Chapter Eight

Graphs and Their Applications

8.1 INTRODUCTION

This chapter investigates another nonlinear data structure: the *graph*. As we have done with other data structures, we discuss the representation of graphs in memory and present various operations and algorithms on them. In particular, we discuss the breadth-first search and the depth-first search of our graphs. Certain applications of graphs, including topological sorting, are also covered.

8.2 GRAPH THEORY TERMINOLOGY

This section summarizes some of the main terminology associated with the theory of graphs. Unfortunately, there is no standard terminology in graph theory. The reader is warned, therefore, that our definitions may be slightly different from the definitions used by other texts on data structures and graph theory.

Graphs and Multigraphs

A graph G consists of two things:

(1) A set V of elements called *nodes* (or *points* or *vertices*)
(2) A set E of *edges* such that each edge e in E is identified with a unique (unordered) pair $[u, v]$ of nodes in V, denoted by $e = [u, v]$

Sometimes we indicate the parts of a graph by writing $G = (V, E)$.

Suppose $e = [u, v]$. Then the nodes u and v are called the *endpoints* of e, and u and v are said to be *adjacent nodes* or *neighbors*. The *degree* of a node u, written $\deg(u)$, is the number of edges containing u. If $\deg(u) = 0$—that is, if u does not belong to any edge—then u is called an *isolated* node.

A *path* P of *length* n from a node u to a node v is defined as a sequence of $n + 1$ nodes.

$$P = (v_0, v_1, v_2, \ldots, v_n)$$

such that $u = v_0$; v_{i-1} is adjacent to v_i for $i = 1, 2, \ldots, n$; and $v_n = v$. The path P is said to be *closed* if $v_0 = v_n$. The path P is said to be *simple* if all the nodes are distinct, with the exception that v_0 may equal v_n; that is, P is simple if the nodes $v_0, v_1, \ldots, v_{n-1}$ are distinct and the nodes $v_1, v_2, \ldots, v_n$ are distinct. A *cycle* is a closed simple path with length 3 or more. A cycle of length k is called a *k-cycle*.

A *graph* G is said to be *connected* if there is a path between any two of its nodes. We will show (in Solved Problem 8.18) that if there is a path P from a node u to a node v, then, by eliminating unnecessary edges, one can obtain a simple path Q from u to v; accordingly, we can state the following proposition.

Proposition 8.1

A graph G is connected if and only if there is a simple path between any two nodes in G.

A graph G is said to be *complete* if every node u in G is adjacent to every other node v in G. Clearly such a graph is connected. A complete graph with n nodes will have $n(n - 1)/2$ edges.

A connected graph T without any cycles is called a *tree graph* or *free tree* or, simply, a *tree*. This means, in particular, that there is a unique simple path P between any two nodes u and v in T (Solved Problem 8.18). Furthermore, if T is a finite tree with m nodes, then T will have $m - 1$ edges (Problem 8.20).

A graph G is said to be *labeled* if its edges are assigned data. In particular, G is said to be *weighted* if each edge e in G is assigned a nonnegative numerical value $w(e)$ called the *weight* or *length* of e. In such a case, each path P in G is assigned a *weight* or *length* which is the sum of the weights of the edges along the path P. If we are given no other information about weights, we may assume any graph G to be weighted by assigning the weight $w(e) = 1$ to each edge e in G.

The definition of a graph may be generalized by permitting the following:

(1) *Multiple edges.* Distinct edges e and e' are called *multiple edges* if they connect the same endpoints, that is, if $e = [u, v]$ and $e' = [u, v]$.

(2) *Loops.* An edge e is called a *loop* if it has identical endpoints, that is, if $e = [u, u]$.

Such a generalization M is called a *multigraph*. In other words, the definition of a graph usually does not allow either multiple edges or loops.

A multigraph M is said to be *finite* if it has a finite number of nodes and a finite number of edges Observe that a graph G with a finite number of nodes must automatically have a finite number of edges and so must be finite; but this is not necessarily true for a multigraph M, since M may have multiple edges. Unless otherwise specified, graphs and multigraphs in this text shall be finite.

Example 8.1

(a) Figure 8.1(a) is a picture of a connected graph with 5 nodes—A, B, C, D and E—and 7 edges:

$$[A, B],\quad [B, C],\quad [C, D],\quad [D, E],\quad [A, E],\quad [C, E]\quad [A, C]$$

There are two simple paths of length 2 from B to E: (B, A, E) and (B, C, E). There is only one simple path of length 2 from B to D: (B, C, D). We note that (B, A, D) is not a path, since $[A, D)$ is not an edge. There are two 4-cycles in the graph:

$$[A, B, C, E, A] \quad \text{and} \quad [A, C, D, E, A].$$

Note that $\deg(A) = 3$, since A belongs to 3 edges. Similarly, $\deg(C) = 4$ and $\deg(D) = 2$.

(b) Figure 8.1(b) is not a graph but a multigraph. The reason is that it has multiple edges—$e_4 = [B, C]$ and $e_5 = [B, C]$—and it has a loop, $e_6 = [D, D)$. The definition of a graph usually does not allow either multiple edges or loops.

(c) Figure 8.1(c) is a tree graph with $m = 6$ nodes and, consequently, $m - 1 = 5$ edges. The reader can verify that there is a unique simple path between any two nodes of the tree graph.

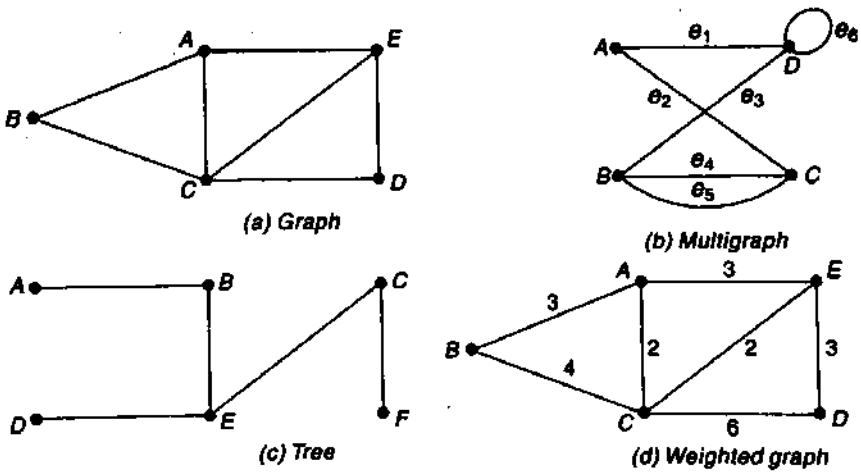


Fig. 8.1

(d) Figure 8.1(d) is the same graph as in Fig. 8.1(a), except that now the graph is weighted. Observe that $P_1 = (B, C, D)$ and $P_2 = (B, A, E, D)$ are both paths from node B to node D. Although P_2 contains more edges than P_1 the weight $w(P_2) = 9$ is less than the weight $w(P_1) = 10$.

Directed Graphs

A *directed graph* G, also called a *digraph* or *graph*, is the same as a multigraph except that each

edge e in G is assigned a direction, or in other words, each edge e is identified with an ordered pair (u, v) of nodes in G rather than an unordered pair $[u, v]$.

Suppose G is a directed graph with a directed edge $e = (u, v)$. Then e is also called an *arc*. Moreover, the following terminology is used:

(1) e *begins* at u and *ends* at v.
(2) u is the *origin* or *initial point* of e, and v is the *destination* or *terminal point* of e.
(3) u is a *predecessor* of v, and v is a *successor* or *neighbor* of u.
(4) u is *adjacent to* v, and v is *adjacent to* u.

The *outdegree* of a node u in G, written outdeg(u), is the number of edges beginning at u. Similarly, the *indegree* of u, written indeg(u), is the number of edges ending at u. A node u is called a *source* if it has a positive outdegree but zero indegree. Similarly, u is called a *sink* if it has a zero outdegree but a positive indegree.

The notions of *path*, *simple path* and *cycle* carry over from undirected graphs to directed graphs except that now the direction of each edge in a path (cycle) must agree with the direction of the path (cycle). A node v is said to be *reachable* from a node u if there is a (directed) path from u to v.

A directed graph G is said to be *connected*, or *strongly connected*, if for each pair u, v of nodes in G there is a path from u to v and there is also a path from v to u. On the other hand, G is said to be *unilaterally connected* if for any pair u, v of nodes in G there is a path from u to v or a path from v to u.

Example 8.2

Figure 8.2 shows a directed graph G with 4 nodes and 7 (directed) edges. The edges e_2 and e_3 are said to be *parallel*, since each begins at B and ends at A. The edge e_7 is a *loop*, since it begins and ends at the same point, B. The sequence $P_1 = (D, C, B, A)$ is not a path, since (C, B) is not an edge—that is, the direction of the edge $e_5 = (B, C)$ does not agree with the direction of the path P_1. On the other hand, $P_2 = (D, B, A)$ is a path from D to A, since (D, B) and (B, A) are edges. Thus A is reachable from D. There is no path from C to any other node, so G is not strongly connected. However, G is unilaterally connected. Note that indeg(D) = 1 and outdeg(D) = 2. Node C is a sink, since indeg(C) = 2 but outdeg(C) = 0. No node in G is a source.

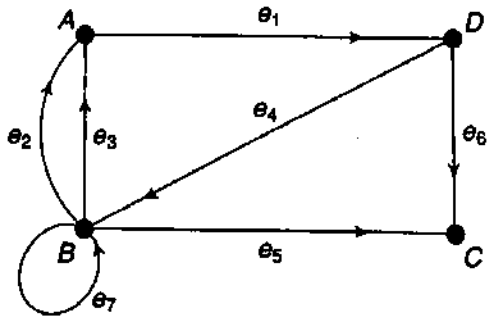


Fig. 8.2

Let T be any nonempty tree graph. Suppose we choose any node R in T. Then T with this designated node R is called a *rooted tree* and R is called its *root*. Recall that there is a unique simple path from the root R to any other node in T. This defines a direction to the edges in T, so the rooted tree T may be viewed as a directed graph. Furthermore, suppose we also order the successors of each node v in T. Then T is called an *ordered rooted tree*. Ordered rooted trees are nothing more than the general trees discussed in Chapter 7.

