the completion of the entire project. A student working toward a degree in computer science must complete several courses successfully. The project in this case is to complete the major, and the activities are the individual courses that have to be taken. Figure 6.37 lists the courses needed for a computer science major at a hypothetical university. Some of these courses may be taken independently of others; other courses have prerequisites and can be taken only if all the prerequisites have already been taken. The data structures course cannot be started until certain programming and math courses have been completed. Thus, prerequisites define precedence relations between courses. The relationships defined may be more clearly represented using a directed graph in which the vertices represent courses and the directed edges represent prerequisites.

Definition: A directed graph *G* in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an *activity-on-vertex network* or AOV network. □

Figure 6.37(b) is the AOV network corresponding to the courses of Figure 6.37(a). Each edge $<i, j>$ implies that course *i* is a prerequisite of course *j*.

Definition: Vertex *i* in an AOV network *G* is a *predecessor* of vertex *j* iff there is a directed path from vertex *i* to vertex *j*. *i* is an *immediate predecessor* of *j* iff $<i,j>$ is an edge in *G*. If *i* is a predecessor of *j*, then *j* is a *successor* of *i*. If *i* is an immediate predecessor of *j*, then *j* is an *immediate successor* of *i*. □

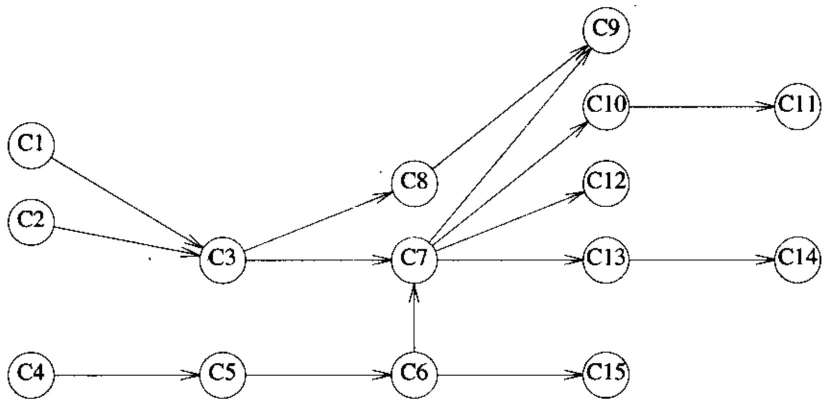
C3 and C6 are immediate predecessors of C7. C9, C10, C12, and C13 are immediate successors of C7. C14 is a successor, but not an immediate successor, of C3.

Definition: A relation $\cdot$ is *transitive* iff it is the case that for all triples i,j,k, $i \cdot j$ and $j \cdot k \Rightarrow i \cdot k$. A relation $\cdot$ is *irreflexive* on a set *S* if for no element *x* in *S* is it the case that $x \cdot x$. A precedence relation that is both transitive and irreflexive is a *partial order*. □

Notice that the precedence relation defined by course prerequisites is transitive.

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university



(b) AOV network representing courses as vertices and prerequisites as edges

Figure 6.37: An activity-on-vertex (AOV) network

That is, if course i must be taken before course j (as i is a prerequiste of j), and if j must be taken before k, then i must be taken before k. This fact is not obvious from the AOV network. For example, <C4, C5> and <C5, C6> are edges in the AOV network of Figure 6.37(b). However, <C4, C6> is not. Generally, AOV networks are incompletely specified, and the edges needed to make the precedence relation transitive are implied.

If the precedence relation defined by the edges of an AOV network is not irreflexive, then there is an activity that is a predecessor of itself and so must be completed before it can be started. This is clearly impossible. When there are no inconsistencies of this type, the project is feasible. Given an AOV network, one of our concerns would be to determine whether or not the precedence relation defined by its edges is irreflexive. This is identical to determining whether or not the network contains any directed cycles. A directed graph with no directed cycles is an *acyclic* graph. Our algorithm to test an AOV network for feasibility will also generate a linear ordering, $v_0, v_1, \cdots, v_{n-1}$, of the vertices (activities). This linear ordering will have the property that if vertex i is a predecessor of j in the network, then i precedes j in the linear ordering. A linear ordering with this property is called a *topological order.*

Definition: A *topological order* is a linear ordering of the vertices of a graph such that, for any two vertices i and j, if i is a predecessor of j in the network, then i precedes j in the linear ordering. □

There are several possible topological orders for the network of Figure 6.37(b). Two of these are

C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

and

C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14

If a student were taking just one course per term, then she or he would have to take them in topological order. If the AOV network represented the different tasks involved in assembling an automobile, then these tasks would be carried out in topological order on an assembly line. The algorithm to sort the tasks into topological order is straightforward and proceeds by listing a vertex in the network that has no predecessor. Then, this vertex together with all edges leading out from it is deleted from the network. These two steps are repeated until all vertices have been listed or all remaining vertices in the network have predecessors, and so none can be removed. In this case there is a cycle in the network, and the project is infeasible. The algorithm is stated more formally in Program 6.13.