A directed graph G is said to be *simple* if G has no parallel edges. A simple graph G may have loops, but it cannot have more than one loop at a given node. A nondirected graph G may be viewed as a simple directed graph by assuming that each edge $[u, v]$ in G represents two directed edges, (u, v) and (v, u). (Observe that we use the notation $[u, v]$ to denote an unordered pair and the notation (u, v) to denote an ordered pair.)

Warning: The main subject matter of this chapter is simple directed graphs. Accordingly, unless otherwise stated or implied, the term "graph" shall mean simple directed graph, and the term "edge" shall mean directed edge.

8.3 SEQUENTIAL REPRESENTATION OF GRAPHS; ADJACENCY MATRIX; PATH MATRIX

There are two standard ways of maintaining a graph G in the memory of a computer. One way, called the *sequential representation* of G, is by means of its adjacency matrix A. The other way, called the *linked representation* of G, is by means of linked lists of neighbors. This section covers the first representation, and shows how the adjacency matrix A of G can be used to easily answer certain questions of connectivity in G. The linked representation of G will be covered in Sec. 8.5.

Regardless of the way one maintains a graph G in the memory of the computer, the graph G is normally input into the computer by using its formal definition: a collection of nodes and a collection of edges.

Adjacency Matrix

Suppose G is a simple directed graph with m nodes, and suppose the nodes of G have been ordered and are called $v_1, v_2, \ldots, v_m$. Then the *adjacency matrix* $A = (a_{ij})$ of the graph G is the $m \times m$ matrix defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j, \text{ that is, if there is an edge } (v_i, v_j) \\ 0 & \text{otherwise} \end{cases}$$

Such a matrix A, which contains entries of only 0 and 1, is called a *bit matrix* or a *Boolean matrix*.

The adjacency matrix A of the graph G does depend on the ordering of the nodes of G; that is, a different ordering of the nodes may result in a different adjacency matrix. However, the matrices resulting from two different orderings are closely related in that one can be obtained from the other by simply interchanging rows and columns. Unless otherwise stated, we will assume that the nodes of our graph G have a fixed ordering.

Suppose G is an undirected graph. Then the adjacency matrix A of G will be a *symmetric matrix*, i.e., one in which $a_{ij} = a_{ji}$ for every i and j. This follows from the fact that each undirected edge $[u, v]$ corresponds to the two directed edges (u, v) and (v, u).

The above matrix representation of a graph may be extended to multigraphs. Specifically, if G is a multigraph, then the *adjacency matrix* of G is the $m \times m$ matrix $A = (a_{ij})$ defined by setting a_{ij} equal to the number of edges from v_i to v_j.

Example 8.3

Consider the graph G in Fig. 8.3. Suppose the nodes are stored in memory in a linear array DATA as follows:

DATA: X, Y, Z, W

Then we assume that the ordering of the nodes in G is as follows: $v_1 = X$, $v_2 = Y$, $v_3 = Z$ and $v_4 = W$. The adjacency matrix A of G is as follows:

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Note that the number of 1's in A is equal to the number of edges in G.

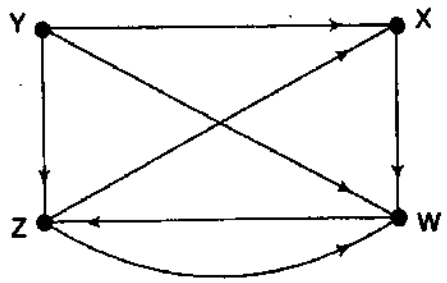


Fig. 8.3

Consider the powers $A, A^2, A^3, \ldots$ of the adjacency matrix A of a graph G. Let

$$a_K(i, j) = \text{the } ij \text{ entry in the matrix } A^K$$

Observe that $a_1(i, j) = a_{ij}$ gives the number of paths of length 1 from node v_i to node v_j. One can show that $a_2(i, j)$ gives the number of paths of length 2 from v_i to v_j. In fact, we prove in Miscellaneous Problem 8.3 the following general result.

Proposition 8.2

Let A be the adjacency matrix of a graph G. Then $a_K(i, j)$, the ij entry in the matrix A^K, gives the number of paths of length K from v_i to v_j.

Consider again the graph G in Fig. 8.3, whose adjacency matrix A is given in Example 8.3. The powers A^2, A^3 and A^4 of the matrix A follow:

$$A^2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 2 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} \qquad A^3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 2 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \qquad A^4 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 2 & 0 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 1 & 0 & 1 & 1 \end{pmatrix}.$$

Accordingly, in particular, there is a path of length 2 from v_4 to v_1, there are two paths of length 3 from v_2 to v_3, and there are three paths of length 4 from v_2 to v_4. (Here, $v_1 = X$, $v_2 = Y$, $v_3 = Z$ and $v_4 = W$.)

Suppose we now define the matrix B_r as follows:

$$B_r = A + A^2 + A^3 + \cdots + A^r$$

Then the ij entry of the matrix B_r gives the number of paths of length r or less from node v_i to v_j.

Path Matrix

Let G be a simple directed graph with m nodes, $v_1, v_2, \ldots, v_m$. The *path matrix* or *reachability matrix* of G is the m-square matrix $P = (p_{ij})$ defined as follows:

$$P_{ij} = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Suppose there is a path from v_i to v_j. Then there must be a simple path from v_i to v_j when $v_i \neq v_j$, or there must be a cycle from v_i to v_j when $v_i = v_j$. Since G has only m nodes, such a simple path must have length $m - 1$ or less, or such a cycle must have length m or less. This means that there is a nonzero ij entry in the matrix B_m, defined at the end of the preceding subsection. Accordingly, we have the following relationship between the path matrix P and the adjacency matrix A.

Proposition 8.3

Let A be the adjacency matrix and let $P = (p_{ij})$ be the path matrix of a digraph G. Then $P_{ij} = 1$ if and only if there is a nonzero number in the ij entry of the matrix

$$B_m = A + A^2 + A^3 + \cdots + A^m$$

Consider the graph G with $m = 4$ nodes in Fig. 8.3. Adding the matrices A, A^2, A^3 and A^4, we obtain the following matrix B_4, and, replacing the nonzero entries in B_4 by 1, we obtain the path matrix P of the graph G:

$$B_4 = \begin{pmatrix} 1 & 0 & 2 & 3 \\ 5 & 0 & 6 & 8 \\ 3 & 0 & 3 & 5 \\ 2 & 0 & 3 & 3 \end{pmatrix} \quad \text{and} \quad P = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Examining the matrix P, we see that the node v_2 is not reachable from any of the other nodes.

Recall that a directed graph G is said to be *strongly connected* if, for any pair of nodes u and v in G, there are both a path from u to v and a path from v to u. Accordingly, G is strongly connected if and only if the path matrix P of G has no zero entries. Thus the graph G in Fig. 8.3 is not strongly connected.

The *transitive closure* of a graph G is defined to be the graph G' such that G' has the same nodes as G and there is an edge (v_i, v_j) in G' whenever there is a path from v_i to v_j in G. Accordingly, the path matrix P of the graph G is precisely the adjacency matrix of its transitive closure G'. Furthermore, a graph G is strongly connected if and only if its transitive closure is a complete graph:

Remark: The adjacency matrix A and the path matrix P of a graph G may be viewed as logical (Boolean) matrices, where 0 represents "false" and 1 represents "true." Thus, the logical operations of $\wedge$(AND) and $\vee$(OR) may be applied to the entries of A and P. The values of $\wedge$ and $\vee$ appear in Fig. 8.4. These operations will be used in the next section.

$\wedge$	0	1
0	0	0
1	0	1

(a) AND

$\vee$	0	1
0	0	1
1	1	1

(b) OR

Fig. 8.4

8.4 WARSHALL'S ALGORITHM; SHORTEST PATHS

Let G be a directed graph with m nodes, $v_1, v_2, \ldots, v_m$. Suppose we want to find the path matrix P of the graph G. Warshall gave an algorithm for this purpose that is much more efficient than calculating the powers of the adjacency matrix A and using Proposition 8.3. This algorithm is described in this section, and a similar algorithm is used to find shortest paths in G when G is weighted.

First we define m-square Boolean matrices $P_0, P_1, \ldots, P_m$ as follows. Let $P_k[i, j]$ denote the ij entry of the matrix P_k. Then we define:

$$P_k[i, j] = \begin{cases} 1 & \text{if there is a simple path from } v_i \text{ to } v_j \text{ which does not} \\ & \text{use any other nodes except possibly } v_1, v_2, \ldots, v_k \\ 0 & \text{otherwise} \end{cases}$$

In other words,

$P_0[i, j] = 1$ if there is an edge from v_i to v_j

$P_1[i, j] = 1$ if there is a simple path from v_i to v_j which does not use any other nodes except possibly v_1

$P_2[i, j] = 1$ if there is a simple path from v_i to v_j which does not use any other nodes except possibly v_1 and v_2

First observe that the matrix $P_0 = A$, the adjacency matrix of G. Furthermore, since G has only m nodes, the last matrix $P_m = P$, the path matrix of G.

Warshall observed that $P_k[i, j] = 1$ can occur only if one of the following two cases occurs:

(1) There is a simple path from v_i to v_j which does not use any other nodes except possibly v_1, v_2, ..., v_{k-1}; hence

$$P_{k-1}[i, j] = 1$$

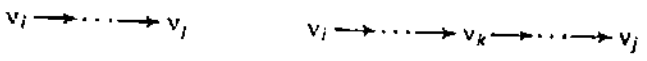
(2) There is a simple path from v_i to v_k and a simple path from v_k to v_j where each path does not use any other nodes except possibly v_1, v_2, ..., v_{k-1}; hence

$$P_{k-1}[i, k] = 1 \quad \text{and} \quad P_{k-1}[k, j] = 1$$

These two cases are pictured, respectively, in Fig. 8.5(*a*) and (*b*), where

$$\longrightarrow\cdots\longrightarrow$$

denotes part of a simple path which does not use any nodes except possibly $v_1, v_2, \ldots, v_{k-1}$.