Example 6.8: Let us try out our topological sorting algorithm on the network of Figure 6.38(a). The first vertex to be picked in line 6 is 0, as it is the only one with no predecessors. Vertex 0 and the edges <0, 1>, <0, 2>, and <0, 3> are deleted. In the resulting network (Figure 6.38(b)), vertices 1, 2, and 3 have no predecessor. Any of these can be the next vertex in the topological order. Assume that 3 is picked. Deletion of vertex 3 and

```
Input the AOV network.  Let n be the number of vertices.
for (i = 0; i < n; i++) /* output the vertices */
{
  if (every vertex has a predecessor) return;
     /* network has a cycle and is infeasible */
  pick a vertex v that has no predecessors;
  output v;
  delete v and all edges leading out of v;
}
```

Program 6.13: Design of an algorithm for topological sorting

the edges <3, 5> and <3, 4> results in the network of Figure 6.38(c). Either 1 or 2 may be picked next. Figure 6.38 shows the progress of the algorithm on the network. □

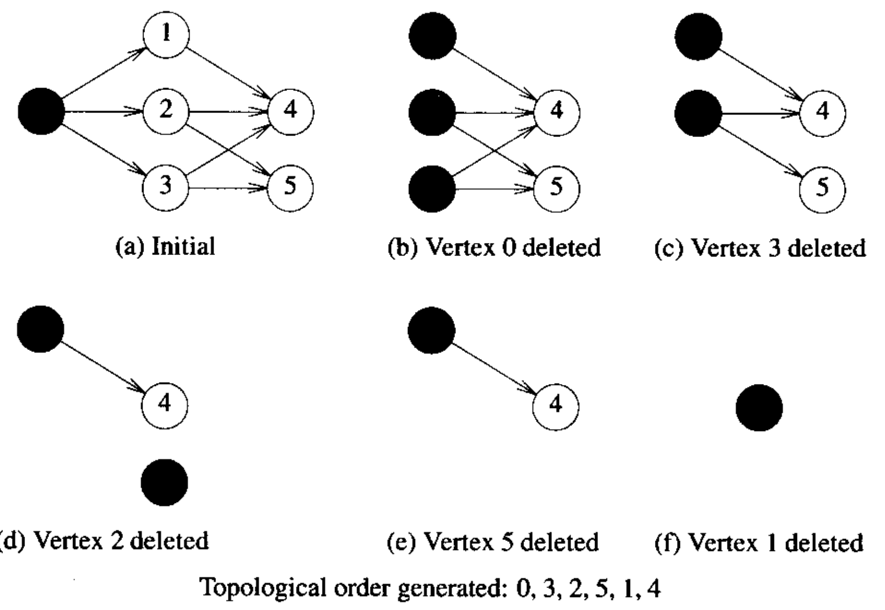


Figure 6.38: Action of Program 6.13 on an AOV network (shaded vertices represent candidates for deletion)

To obtain a complete algorithm that can be easily translated into a computer program, it is necessary to specify the data representation for the AOV network. The choice of a data representation, as always, depends on the functions you wish to perform. In this problem, the functions are

(1) decide whether a vertex has any predecessors (line 4)

(2) delete a vertex together with all its incident edges (line 8)

To perform the first task efficiently, we maintain a count of the number of immediate predecessors each vertex has. The second task is easily implemented if the network is represented by its adjacency lists. Then the deletion of all edges leading out of vertex *v* can be carried out by decreasing the predecessor count of all vertices on its adjacency list. Whenever the count of a vertex drops to zero, that vertex can be placed onto a list of vertices with a zero count. Then the selection in line 6 just requires removal of a vertex from this list.

As a result of the preceding analysis, we represent the AOV network using adjacency lists. The complete C function for performing a topological sort on a network is *topSort* (Program 6.14). This function assumes that the network is represented by its adjacency lists. The header nodes of these lists now contain count and link fields.

The declarations used in *topSort* are:

```
typedef struct node *nodePointer;
typedef struct {
        int vertex;
        nodePointer link;
        } node;
typedef struct {
        int count;
        nodePointer link;
        } hdnodes;
hdnodes graph[MAX_VERTICES];
```

The *count* field contains the in-degree of that vertex and *link* is a pointer to the first node on the adjacency list. Each node has two fields, *vertex* and *link*. This can be done easily at the time of input. When edge <*i*,*j*> is input, the count of vertex *j* is incremented by 1. Figure 6.39(a) shows the internal representation of the network of Figure 6.38(a).