(a) (b)

Fig. 8.5

Accordingly, the elements of the matrix P_k can be obtained by

$$P_k[i, j] = P_{k-1}[i, j] \vee (P_{k-1}[i, k] \wedge P_{k-1}[k, j])$$

where we use the logical operations of $\wedge$ (AND) and $\vee$ (OR). In other words we can obtain each entry in the matrix P_k by looking at only three entries in the matrix P_{k-1}. Warshall's algorithm follows.

Algorithm 8.1: (Warshall's Algorithm) A directed graph G with M nodes is maintained in memory by its adjacency matrix A. This algorithm finds the (Boolean) path matrix P of the graph G.

```
1. Repeat for I, J = 1, 2,..., M: [Initializes P.]
       If A[I, J] = 0, then: Set P[I, J] := 0;
       Else: Set P[I, J] := 1.
   [End of loop.]
2. Repeat Steps 3 and 4 for K = 1, 2, ..., M: [Updates P.]
3.     Repeat Step 4 for I = 1, 2, ..., M:
4.         Repeat for J = 1, 2, ..., M:
               Set P[I, J] : = P[I, J] ∨ (P[I, K] ∧ P[K, J]).
           [End of loop.]
       [End of Step 3 loop.]
   [End of Step 2 loop.]
5. Exit.
```

Shortest-Path Algorithm

Let G be a directed graph with m nodes, $v_1, v_2, \ldots, v_m$. Suppose G is *weighted;* that is, suppose each edge e in G is assigned a nonnegative number $w(e)$ called the *weight* or *length* of the edge e. Then G may be maintained in memory by its *weight matrix* $W = (w_{ij})$, defined as follows:

$$w_{ij} = \begin{cases} w(e) & \text{if there is an edge } e \text{ from } v_i \text{ to } v_j \\ 0 & \text{if there is no edge from } v_i \text{ to } v_j \end{cases}$$

The path matrix P tells us whether or not there are paths between the nodes. Now we want to find a matrix Q which will tell us the lengths of the shortest paths between the nodes or, more exactly, a matrix $Q = (q_{ij})$ where

$$q_{ij} = \text{length of a shortest path from } v_i \text{ to } v_j$$

Next we describe a modification of Warshall's algorithm which finds us the matrix Q.

Here we define a sequence of matrices $Q_0, Q_1, \ldots, Q_m$ (analogous to the above matrices $P_0, P_1, \ldots, P_m$) whose entries are defined as follows:

$Q_k[i, j]$ = the smaller of the length of the preceding path from v_i to v_j or the sum of the lengths of the preceding paths from vi to v_i and from v_k to v_j

More exactly,

$$Q_k[i, j] = \text{MIN}(Q_{k-1}[i, j]), \quad Q_{k-1}[i, k] + Q_{k-1}[k, j])$$

The initial matrix Q_0 is the same as the weight matrix W except that each 0 in W is replaced by ∞ (or a very, very large number). The final matrix Q_m will be the desired matrix Q.

Example 8.4

Consider the weighted graph G in Fig. 8.6. Assume $v_1 = R$, $v_2 = S$, $v_3 = T$ and $v_4 = U$. Then the weight matrix W of G is as follows:

$$W = \begin{pmatrix} 7 & 5 & 0 & 0 \\ 7 & 0 & 0 & 2 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 1 & 0 \end{pmatrix}$$

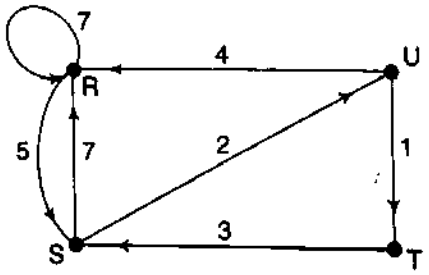


Fig. 8.6

Applying the modified Warshall's algorithm, we obtain the following matrices Q_0, Q_1, Q_2, Q_3 and $Q_4 = Q$. To the right of each matrix Q_k, we show the matrix of paths which correspond to the lengths in the matrix Q_k.

$$Q_0 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{pmatrix} \quad \begin{pmatrix} RR & RS & — & — \\ SR & — & — & SU \\ — & TS & — & — \\ UR & — & UT & — \end{pmatrix}$$

$$Q_1 = \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & 12 & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \textcircled{9} & 1 & \infty \end{pmatrix} \quad \begin{pmatrix} RR & RS & — & — \\ SR & SRS & — & SU \\ — & TS & — & — \\ UR & URS & UT & — \end{pmatrix}$$

$$Q_2 = \begin{pmatrix} 7 & 5 & \textcircled{\infty} & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & 9 & 1 & 11 \end{pmatrix} \quad \begin{pmatrix} RR & RS & — & RSU \\ SR & SRS & — & SU \\ TSR & TS & — & TSU \\ UR & URS & UT & URS \end{pmatrix}$$

$$Q_3 = \begin{pmatrix} 7 & 5 & \infty & 7 \\ 7 & 12 & \infty & 2 \\ 10 & 3 & \infty & 5 \\ 4 & \textcircled{4} & 1 & 6 \end{pmatrix} \quad \begin{pmatrix} RR & RS & — & RSU \\ SR & SRS & — & SU \\ TSR & TS & — & TSU \\ UR & UTS & UT & UTSU \end{pmatrix}$$

$$Q_4 = \begin{pmatrix} 7 & 5 & 8 & 7 \\ 7 & 11 & 3 & 2 \\ \textcircled{9} & 3 & 6 & 5 \\ 4 & 4 & 1 & 6 \end{pmatrix} \quad \begin{pmatrix} RR & RS & RSUT & RSU \\ SR & SURS & SUT & SU \\ TSUR & TS & TSUT & TSU \\ UR & UTS & UT & UTSU \end{pmatrix}$$

We indicate how the circled entries are obtained:

$$Q_1[4, 2] = \text{MIN}(Q_0[4, 2], Q_0[4, 1] + Q_0[1, 2]) = \text{MIN}(\infty, 4 + 5) = 9$$
$$Q_2[1, 3] = \text{MIN}(Q_1[1, 3], Q_1[1, 2] + Q_1[2, 3]) = \text{MIN}(\infty, 5 + \infty) = \infty$$
$$Q_3[4, 2] = \text{MIN}(Q_2[4, 2], Q_2[4, 3] + Q_2[3, 2]) = \text{MIN}(9, 3 + 1) = 4$$
$$Q_4[3, 1] = \text{MIN}(Q_3[3, 1], Q_3[3, 4] + Q_3[4, 1]) = \text{MIN}(10, 5 + 4) = 9$$

The formal statement of the algorithm follows.

Algorithm 8.2: (Shortest-Path Algorithm) A weighted graph G with M nodes is maintained in memory by its weight matrix W. This algorithm finds a matrix Q such that Q[I, J] is the length of a shortest path from node V_I to node V_J. INFINITY is a very large number, and MIN is the minimum value function.

1. Repeat for I, J = 1, 2, ..., M: [Initializes Q.]
   W[I, J] = 0, then: Set Q[I, J] := INFINITY;
   Else: Set Q[I, J]:= W[I, J].
   [End of loop.]
2. Repeat Steps 3 and 4 for K = 1, 2, ..., M: [Updates Q.]

```
3.      Repeat Step 4 for I = 1, 2, ..., M:
4.          Repeat for J = 1, 2, ..., M:
                Set Q[I, J]:= MIN(Q[I, J], Q[I, K] + Q[K, J]).
            [End of loop.]
        [End of Step 3 loop.]
    [End of Step 2 loop.]
5.  Exit.
```

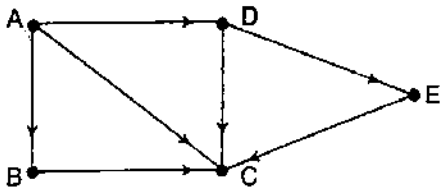
Observe the similarity between Algorithm 8.1 and Algorithm 8.2.

Algorithm 8.2 can also be used for a graph G without weights by simply assigning the weight $w(e) = 1$ to each edge e in G.

8.5 LINKED REPRESENTATION OF A GRAPH

Let G be a directed graph with m nodes. The sequential representation of G in memory—i.e., the representation of G by its adjacency matrix A—has a number of major drawbacks. First of all, it may be difficult to insert and delete nodes in G. This is because the size of A may need to be changed and the nodes may need to be reordered, so there may be many, many changes in the matrix A. Furthermore, if the number of edges is $O(m)$ or $O(m \log_2 m)$, then the matrix A will be sparse (will contain many zeros); hence a great deal of space will be wasted. Accordingly, G is usually represented in memory by a linked *representation*, also called an *adjacency structure*, which is described in this section.

Consider the graph G in Fig. 8.7(a). The table in Fig. 8.7(b) shows each node in G followed by its *adjacency list*, which is its list of adjacent nodes, also called its *successors* or *neighbors*. Figure 8.8 shows a schematic diagram of a linked representation of G in memory. Specifically, the linked representation will contain two lists (or files), a node list NODE and an edge list EDGE, as follows.