Inserting these details into Program 6.13, we obtain the C function *topSort* (Program 6.14). The list of vertices with zero count is maintained as a custom stack. A queue could have been used instead, but a stack is slightly simpler. The stack is linked through the *count* field of the header nodes, since this field is of no use after a vertex's

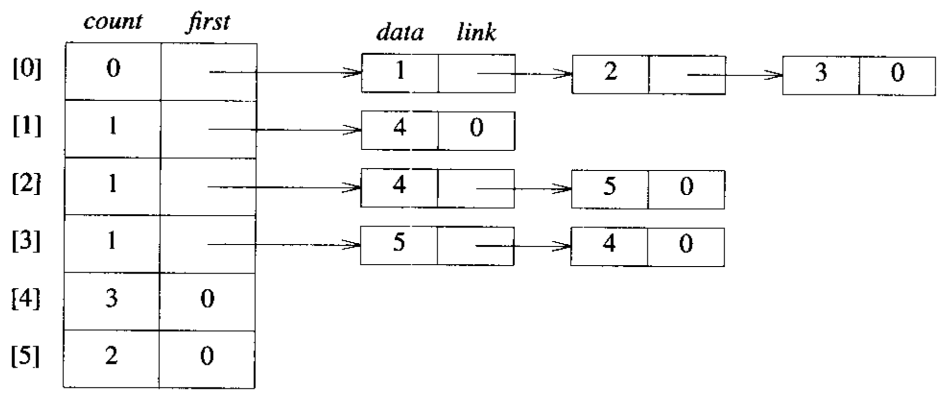


Figure 6.39: Internal representation used by topological sorting algorithm

count has become zero.

Analysis of *topSort*: As a result of a judicious choice of data structures, *topSort* is very efficient. The first **for** loop takes O(n) time, on a network with n vertices and e edges. The second **for** loop is iterated n times. The **if** clause is executed in constant time; the **for** loop within the **else** clause takes time O(d_i), where d_i is the out-degree of vertex i. Since this loop is encountered once for each vertex that is printed, the total time for this part of the algorithm is:

$$O\left(\left(\sum_{i=0}^{n-1} d_i \right) + n \right) = O(e + n)$$

Thus, the asymptotic computing time of the algorithm is O(e + n). It is linear in the size of the problem! □

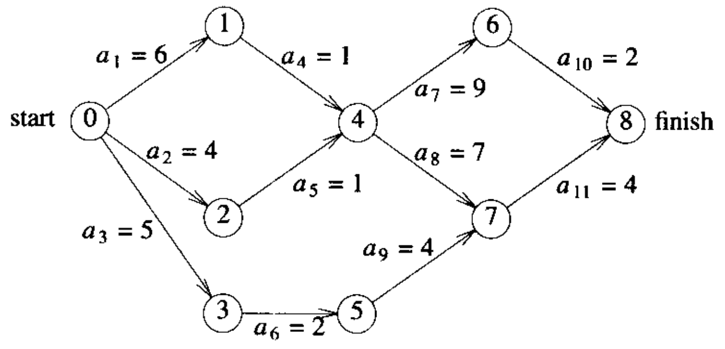
6.5.2 Activity-on-Edge (AOE) Networks

An activity network closely related to the AOV network is the *activity-on-edge*, or *AOE, network*. The tasks to be performed on a project are represented by directed edges. Vertices in the network represent events. Events signal the completion of certain activities. Activities represented by edges leaving a vertex cannot be started until the event at that vertex has occurred. An event occurs only when all activities entering it have been

```
void topSort(hdnodes graph[], int n)
{
  int i,j,k,top;
  nodePointer ptr;
  /* create a stack of vertices with no predecessors */
  top = -1;
  for (i = 0; i < n; i++)
    if (!graph[i].count) {
      graph[i].count = top;
      top = i;
  }
  for (i = 0; i < n; i++)
    if (top == -1) {
      fprintf(stderr,
        "\nNetwork has a cycle. Sort terminated. \n");
      exit(EXIT_FAILURE);
    }
    else {
      j = top;   /* unstack a vertex */
      top = graph[top].count;
      printf("v%d, ",j);
      for (ptr = graph[j].link; ptr; ptr = ptr→link) {
      /* decrease the count of the successor vertices
        of j */
        k = ptr→vertex;
        graph[k].count--;
        if (!graph[k].count) {
        /* add vertex k to the stack */
          graph[k].count = top;
          top = k;
        }
      }
    }
}
```