(a) Graph G

Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

(b) Adjacency list of G

Fig. 8.7

(a) *Node list.* Each element in the list NODE will correspond to a node in G, and it will be a record of the form:

NODE	NEXT	ADJ	

Here NODE will be the name or key value of the node, NEXT will be a pointer to the next node in the list NODE and ADJ will be a pointer to the first element in the adjacency list of the node, which is maintained in the list EDGE. The shaded area indicates that there may be

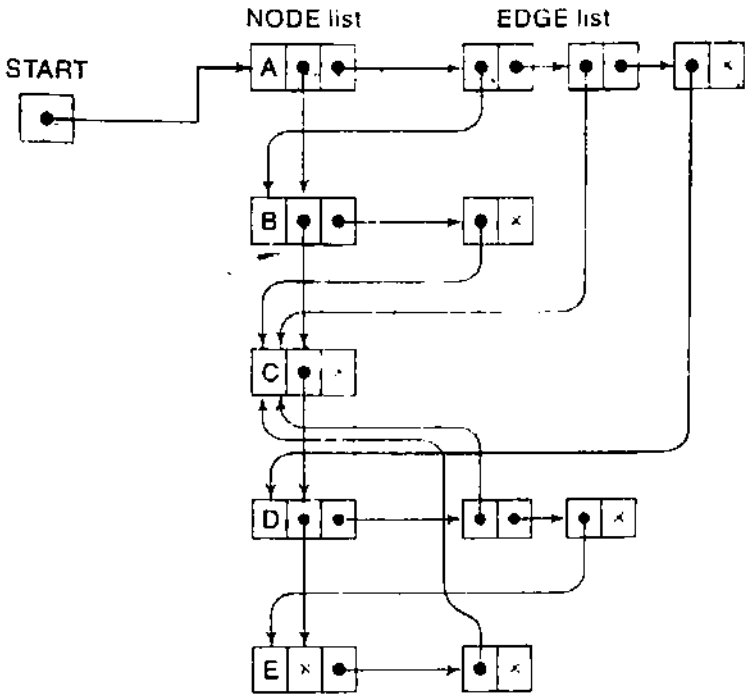


Fig. 8.8

other information in the record, such as the indegree INDEG of the node, the outdegree OUTDEG of the node, the STATUS of the node during the execution of an algorithm, and so on. (Alternatively, one may assume that NODE is an array of records containing fields such as NAME, INDEG, OUTDEG, STATUS, ...) The nodes themselves, as pictured in Fig. 8.7, will be organized as a linked list and hence will have a pointer variable START for the beginning of the list and a pointer variable AVAILN for the list of available space. Sometimes, depending on the application, the nodes may be organized as a sorted array or a binary search tree instead of a linked list.

(b) *Edge list.* Each element in the list EDGE will correspond to an edge of G and will be a record of the form:

DEST	LINK	

The field DEST will point to the location in the list NODE of the destination or terminal node of the edge. The field LINK will link together the edges with the same initial node, that is, the nodes in the same adjacency list. The shaded area indicates that there may be other information in the record corresponding to the edge, such as a field EDGE containing the labeled data of the edge when G is a labeled graph, a field WEIGHT containing the weight of the edge when G is a weighted graph, and so on. We also need a pointer variable AVAILE for the list of available space in the list EDGE.

Figure 8.9 shows how the graph G in Fig. 8.7(a) may appear in memory. The choice of 10 locations for the list NODE and 12 locations for the list EDGE is arbitrary.

The linked representation of a graph G that we have been discussing may be denoted by

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

The representation may also include an array WEIGHT when G is weighted or may include an array EDGE when G is a labeled graph.

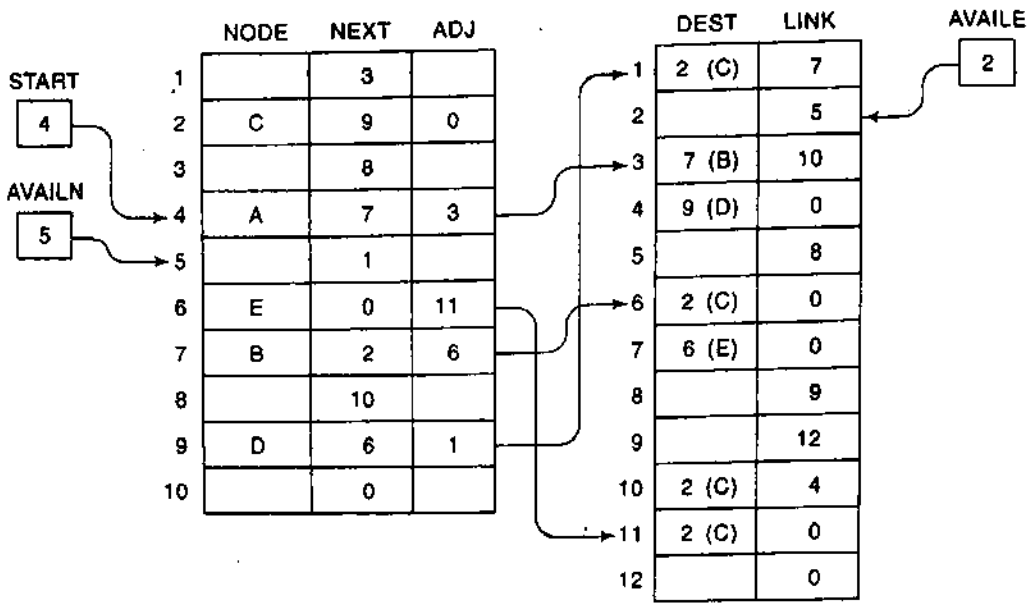


Fig. 8.9

Example 8.5

Suppose Friendly Airways has nine daily flights, as follows:

103	Atlanta to Houston	203	Boston to Denver	305	Chicago to Miami
106	Houston to Atlanta	204	Denver to Boston	308	Miami to Boston
201	Boston to Chicago	301	Denver to Reno	402	Reno to Chicago

Clearly, the data may be stored efficiently in a file where each record contains three fields:

Flight Number, City of Origin, City of Destination

However, such a representation does not easily answer the following natural questions:

(a) Is there a direct flight from city X to city Y?
(b) Can one fly, with possible stops, from city X to city Y?
(c) What is the most direct route, i.e., the route with the smallest number of stops, from city X to city Y?

To make the answers to these questions more readily available, it may be very useful for the data to be organized also as a graph G with the cities as nodes and with the flights as edges. Figure 8.10 is a picture of the graph G.

Figure 8.11 shows how the graph G may appear in memory using the linked representation. We note that G is a labeled graph, not a weighted graph, since the flight number is simply for identification. Even though the data are organized as a graph, one still would require some type of algorithm to answer questions (*b*) and (*c*). Such algorithms are discussed later in the chapter.

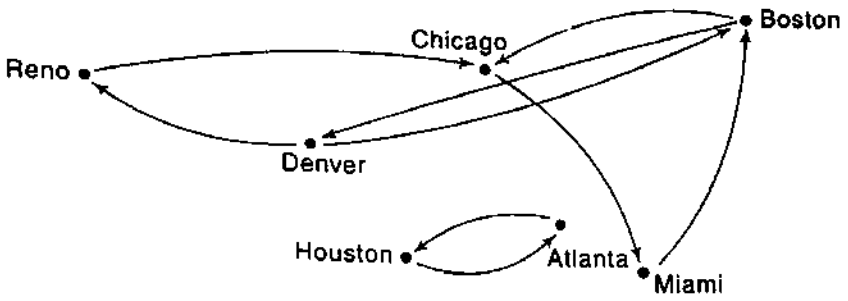


Fig. 8.10

NODE list

	CITY	NEXT	ADJ
1		0	
2	Atlanta	12	1
3	Chicago	11	7
4	Houston	7	2
5		6	
6		8	
7	Miami	10	8
8		9	
9		1	
10	Reno	0	9
11	Denver	4	5
12	Boston	3	3

START = 2, AVAILN = 5

EDGE list

	NUMBER	ORIG	DEST	LINK
1	103	2	4	0
2	106	4	2	0
3	201	12	3	4
4	203	12	11	0
5	204	11	12	6
6	301	11	10	0
7	305	3	7	0
8	308	7	12	0
9	402	10	3	0
10				11
11				12
12				0

AVAILE = 10

Fig. 8.11

8.6 OPERATIONS ON GRAPHS

Suppose a graph G is maintained in memory by the linked representation

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

as discussed in the preceding section. This section discusses the operations of searching, inserting and deleting nodes and edges in the graph G. The operation of traversing is treated in the next section.

The operations in this section use certain procedures from Chapter 5, on linked lists. For completeness, we restate these procedures below, but in a slightly different manner than in Chapter 5. Naturally, if a circular linked list or a binary search tree is used instead of a linked list, then the analogous procedures must be used.

Procedure 8.3 (originally Algorithm 5.2) finds the location LOC of an ITEM in a linked list.

Procedure 8.4 (originally Procedure 5.9 and Algorithm 5.10) deletes a given ITEM from a linked list. Here we use a logical variable FLAG to tell whether or not ITEM originally appears in the linked list.

Searching in a Graph

Suppose we want to find the location LOC of a node N in a graph G. This can be accomplished by using Procedure 8.3, as follows:

Call FINP(NODE, NEXT, START, N, LOC)

That is, this Call statement searches the list NODE for the node N.

On the other hand, suppose we want to find the location LOC of an edge (A, B) in the graph G. First we must find the location LOCA of A and the location LOCB of B in the list NODE. Then we must find in the list of successors of A, which has the list pointer ADJ[LOCA], the location LOC of LOCB. This is implemented by Procedure 8.5, which also checks to see whether A and B are nodes in G. Observe that LOC gives the location of LOCB in the list EDGE.

Inserting in a Graph

Suppose a node N is to be inserted in the graph G. Note that N will be assigned to NODE[AVAILN], the first available node. Moreover, since N will be an isolated node, one must also set ADJ[AVAILN] := NULL. Procedure 8.6 accomplishes this task using a logical variable FLAG to indicate overflow.

Clearly, Procedure 8.6 must be modified if the list NODE is maintained as a sorted list or a binary search tree.

Procedure 8.3: FIND(INFO, LINK START, ITEM, LOC) [Algorithm 5.2]
Finds the location LOC of the first node containing ITEM, or sets LOC := NULL.