Program 6.14: Topological sort

completed. Figure 6.40(a) is an AOE network for a hypothetical project with 11 tasks or activities: $a_1, \cdots, a_{11}$. There are nine events: 0, 1, $\cdots$, 8. The events 0 and 8 may be interpreted as "start project" and "finish project," respectively. Figure 6.40(b) gives interpretations for some of the nine events. The number associated with each activity is the time needed to perform that activity. Thus, activity a_1 requires 6 days, whereas a_{11} requires 4 days. Usually, these times are only estimates. Activities a_1, a_2, and a_3 may be carried out concurrently after the start of the project. Activities a_4, a_5, and a_6 cannot be started until events 1, 2, and 3, respectively, occur. Activities a_7 and a_8 can be carried out concurrently after the occurrence of event 4 (i.e., after a_4 and a_5 have been completed). If additional ordering constraints are to be put on the activities, dummy activities whose time is zero may be introduced. Thus, if we desire that activities a_7 and a_8 not start until both events 4 and 5 have occurred, a dummy activity a_{12} represented by an edge <5,4> may be introduced.



(a) Activity network of a hypothetical project

event	*interpretation*
0	start of project
1	completion of activity a_1
4	completion of activities a_4 and a_5
7	completion of activities a_8 and a_9
8	completion of project

(b) Interpretation of some of the events in the network of (a)

Figure 6.40: An AOE network

Activity networks of the AOE type have proved very useful in the performance evaluation of several types of projects. This evaluation includes determining such facts about the project as what is the least amount of time in which the project may be completed (assuming there are no cycles in the network), which activities should be speeded to reduce project length, and so on.

Since the activities in an AOE network can be carried out in parallel, the minimum time to complete the project is the length of the longest path from the start vertex to the finish vertex (the length of a path is the sum of the times of activities on this path). A path of longest length is a *critical path*. The path 0, 1, 4, 6, 8 is a critical path in the network of Figure 6.40(a). The length of this critical path is 18. A network may have more than one critical path (the path 0, 1, 4, 7, 8 is also critical).

The *earliest time* that an event i can occur is the length of the longest path from the start vertex 0 to the vertex i. The earliest time that event v_4 can occur is 7. The earliest time an event can occur determines the *earliest start time* for all activities represented by edges leaving that vertex. Denote this time by $e(i)$ for activity a_i. For example, $e(7)=e(8)=7$.

For every activity a_i, we may also define the *latest time*, $l(i)$, that an activity may start without increasing the project duration (i.e., length of the longest path from start to finish). In Figure 6.40(a) we have $e(6)=5$ and $l(6)=8$, $e(8)=7$ and $l(8)=7$.

All activities for which $e(i)=l(i)$ are called *critical activities*. The difference $l(i)-e(i)$ is a measure of the criticality of an activity. It gives the time by which an activity may be delayed or slowed without increasing the total time needed to finish the project. If activity a_6 is slowed down to take 2 extra days, this will not affect the project finish time. Clearly, all activities on a critical path are strategic, and speeding up noncritical activities will not reduce the project duration.

The purpose of critical-path analysis is to identify critical activities so that resources may be concentrated on these activities in an attempt to reduce project finish time. Speeding a critical activity will not result in a reduced project length unless that activity is on all critical paths. In Figure 6.40(a) the activity a_{11} is critical, but speeding it up so that it takes only 3 days instead of 4 does not reduce the finish time to 17 days. This is so because there is another critical path (0, 1, 4, 6, 8) that does not contain this activity. The activities a_1 and a_4 are on all critical paths. Speeding a_1 by 2 days reduces the critical path length to 16 days. Critical-path methods have proved very valuable in evaluating project performance and identifying bottlenecks.

Critical-path analysis can also be carried out with AOV networks. The length of a path would now be the sum of the activity times of the vertices on that path. By analogy, for each activity or vertex we could define the quantities $e(i)$ and $l(i)$. Since the activity times are only estimates, it is necessary to reevaluate the project during several stages of its completion as more accurate estimates of activity times become available. These changes in activity times could make previously noncritical activities critical, and vice versa.

Before ending our discussion on activity networks, let us design an algorithm to calculate $e(i)$ and $l(i)$ for all activities in an AOE network. Once these quantities are known, then the critical activities may easily be identified. Deleting all noncritical activities from the AOE network, all critical paths may be found by just generating all paths from the start-to-finish vertex (all such paths will include only critical activities and so must be critical, and since no noncritical activity can be on a critical path, the network with noncritical activities removed contains all critical paths present in the original network).

6.5.2.1 Calculation of Early Activity Times

When computing the early and late activity times, it is easiest first to obtain the earliest event time, $ee[j]$, and latest event time, $le[j]$, for all events, j, in the network. Thus if activity a_i is represented by edge $<k,l>$, we can compute $e(i)$ and $l(i)$ from the following formulas:

$$e(i)=ee[k]$$

and (6.1)

$$l(i)=le[l]-\text{duration of activity } a_i$$

The times $ee[j]$ and $le[j]$ are computed in two stages: a forward stage and a backward stage. During the forward stage we start with $ee[0]=0$ and compute the remaining early start times, using the formula

$$ee[j]=\max_{i\in P(j)}\{ee[i]+\text{duration of} <i,j>\} \qquad (6.2)$$

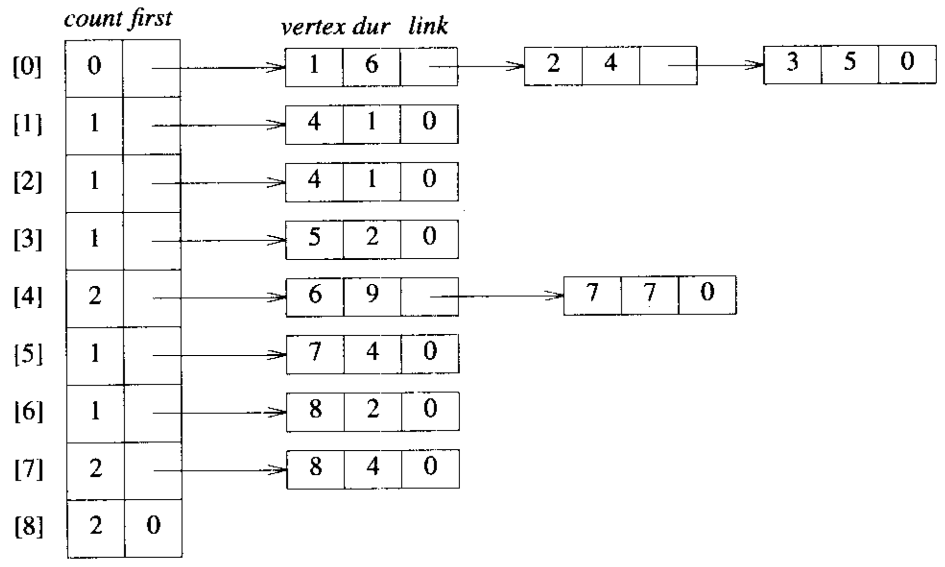
where $P(j)$ is the set of all vertices adjacent to vertex j. If this computation is carried out in topological order, the early start times of all predecessors of j would have been computed prior to the computation of $ee[j]$. So, if we modify *topSort* (Program 6.14) so that it returns the vertices in topological order (rather than outputs them in this order), then we may use this topological order and Eq. 6.2 to compute the early event times. To use Eq. 6.2, however, we must have easy access to the vertex set $P(j)$. Since the adjacency list representation does not provide easy access to $P(j)$, we make a more major modification to Program 6.14. We begin with the *ee* array initialized to zero and insert the code

```
if (earliest[k] < earliest[j] + ptr→duration)
    earliest[k] = earliest[j] + ptr→duration;
```

just after the line

```
k = ptr→vertex;
```

This modification results in the evaluation of Eq. (6.2) in parallel with the generation of a topological order. $ee(k)$ is updated each time the $ee()$ of one of its predecessors is known (i.e., when j is ready for output).



(a) Adjacency lists for Figure 6.40(a)

ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	0	0	0	0	0	0	0	0	0	[0]
output 0	0	6	4	5	0	0	0	0	0	[3, 2, 1]
output 3	0	6	4	5	0	7	0	0	0	[5, 2, 1]
output 5	0	6	4	5	0	7	0	11	0	[2, 1]
output 2	0	6	4	5	5	7	0	11	0	[1]
output 1	0	6	4	5	7	7	0	11	0	[4]
output 4	0	6	4	5	7	7	16	14	0	[7, 6]
output 7	0	6	4	5	7	7	16	14	18	[6]
output 6	0	6	4	5	7	7	16	14	18	[8]
output 8										

(b) Computation of *ee*

Figure 6.41: Computing *ee* using modified *topSort* (Program 6.14)

To illustrate the working of the modified *topSort* algorithm, let us try it out on the network of Figure 6.40(a). The adjacency lists for the network are shown in Figure 6.41(a). The order of nodes on these lists determines the order in which vertices will be considered by the algorithm. At the outset, the early start time for all vertices is 0, and the start vertex is the only one in the stack. When the adjacency list for this vertex is processed, the early start time of all vertices adjacent from 0 is updated. Since vertices 1, 2, and 3 are now in the stack, all their predecessors have beer processed, and Eq. (6.2) has been evaluated for these three vertices. *ee* [5] is the next one determined. When vertex 5 is being processed, *ee* [7] is updated to 11. This, however, is not the true value for *ee* [7], since Eq. (6.2) has not been evaluated over all predecessors of 7 (v_4 has not yet been considered). This does not matter, as 7 cannot get stacked until all its predecessors have been processed. *ee* [4] is next updated to 5 and finally to 7. At this point *ee* [4] has been determined, as all the predecessors of 4 have been examined. The values of *ee* [6] and *ee* [7] are next obtained. *ee* [8] is ultimately determined to be 18, the length of a critical path. You may readily verify that when a vertex is put into the stack, its early time has been correctly computed. The insertion of the new statement does not change the asymptotic computing time; it remains $O(e+n)$.