1. Set PTR := START.
2. Repeat while PTR ≠ NULL:
   If ITEM = INFO[PTR], then: Set LOC := PTR, and Return.
   Else: Set PTR := LINK[PTR].
   [End of loop.]
3. Set LOC := NULL, and Return.

Procedure 8.4: DELETE(INFO, LINK, START, AVAIL, ITEM, FLAG) [Algorithm 5.10]
Deletes the first node in the list containing ITEM, or sets FLAG := FALSE when ITEM does not appear in the list.

1. [List empty?] If START = NULL, then: Set FLAG := FALSE, and Return.
2. [ITEM in first node?] If INFO[START], = ITEM, then:
   Set PTR := START, START := LINK[START],
   LINK[PTR] := AVAIL, AVAIL := PTR,
   FLAG := TRUE, and Return.
   [End of If structure.]

3. Set PTR := LINK[START] and SAVE := START. [Initializes pointers.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL:
5. If INFO[PTR] = ITEM, then:
   Set LINK[SAVE] := LINK[PTR], LINK[PTR] := AVAIL, AVAIL := PTR, FLAG := TRUE, and Return.
   [End of If structure.]
6. Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers]
[End of Step 4 loop.]
7. Set FLAG := FALSE, and Return.

Procedure 8.5: FINDEDGE(NODE, NEXT, ADJ, START, DEST, LINK, A, B, LOC)
This procedure finds the location LOC of an edge (A, B) in the graph G, or sets LOC := NULL.

1. Call FIND(NODE, NEXT, START, A, LOCA).
2. CALL FIND(NODE, NEXT, START, B, LOCB).
3. If LOCA = NULL or LOCB = NULL, then: Set LOC := NULL.
   Else: Call FIND(DEST, LINK, ADJ[LOCA], LOCB, LOC).
4. Return.

Procedure 8.6: INSNODE(NODE, NEXT, ADJ, START, AVAILN, N, FLAG)
This procedure inserts the node N in the graph G.

1. [OVERFLOW?] If AVAILN = NULL, then: Set FLAG := FALSE, and Return.
2. Set ADJ[AVAILN] := NULL.
3. [Removes node from AVAILN list]
   Set NEW := AVAILN and AVAILN := NEXT[AVAILN].
4. [Inserts node N in the NODE list.]
   Set NODE[NEW] := N, NEXT[NEW] := START and START := NEW.
5. Set FLAG := TRUE, and Return.

Suppose an edge (A, B) is to be inserted in the graph *G*. (The procedure will assume that both A and B are already nodes in the graph *G*.) The procedure first finds the location LOCA of A and the location LOCB of B in the node list. Then (A, B) is inserted as an edge in *G* by inserting LOCB in the list of successors of A, which has the list pointer ADJ[LOCA]. Again, a logical variable FLAG is used to indicate overflow. The procedure follows.

Procedure 8.7: INSEDGE(NODE, NEXT, ADJ, START, DEST, LINK, AVAILE, A, B, FLAG)
This procedure inserts the edge (A, B) in the graph G.

1. Call FIND(NODE, NEXT, START, A, LOCA).
2. Call FIND(NODE, NEXT, START, B, LOCB).
3. [OVERFLOW?] If AVAILE = NULL, then: Set FLAG := FALSE, and Return.
4. [Remove node from AVAILE list.] Set NEW := AVAILE and AVAILE := LINK[AVAILE].

5. [Insert LOCB in list of successors of A.]
   Set DEST[NEW] := LOCB, LINK[NEW] := ADJ[LOCA] and ADJ[LOCA] := NEW.
6. Set FLAG := TRUE, and Return.

The procedure must be modified by using Procedure 8.6 if A or B is not a node in the graph *G*.

Deleting from a Graph

Suppose an edge (A, B) is to be deleted from the graph *G*. (Our procedure will assume that A and B are both nodes in the graph *G*.) Again, we must first find the location LOCA of A and the location LOCB of B in the node list. Then we simply delete LOCB from the list of successors of A, which has the list pointer ADJ[LOCA]. A logical variable FLAG is used to indicate that there is no such edge in the graph *G*. The procedure follows.

Procedure 8.8: DELEDGE(NODE, NEXT, ADJ, START, DEST, LINK, AVAILE, A, B, FLAG)
This procedure deletes the edge (A, B) from the graph G.

1. Call FIND(NODE, NEXT, START, A, LOCA). [Locates node A.]
2. Call FIND(NODE, NEXT, START, B, LOCB). [Locates node B.]
3. Call DELETE(DEST, LINK, ADJ[LOCA], AVAILE, LOCB, FLAG). [Uses Procedure 8.4.]
4. Return.

Suppose a node N is to be deleted from the graph *G*. This operation is more complicated than the search and insertion operations and the deletion of an edge, because we must also delete all the edges that contain N. Note these edges come in two kinds; those that begin at N and those that end at N. Accordingly, our procedure will consist mainly of the following four steps:

(1) Find the location LOC of the node N in *G*.
(2) Delete all edges ending at N; that is, delete LOC from the list of successors of each node M in *G*. (This step requires traversing the node list of *G*.)
(3) Delete all the edges beginning at N. This is accomplished by finding the location BEG of the first successor and the location END of the last successor of N, and then adding the successor list of N to the free AVAILE list.
(4) Delete N itself from the list NODE.

The procedure follows.

Procedure 8.9: DELNODE(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE, N, FLAG)
This procedure deletes the node N from the graph G.

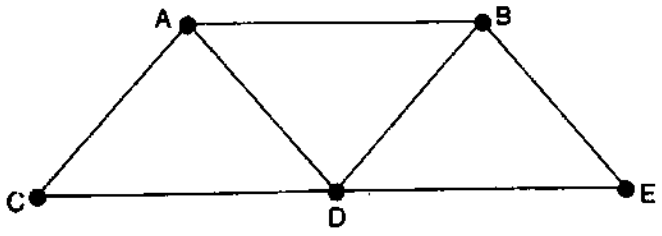
1. Call FIND(NODE, NEXT, START, N, LOC). [Locates node N.]
2. If LOC = NULL, then: Set FLAG := FALSE, and Return.
3. [Delete edges ending at N.]

(a) Set PTR := START.
(b) Repeat while PTR ≠ NULL:
  (i) Call DELETE(DEST, LINK, ADJ[PTR], AVAILE, LOC, FLAG).
  (ii) Set PTR := NEXT[PTR].
  [End of loop.]

4. [Successor list empty?] If ADJ[LOC] = NULL, then: Go to Step 7.
5. [Find the first and last successor of N.]
   (a) Set BEG := ADJ[LOC], END := ADJ[LOC] and PTR := LINK[END].
   (b) Repeat while PTR ≠ NULL:
     Set END := PTR and PTR := LINK[PTR].
   [End of loop.]
6. [Add successor list of N to AVAILE list.]
   Set LINK[END] := AVAILE and AVAILE := BEG.
7. [Delete N using Procedure 8.4.]
   Call DELETE(NODE, NEXT, START, AVAILN, N, FLAG).
8. Return.

Example 8.6

Consider the (undirected) graph G in Fig. 8.12(a), whose adjacency lists appear in Fig. 8.12(b). Observe that G has 14 directed edges, since there are 7 undirected edges.



(a)

Adjacency lists
A: B, C, D
B: A, D, E
C: A, D
D: A, B, C, E
E: B, D

(b)

Fig. 8.12

Suppose G is maintained in memory as in Fig. 8.13(a). Furthermore, suppose node B is deleted from G by using Procedure 8.9. We obtain the following steps:

Step 1. Finds LOC = 2, the location of B in the node list.
Step 3. Deletes LOC = 2 from the edge list, that is, from each list of successors.
Step 5. Finds BEG = 4 and END = 6, the first and last successors of B.
Step 6. Deletes the list of successors from the edge list.
Step 7. Deletes node B from the node list.
Step 8. Returns.

The deleted elements are circled in Fig. 8.13(a). Figure 8.13(b) shows G in memory after node B (and its edges) are deleted.

	NODE	NEXT	ADJ
1	A	2	1
2	Ⓑ	3	4
3	C	4	7
4	D	5	9
5	E	0	13
6		7	
7		8	
8		0	

START = 1
AVAILN = 6

	NODE	NEXT	ADJ
1	A	3	2
2		6	
3	C	4	7
4	D	5	9
5	E	0	14
6		7	
7		8	
8		0	

START = 1
AVAILN = 2

	DEST	LINK
1	②	2
2	3	3
3	4	0
4	①	5
5	④	6
6	⑤	0
7	1	8
8	4	0
9	1	10
10	②	11
11	3	12
12	5	0
13	②	14
14	4	0
15		16
16		0

AVAILE = 16
(a) Before deletion

	DEST	LINK
1		15
2	3	3
3	4	0
4		5
5		6
6		13
7	1	8
8	4	0
9	1	11
10		1
11	3	12
12	5	0
13		10
14	4	0
15		16
16		0

AVAILE = 4
(b) After deleting B

Fig. 8.13

8.7 TRAVERSING A GRAPH

Many graph algorithms require one to systematically examine the nodes and edges of a graph G. There are two standard ways that this is done. One way is called a breadth-first search, and the other is called a depth-first search. The breadth-first search will use a queue as an auxiliary structure to hold nodes for future processing, and analogously, the depth-first search will use a stack.

During the execution of our algorithms, each node N of G will be in one of three states, called the *status* of N, as follows:

STATUS = 1: (Ready state.) The initial state of the node N.
STATUS = 2: (Waiting state.) The node N is on the queue or stack, waiting to be processed.
STATUS = 3: (Processed state.) The node N has been processed.

We now discuss the two searches separately.

Breadth-First Search

The general idea behind a breadth-first search beginning at a starting node A is as follows. First we examine the starting node A. Then we examine all the neighbors of A. Then we examine all the neighbors of the neighbors of A. And so on. Naturally, we need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a queue to hold nodes that are waiting to be processed, and by using a field STATUS which tells us the current status of any node. The algorithm follows.