6.5.2.2 Calculation of Late Activity Times

In the backward stage the values of $le[i]$ are computed using a function analogous to that used in the forward stage. We start with $le[n-1]=ee[n-1]$ and use the equation

$$le[j] = \min_{i \in S(j)} \{ le[i] - \text{duration of } <j,i> \} \qquad (6.3)$$

where $S(j)$ is the set of vertices adjacent from vertex j. The initial values for $le[i]$ may be set to $ee[n-1]$. Basically, Eq. (6.3) says that if $<j,i>$ is an activity and the latest start time for event i is $le[i]$, then event j must occur no later than $le[i]$ – duration of $<j,i>$. Before $le[j]$ can be computed for some event j, the latest event time for all successor events (i.e., events adjacent from j) must be computed. Once we have obtained the topological order and $ee[n-1]$ from the modified version of Program 6.14, we may compute the late event times in reverse toplogical order using the adjacency list of vertex j to access the vertices in $S(j)$. This computation is shown below for our example of Figure 6.40(a).

$le[8] = ee[8] = 18$
$le[6] = \min\{le[8] - 2\} = 16$
$le[7] = \min\{le[8] - 4\} = 14$
$le[4] = \min\{le[6] - 9, le[7] - 7\} = 7$
$le[1] = \min\{le[4] - 1\} = 6$
$le[2] = \min\{le[4] - 1\} = 6$

le [5] = min{*le* [7] − 4} = 10
le [3] = min{*le* [5] − 2} = 8
le [0] = min{*le* [1] − 6, *le* [2] − 4, *le* [3] − 5} = 0

If the forward stage has already been carried out and a topological ordering of the vertices obtained, then the values of *le* [*i*] can be computed directly, using Eq. (6.3), by performing the computations in the reverse topological order. The topological order generated in Figure 6.41(b) is 0, 3, 5, 2, 1, 4, 7, 6, 8. We may compute the values of *le* [*i*] in the order 8, 6, 7, 4, 1, 2, 5, 3, 0, as all successors of an event precede that event in this order. In practice, one would usually compute both *ee* and *le*. The procedure would then be to compute *ee* first, using algorithm *topSort*, modified as discussed for the forward stage, and then to compute *le* directly from Eq. (6.3) in reverse topological order.

Using the values of *ee* (Figure 6.41) and of *le* (above), and Eq. (6.1), we may compute the early and late times $e(i)$ and $l(i)$ and the degree of criticality (also called slack) of each task. Figure 6.42 gives the values. The critical activities are $a_1, a_4, a_7, a_8, a_{10}$, and a_{11}. Deleting all noncritical activities from the network, we get the directed graph or critical network of Figure 6.43. All paths from 0 to 8 in this graph are critical paths, and there are no critical paths in the original network that are not paths in this graph.

	early time	late time	slack	critical
activity	e	l	$l-e$	$l-e=0$
a_1	0	0	0	Yes
a_2	0	2	2	No
a_3	0	3	3	No
a_4	6	6	0	Yes
a_5	4	6	2	No
a_6	5	8	3	No
a_7	7	7	0	Yes
a_8	7	7	0	Yes
a_9	7	10	3	No
a_{10}	16	16	0	Yes
a_{11}	14	14	0	Yes

Figure 6.42: Early, late, and criticality values

As a final remark on activity networks, we note that the function *topSort* detects only directed cycles in the network. There may be other flaws, such as vertices not reachable from the start vertex (Figure 6.44). When a critical-path analysis is carried out on such networks, there will be several vertices with *ee* [*i*] = 0. Since all activity times are assumed > 0, only the start vertex can have *ee* [*i*] = 0. Hence, critical-path analysis

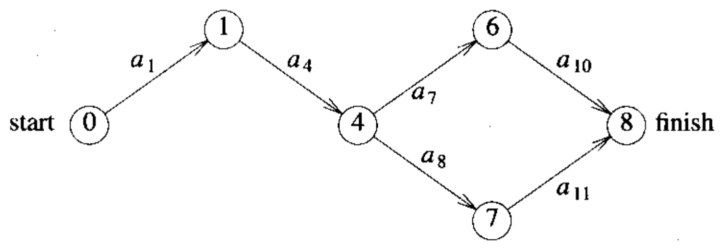


Figure 6.43: Graph obtained after deleting all noncritical activities

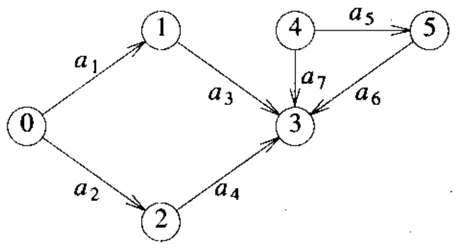


Figure 6.44: AOE network with some nonreachable activities

can also be used to detect this kind of fault in project planning.

EXERCISES

1. Does the following set of precedence relations (<) define a partial order on the elements 0 through 4? Explain your answer.

   $$0 < 1;\ 1 < 4;\ 1 < 2;\ 2 < 3;\ 2 < 4;\ 4 < 0$$

2. (a) For the AOE network of Figure 6.46, obtain the *early* and *late* starting times for each activity. Use the forward-backward approach.

   (b) What is the earliest time the project can finish?

   (c) Which activities are critical?

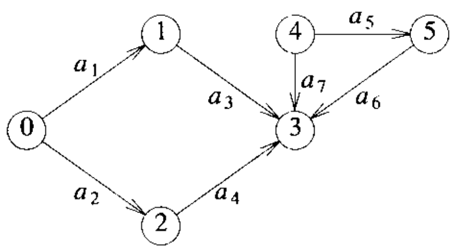


Figure 6.45: AOE network with unreachable activities

(d) Is there a single activity whose speed up would result in a reduction of the project length?

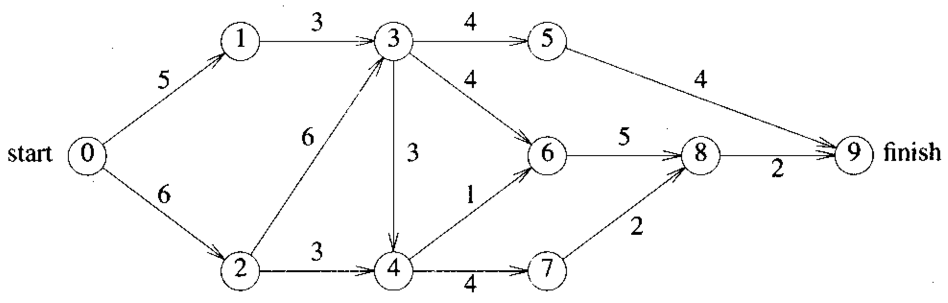


Figure 6.46: An AOE network

3. § [***Programming project***] Write a C program that allows the user to input an AOE network. The program should calculate and output the *early(i)* and *late(i)* times and the degree of criticality for each activity. If the project is not feasible, it should indicate this. If the project is feasible it should print out the critical activities in an appropriate format.

4. Define a critical AOE network to be an AOE network in which all activities are critical. Let G be the undirected graph obtained by removing the directions and weights from the edges of the network.

   (a) Show that the project length can be decreased by speeding up exactly one activity if there is an edge in G that lies on every path from the start vertex to the finish vertex. Such an edge is called a bridge. Deletion of a bridge from a connected graph separates the graph into two connected components.

(b) Write an $O(n + e)$ function using adjacency lists to determine whether the connected graph G has a bridge. If G has a bridge, your function should output one such bridge.

5. Write a program that inputs an AOE network and outputs the following:

(a) A table of all events together with their earliest and latest times.

(b) A table of all activities together with their early and late times. This table should also list the slack for each activity and identify all critical activities (see Figure 6.42).

(c) The critical network.

(d) Whether or not the project length can be reduced by speeding a single activity. If so, then by how much?

6.6 REFERENCES AND SELECTED READINGS

Euler's original paper on the Königsberg bridge problem makes interesting reading. This paper has been reprinted in: "Leonhard Euler and the Königsberg bridges," *Scientific American*, 189:1, 1953, pp. 66-70.

The biconnected-component algorithm is due to Robert Tarjan. This, together with a linear-time algorithm to find the strongly connected components of a directed graph, appears in the paper "Depth-first search and linear graph algorithms," by R. Tarjan, *SIAM Journal on Computing*, 1:2, 1972, pp. 146-159.