Algorithm A: This algorithm executes a breadth-first search on a graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Put the starting node A in QUEUE and change its status to the waiting state (STATUS = 2).
3. Repeat Steps 4 and 5 until QUEUE is empty:
4. Remove the front node N of QUEUE. Process N and change the status of N to the processed state (STATUS = 3).
5. Add to the rear of QUEUE all the neighbors of N that are in the steady state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

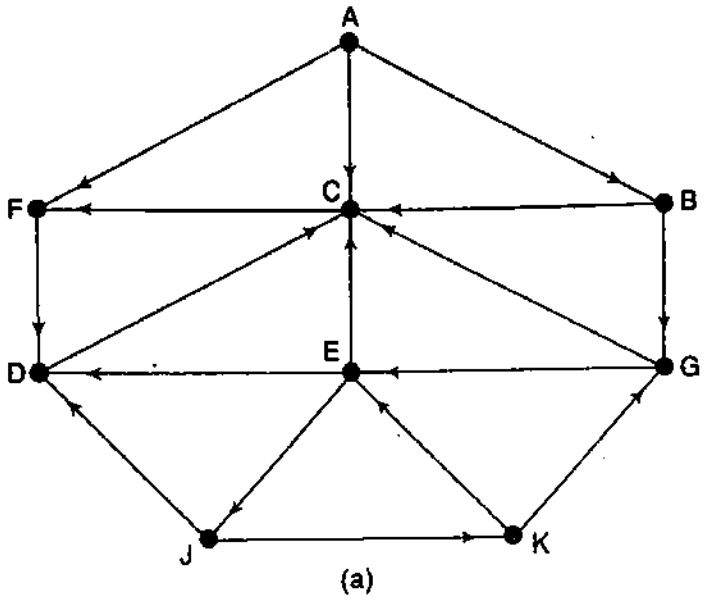
[End of Step 3 loop.]

6. Exit.

The above algorithm will process only those nodes which are reachable from the starting node A. Suppose one wants to examine all the nodes in the graph G. Then the algorithm must be modified so that it begins again with another node (which we will call B) that is still in the ready state. This node B can be obtained by traversing the list of nodes.

Example 8.7

Consider the graph G in Fig. 8.14(a). (The adjacency lists of the nodes appear in Fig. 8.14(b).) Suppose G represents the daily flights between cities of some airline, and suppose we want to fly from city A to city J with the minimum number of stops. In other words, we want the minimum path P from A to J (where each edge has length 1).



(a)

Adjacency lists	
A:	F, C, B
B:	G, C
C:	F
D:	C
E:	D, C, J
F:	D
G:	C, E
J:	D, K
K:	E, G

(b)

Fig. 8.14

The minimum path P can be found by using a breadth-first search beginning at city A and ending when J is encountered. During the execution of the search, we will also keep track of the origin of each edge by using an array ORIG together with the array QUEUE. The steps of our search follow.

(a) Initially, add A to QUEUE and add NULL to ORIG as follows:

FRONT = 1 QUEUE: A
REAR = 1 ORIG : Ø

(b) Remove the front element A from QUEUE by setting FRONT := FRONT + 1, and add to QUEUE the neighbors of A as follows:

FRONT = 2 QUEUE: A, F, C, B
REAR = 4 ORIG : Ø, A, A, A

Note that the origin A of each of the three edges is added to ORIG.

(c) Remove the front element F from QUEUE by setting FRONT := FRONT + 1, and add to QUEUE the neighbors of F as follows:

FRONT = 3 QUEUE: A, F, C, B, D
REAR = 5 ORIG : Ø, A, A, A, F

(d) Remove the front element C from QUEUE, and add to QUEUE the neighbors of C (which are in the ready state) as follows:

FRONT = 4 QUEUE: A, F, C, B, D
REAR = 5 ORIG : Ø, A, A, A, F

Note that the neighbor F of C is not added to QUEUE, since F is not in the ready state (because F has already been added to QUEUE).

(e) Remove the front element B from QUEUE, and add to QUEUE the neighbors of B (the ones in the ready state) as follows:

FRONT = 5 QUEUE: A, F, C, B, D, G
REAR = 6 ORIG : Ø, A, A, A, F, B

Note that only G is added to QUEUE, since the other neighbor, C is not in the ready state.

(f) Remove the front element D from QUEUE, and add to QUEUE the neighbors of D (the ones in the ready state) as follows:

FRONT = 6 QUEUE: A, F, C, B, D, G
REAR = 6 ORIG : Ø, A, A, A, F, B

(g) Remove the front element G from QUEUE and add to QUEUE the neighbors of G (the ones in the ready state) as follows:

FRONT = 7 QUEUE: A, F, C, B, D, G, E
REAR = 7 ORIG : Ø, A, A, A, F, B, G

(h) Remove the front element E from QUEUE and add to QUEUE the neighbors of E (the ones in the ready state) as follows:

FRONT = 8 QUEUE: A, F, C, B, D, G, E, J
REAR = 8 ORIG : Ø, A, A, A, F, B, G, E

We stop as soon as J is added to QUEUE, since J is our final destination. We now backtrack from J, using the array ORIG to find the path *P*. Thus

$$J \leftarrow E \leftarrow G \leftarrow B \leftarrow A$$

is the required path *P*.

Depth-First Search

The general idea behind a depth-first search beginning at a starting node A is as follows. First we examine the starting node A. Then we examine each node N along a path *P* which begins at A; that is, we process a neighbor of A, then a neighbor of a neighbor of A, and so on. After coming to a "dead end," that is, to the end of the path *P*, we backtrack on *P* until we can continue along another, path *P'*. And so on. (This algorithm is similar to the inorder traversal of a binary tree, and the algorithm is also similar to the way one might travel through a maze.) The algorithm is very similar to the breadth-first search except now we use a stack instead of the queue. Again, a field STATUS is used to tell us the current status of a node. The algorithm follows.

Algorithm B: This algorithm executes a depth-first search on a graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).
2. Push the starting node A onto STACK and change its status to the waiting state (STATUS = 2).
3. Repeat Steps 4 and 5 until STACK is empty.
4. Pop the top node N of STACK. Process N and change its status to the processed state (STATUS = 3).
5. Push onto STACK all the neighbors of N that are still in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).

   [End of Step 3 loop.]
6. Exit.

Again, the above algorithm will process only those nodes which are reachable from the starting node A. Suppose one wants to examine all the nodes in *G*. Then the algorithm must be modified so that it begins again with another node which we will call B—that is still in the ready state. This node B can be obtained by traversing the list of nodes.

Example 8.8

Consider the graph *G* in Fig. 8.14(a). Suppose we want to find and print all the nodes reachable from the node J (including J itself). One way to do this is to use a depth-first search of *G* starting at the node J. The steps of our search follow.

(a) Initially, push J onto the stack as follows:

STACK: J

(b) Pop and print the top element J, and then push onto the stack all the neighbors of J (those that are in the ready state) as follows:

Print J STACK: D, K

(c) Pop and print the top element K, and then push onto the stack all the neighbors of K (those that are in the ready state) as follows:

Print K STACK: D, E, G

(d) Pop and print the top element G, and then push onto the stack all the neighbors of G (those in the ready state) as follows:

Print G STACK: D, E, C

Note that only C is pushed onto the stack, since the other neighbor, E, is not in the ready state (because E has already been pushed onto the stack).

(e) Pop and print the top element C, and then push onto the stack all the neighbors of C (those in the ready state) as follows:

Print C STACK: D, E, F

(f) Pop and print the top element F, and then push onto the stack all the neighbors of F (those in the ready state) as follows:

Print F STACK: D, E

Note that the only neighbor D of F is not pushed onto the stack, since D is not in the ready state (because D has already been pushed onto the stack).

(g) Pop and print the top element E, and push onto the stack all the neighbors of E (those in the ready state) as follows:

Print E STACK: D

(Note that none of the three neighbors of E is in the ready state.)

(h) Pop and print the top element D, and push onto the stack all the neighbors of D (those in the ready state) as follows:

Print D STACK:

The stack is now empty, so the depth-first search of G starting at J is now complete. Accordingly, the nodes which were printed,

J, K, G, C, F, E, D

are precisely the nodes which are reachable from J.

8.8 POSETS; TOPOLOGICAL SORTING

Suppose S is a graph such that each node v_i of S represents a task and each edge (u, v) means that the completion of the task u is a prerequisite for starting the task v. Suppose such a graph S contains a cycle, such as

$$P = (u, v, w, u)$$

This means that we cannot begin v until completing u, we cannot begin w until completing v and we cannot begin u until completing w. Thus we cannot complete any of the tasks in the cycle. Accordingly, such a graph S, representing tasks and a prerequisite relation, cannot have cycles.

Suppose S is a graph without cycles. Consider the relation $<$ on S defined as follows:

$$u < v \qquad \text{if there is a path from } u \text{ to } v$$

This relation has the following three properties:

(1) For each element u in S, we have $u \nless u$. (Irreflexivity.)
(2) If $u < v$, then $v \nless u$. (Asymmetry.)
(3) If $u < v$ and $v < w$, then $u < w$. (Transitivity.)

Such a relation $<$ on S is called a *partial ordering* of S, and S with such an ordering is called a *partially ordered set*, or *poset*. Thus a graph S without cycles may be regarded as a partially ordered set.

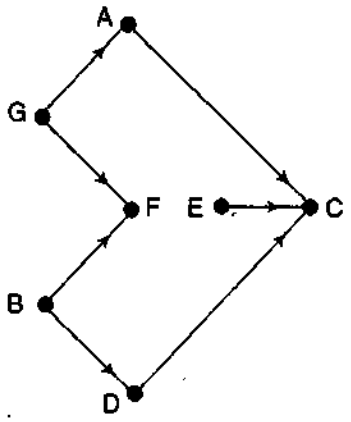
On the other hand, suppose S is a partially ordered set with the partial ordering denoted by $<$. Then S may be viewed as a graph whose nodes are the elements of S and whose edges are defined as follows:

(u, v) is an edge in S if $u < v$

Furthermore, one can show that a partially ordered set S, regarded as a graph, has no cycles.