Prim's minimum-cost spanning tree algorithm was actually first proposed by Jarnik in 1930 and rediscovered by Prim in 1957. Since virtually all references to this algorithm give credit to Prim, we continue to refer to it as Prim's algorithm. Similarly, the algorithm we refer to as Sollin's algorithm was first proposed by Boruvka in 1926 and rediscovered by Sollin several years later. For an interesting discussion of the history of the minimum spanning tree problem, see "On the history of the minimum spanning tree problem," by R. Graham and P. Hell, *Annals of the History of Computing*, 7:1, 1985, pp. 43-57.

Further algorithms on graphs may be found in *Graphs: Theory and applications*, by K. Thulasiraman and M. Swamy, Wiley Interscience, 1992.

6.7 ADDITIONAL EXERCISES

1. A *bipartite graph* $G = (V, E)$ is an undirected graph whose vertices can be partitioned into two disjoint sets, A and $B = V - A$, with the following properties: (1) No two vertices in A are adjacent in G, and (2) no two vertices in B are adjacent in G. The graph G_4 of Figure 6.5 is bipartite. A possible partitioning of V is $A = \{0,3,4,6\}$ and $B = \{1,2,5,7\}$. Write an algorithm to determine whether a graph G is bipartite. If G is bipartite your algorithm should obtain a partitioning of the vertices into two disjoint sets, A and B, satisfying properties (1) and (2) above. Show that if G is represented by its adjacency lists, then this algorithm can be made to work in time $O(n + e)$, where $n = |V|$ and $e = |E|$.
2. Show that every tree is a bipartite graph.
3. Prove that a graph G is bipartite iff it contains no cycles of odd length.
4. The *radius* of a tree is the maximum distance from the root to a leaf. Given a connected, undirected graph, write a function to find a spanning tree of minimum radius. (Hint: Use breadth-first search.) Prove that your algorithm is correct.
5. The *diameter* of a tree is the maximum distance between any two vertices. Given a connected, undirected graph, write an algorithm for finding a spanning tree of minimum diameter. Prove the correctness of your algorithm.
6. Let $G[n][n]$ be a wiring grid. $G[i][j] > 0$ represents a grid position that is blocked; $G[i][j] = 0$ represents an unblocked position. Assume that positions $[a][b]$ and $[c][d]$ are blocked positions. A path from $[a][b]$ to $[b][c]$ is a sequence of grid positions such that

   (a) $[a][b]$ and $[c][d]$ are, respectively, the first and last positions on the path

   (b) successive positions of the sequence are vertically or horizontally adjacent in the grid

   (c) all positions of the sequence other than the first and last are unblocked positions

   The length of a path is the number of grid positions on the path. We wish to connect positions $[a][b]$ and $[c][d]$ by a wire of shortest length. The wire path is a shortest grid path between these two vertices. Lee's algorithm for this works in the following steps:

   (a) [Forward step] Start a breadth-first search from position $[a][b]$, labeling unblocked positions by their shortest distance from $[a][b]$. To avoid conflicts with existing labels, use negative labels. The labeling stops when the position $[c][d]$ is reached.

   (b) [Backtrace] Use the labels of (a) to label the shortest path between $[a][b]$ and $[c][d]$, using the unique label $w > 0$ for the wire. For this, start at position $[c][d]$.

   (c) [Clean-up] Change the remaining negative labels to 0.

Write algorithms for each of the three steps of Lee's algorithm. What is the complexity of each step?

7. Another way to represent a graph is by its incidence matrix, INC. There is one row for each vertex and one column for each edge. Then INC[i][j] = 1 if edge j is incident to vertex i. The incidence matrix for the graph of Figure 6.16(a) is given in Figure 6.47.

$$\begin{array}{c|cccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}$$

Figure 6.47: Incidence matrix of graph of Figure 6.16(a)

The edges of Figure 6.16(a) have been numbered from left to right and top to bottom. Rewrite function *DFS* (Program 6.15) so that it works on a graph represented by its incidence matrix.

8. If ADJ is the adjacency matrix of a graph $G = (V,E)$, and INC is the incidence matrix, under what conditions will ADJ = INC $\times$ INC^T − I, where INC^T is the transpose of matrix INC? I is the identity matrix, and the matrix product $C = A \times B$, where all matrices are $n \times n$, is defined as $c_{ij} = \vee_{k=0}^{n-1} a_{ik} \wedge b_{kj}$. $\vee$ is the || operation, and $\wedge$ is the && operation.

9. An edge (u, v) of a connected, undirected graph G is a *bridge* iff its deletion from G results in a graph that is not connected. In the graph of Figure 6.48, the edges (0, 1), (3, 5), (7, 8), and (7, 9) are bridges. Write an algorithm that runs in O(n +e) time to find the bridges of G. n and e are, respectively, the number of vertices and edges of G. (Hint: Use the ideas in function *Biconnected* (Program 6.16).)