Example 8.9

Let S be the graph in Fig. 8.15. Observe that S has no cycles. Thus S may be regarded as a partially ordered set. Note that G < C, since there is a path from G to C. Similarly, B < F and B < C. On the other hand, B ≮ A, since there is no path from B to A. Also, A ≮ B.



(a)

Adjacency lists	
A:	C
B:	D, F
C:	
D:	C
E:	C
F:	
G:	A, F

(b)

Fig. 8.15

Topological Sorting

Let S be a directed graph without cycles (or a partially ordered set). A topological sort T of S is a linear ordering of the nodes of S which preserves the original partial ordering of S. That is: *If $u < v$ in S (i.e., if there is a path from u to v in S), then u comes before v in the linear ordering T.* Figure 8.16 shows two different topological sorts of the graph S in Fig. 8.15. We have included the edges in Fig. 8.16 to indicate that they agree with the direction of the linear ordering.

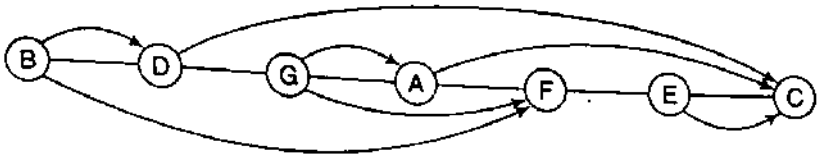
The following is the main theoretical result in this section.

Proposition 8.4

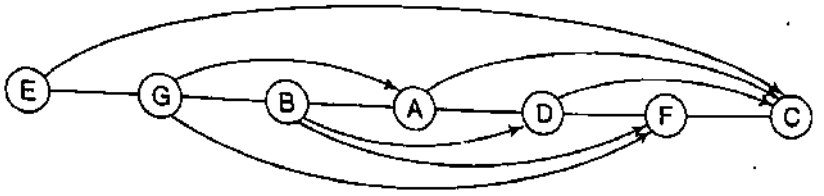
Let S be a finite directed graph without cycles or a finite partially ordered set. Then there exists a topological sort T of the set S.

Note that the proposition states only that a topological sort exists. We now give an algorithm which will find such a topological sort.

The main idea behind our algorithm to find a topological sort T of a graph S without cycles is that any node N with zero indegree, i.e., without any predecessors, may be chosen as the first element in the sort T. Accordingly, our algorithm will repeat the following two steps until the graph S is empty:



(a)



(b)

Fig. 8.16

(1) Finding a node N with zero indegree
(2) Deleting N and its edges from the graph S

The order in which the nodes are deleted from the graph S will use an auxiliary array QUEUE which will temporarily hold all the nodes with zero indegree. The algorithm also uses a field INDEG such that INDEG(N) will contain the current indegree of the node N. The algorithm follows.

Algorithm C: This algorithm finds a topological sort T of a graph S without cycles.

1. Find the indegree INDEG(N) of each node N of S. (This can be done by traversing each adjacency list as in Problem 8.15.)
2. Put in a queue all the nodes with zero indegree.
3. Repeat Steps 4 and 5 until the queue is empty.
4. Remove the front node N of the queue (by setting FRONT := FRONT + 1).
5. Repeat the following for each neighbor M of the node N:
   (a) Set INDEG(M) := INDEG(M) – 1.
   [This deletes the edge from N to M.]
   (b) If INDEG(M) = 0, then: Add M to the rear of the queue.
   [End of loop.]
   [End of Step 3 loop.]
6. Exit.

Example 8.10

Consider the graph S in Fig. 8.15(a). We apply our Algorithm C to find a topological sort T of the graph S. The steps of the algorithm follow.

1. Find the indegree INDEG(N) of each node N of the graph S. This yields:

INDEG(A) = 1 INDEG(B) = 0 INDEG(C) = 3 INDEG(D) = 1
INDEG(E) = 0 INDEG(F) = 2 INDEG(G) = 0

[This can be done as in Problem 8.15.]

2. Initially add to the queue each node with zero indegree as follows:

FRONT = 1, REAR = 3, QUEUE: B, E, G

3a. Remove the front element B from the queue by setting FRONT := FRONT + 1, as follows:

FRONT = 2, REAR = 3 QUEUE: B, E, G

3b. Decrease by 1 the indegree of each neighbor of B, as follows:

INDEG(D) = 1 - 1 = 0 and INDEG(F) = 2 - 1 = 1

[The adjacency list of B in Fig. 8.15(b) is used to find the neighbors D and F of the node B.] The neighbor D is added to the rear of the queue, since its indegree is now zero:

FRONT = 2, REAR = 4 QUEUE: B, E, G, D

[The graph S now looks like Fig. 8.17(a), where the node B and the edges from B have been deleted, as indicated by the dotted lines.]

4a. Remove the front element E from the queue by setting FRONT := FRONT + 1, as follows:

FRONT = 3, REAR = 4 QUEUE: B, E, G, D

4b. Decrease by 1 the indegree of each neighbor of E, as follows:

INDEG(C) = 3 - 1 = 2

[Since the indegree is nonzero, QUEUE is not changed. The graph S now looks like Fig. 8.17(b), where the node E and its edge have been deleted.]

5a. Remove the front element G from the queue by setting FRONT := FRONT + 1, as follows:

FRONT = 4, REAR = 4 QUEUE: B, E, G, D

5b. Decrease by 1 the indegree of each neighbor of G, as follows:

INDEG(A) = 1 - 1 = 0 and INDEG(F) = 1 - 1 = 0

Both A and F are added to the rear of the queue, as follows:

FRONT = 4, REAR = 6 QUEUE: B, E, G, D, A, F

[The graph S now looks like Fig. 8.17(c), where G and its two edges have been deleted.]

6a. Remove the front element D from the queue by setting FRONT := FRONT + 1, as follows:

FRONT = 5, REAR=6 QUEUE: B, E, G, D, A, F

6b. Decrease by 1 the indegree of each neighbor of D, as follows:

INDEG(C) = 2 - 1 = 1

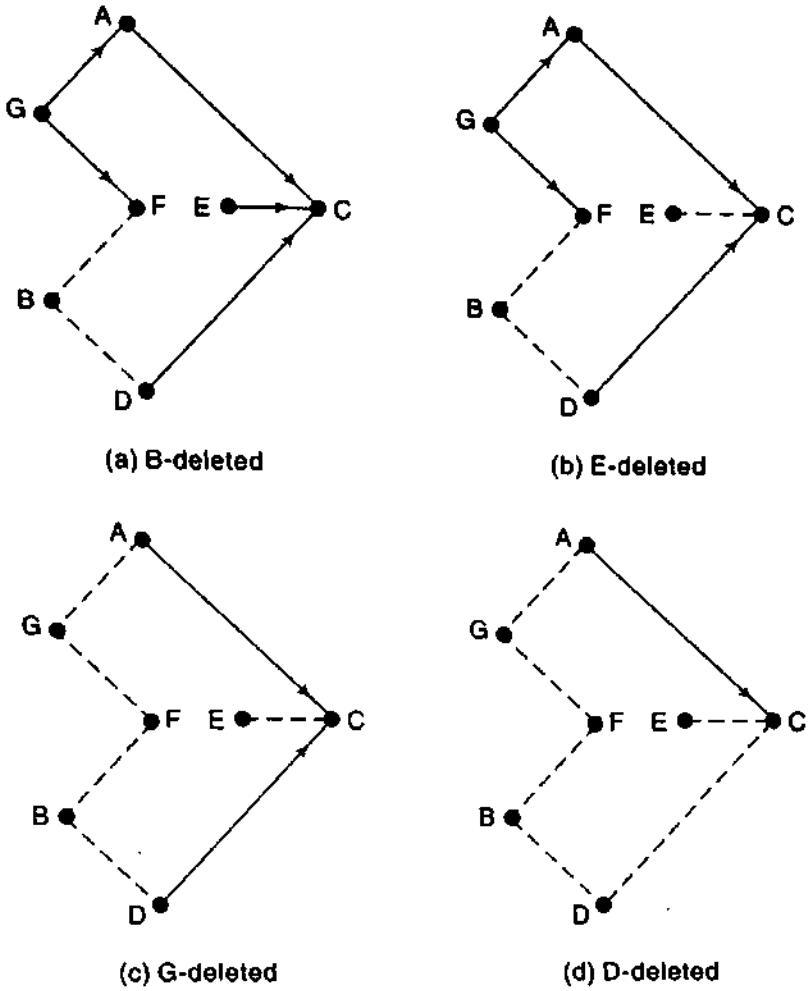


Fig. 8.17

[Since the indegree is nonzero, QUEUE is not changed. The graph S now looks like Fig. 8.17(d), where D and its edge have been deleted.]

7a. Remove the front element A from the queue by setting FRONT := FRONT + 1, as follows:

FRONT = 6, REAR = 6 QUEUE: B, E, G, D, A, F

7b. Decrease by 1 the indegree of each neighbor of A, as follows:

INDEG(C) = 1 – 1 = 0

Add C to the rear of the queue, since its indegree is now zero:

FRONT = 6, REAR = 7 QUEUE: B, E, G, D, A, F, C

8a. Remove the front element F from the queue by setting FRONT := FRONT + 1, as follows:

FRONT = 7, REAR = 7 QUEUE: B, E, G, D, A, F, C

8b. The node F has no neighbors, so no change takes place.

9a. Remove the front element C from the queue by setting FRONT := FRONT + 1 as follows:

FRONT = 8, REAR = 7 QUEUE: B, E, G, D, A, F, C

9b. The node C has no neighbors, so no other changes take place.

The queue now has no front element, so the algorithm is completed. The elements in the array QUEUE give the required topological sort T of S as follows:

T: B, E, G, D, A, F, C

The algorithm could have stopped in Step 7b, where REAR is equal to the number of nodes in the graph S.

SOLVED PROBLEMS

Graph Terminology

8.1 Consider the (undirected) graph G in Fig. 8.18. (a) Describe G formally in terms of its set V of nodes and its set E of edges. (b) Find the degree of each node.

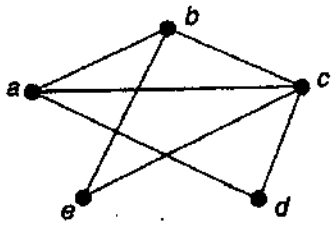


Fig. 8.18

(a) There are 5 nodes, a, b, c, d and e; hence $V = \{a, b, c, d, e\}$. There are 7 pairs $[x, y]$ of nodes such that node x is connected with node y; hence

$$E = \{[a, b], [a, c], [a, d], [b, c], [b, e], [c, d], [c, e]\}$$

(b) The degree of a node is equal to the number of edges to which it belongs; for example, $\deg(a) = 3$, since a belongs to three edges, $[a, b]$, $[a, c]$ and $[a, d]$. Similarly, $\deg(b) = 3$, $\deg(c) = 4$, $\deg(d) = 2$ and $\deg(e) = 2$.

8.2 Consider the multigraphs in Fig. 8.19. Which of them are (a) connected; (b) loop-free (i.e., without loops); (c) graphs?

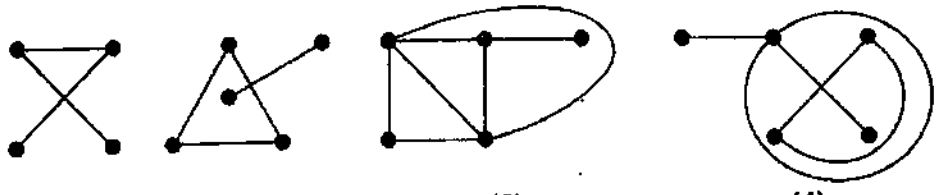


Fig. 8.19

(a) Only multigraphs 1 and 3 are connected.
(b) Only multigraph 4 has a loop (i.e., an edge with the same endpoints).
(c) Only multigraphs 1 and 2 are graphs. Multigraph 3 has multiple edges, and multigraph 4 has multiple edges and a loop.

8.3 Consider the connected graph G in Fig. 8.20. (a) Find all simple paths from node A to node F. (b) Find the distance between A and F. (c) Find the diameter of G. (The diameter of G is the maximum distance existing between any two of its nodes.)

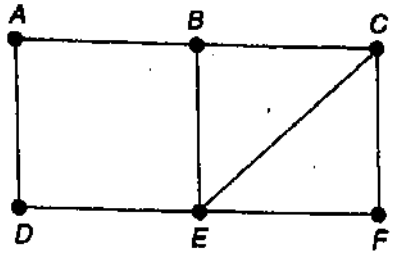


Fig. 8.20

(a) A simple path from A to F is a path such that no node and hence no edge is repeated. There are seven such simple paths:

$$(A, B, C, F) \quad (A, B, E, F) \quad (A, D, E, F) \quad (A, D, E, C, F)$$
$$(A, B, C, E, F) \quad (A, B, E, C, F) \quad (A, D, E, B, C, F)$$

(b) The distance from A to F equals 3, since there is a simple path, (A, B, C, F), from A to F of length 3 and there is no shorter path from A to F.
(c) The distance between A and F equals 3, and the distance between any two nodes does not exceed 3; hence the diameter of the graph G equals 3.

8.4 Consider the (directed) graph G in Fig. 8.21. (a) Find all the simple paths from X to Z. (b) Find all the simple paths from Y to Z. (c) Find indeg(Y) and outdeg(Y). (d) Are there any sources or sinks?

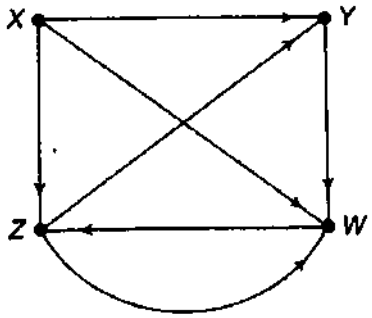


Fig. 8.21

(a) There are three simple paths from X to Z: (X, Z), (X, W, Z) and (X, Y, W, Z).
(b) There is only one simple path from Y to Z: (Y, W, Z).

(c) Since two edges enter Y (i.e., end at Y), we have indeg(Y) = 2. Since only one edge leaves Y (i.e., begins at Y), outdeg(Y) = 1.

(d) X is a source, since no edge enters X (i.e., indeg(X) = 0) but some edges leave X (i. e., outdeg(X) > 0). There are no sinks, since each node has a nonzero outdegree (i.e., each node is the initial point of some edge).

8.5 Draw all (nonsimilar) trees with exactly 6 nodes. (A graph G is *similar* to a graph G' if there is a one-to-one correspondence between the set V of nodes of G and the set V' of nodes of G' such that (u, v) is an edge in G if and only if the corresponding pair (u', v') of nodes is an edge in G'.)

There are six such trees, which are exhibited in Fig. 8.22. The first tree has diameter 5, the next two diameter 4, the next two diameter 3 and the last one diameter 2. Any other tree with 6 nodes will be similar to one of these trees.

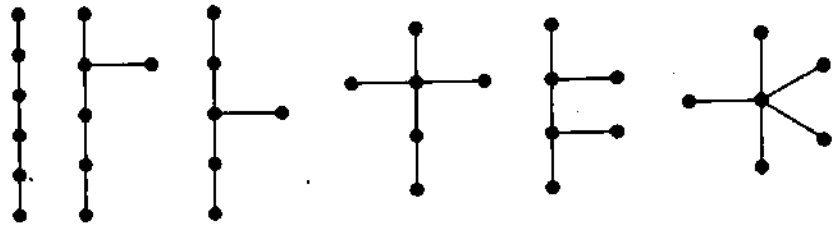


Fig. 8.22

8.6 Find all spanning trees of the graph G shown in Fig. 8.23(a). (A tree T is called a *spanning tree* of a connected graph G if T has the same nodes as G and all the edges of T are contained among the edges of G.)

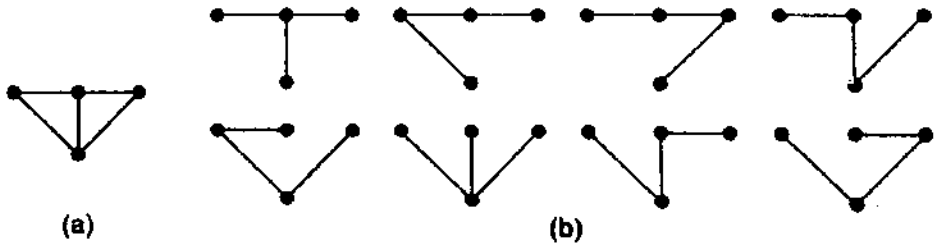


Fig. 8.23

There are eight such spanning trees, as shown in Fig. 8.23(b). Since G has 4 nodes, each spanning tree T must have 4 − 1 = 3 edges. Thus each spanning tree can be obtained by deleting 2 of the 5 edges of G. This can be done in 10 ways, except that two of them lead to disconnected graphs. Hence the eight spanning trees shown are all the spanning trees of G.

Sequential Representation of Graphs

8.7 Consider the graph G in Fig. 8.21. Suppose the nodes are stored in memory in an array DATA as follows:

DATA: X, Y, Z, W

(a) Find the adjacency matrix A of the graph G.
(b) Find the path matrix P of G using powers of the adjacency matrix A.
(c) Is G strongly connected?

(a) The nodes are normally ordered according to the way they appear in memory; that is, we assume $v_1 = X$, $v_2 = Y$ $v_3 = Z$ and $v_4 = W$. The adjacency matrix A of G follows:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Here $a_{ij} = 1$ if there is a node from v_i to v_j otherwise, $a_{ij} = 0$.

(b) Since G has 4 nodes, compute A^2, A^3, A^4 and $B_4 = A + A^2 + A^3 + A^4$:

$$A^2 = \begin{pmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \qquad A^3 = \begin{pmatrix} 0 & 1 & 2 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 2 & 2 & 3 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 1 \end{pmatrix} \qquad B_4 = \begin{pmatrix} 0 & 5 & 6 & 8 \\ 0 & 1 & 2 & 3 \\ 0 & 3 & 3 & 5 \\ 0 & 2 & 3 & 5 \end{pmatrix}$$

The path matrix P is now obtained by setting $p_{ij} = 1$ wherever there is a nonzero entry in the matrix B_4. Thus

$$P = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

(c) The path matrix shows that there is no path from v_2 to v_1. In fact, there is no path from any node to v_1. Thus G is not strongly connected.

8.8 Consider the graph G in Fig. 8.21 and its adjacency matrix A obtained in Problem 8.7. Find the path matrix P of G using Warshall's algorithm rather than the powers of A.

Compute the matrices P_0, P_1, P_2, P_3 and P_4 where initially $P_0 = A$ and

$$P_k[i, j] = P_{k-1}[i, j] \vee (P_{k-1}[i, j] \wedge P_{k-1}[k, j])$$

That is,

$$P_k[i, j] = 1 \quad \text{if} \quad P_{k-1}[i, j] = 1 \quad \text{or both} \quad P_{k-1}[i, k] = 1 \quad \text{and} \quad P_{k-1}[k, j] = 1$$

Then:

$$P_1 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad P_2 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$P_3 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \qquad P_4 = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Observe that $P_0 = P_1 = P_2 = A$. The changes in P_3 occur for the following reasons:

$P_3(4, 2) = 1$	because	$P_2(4, 3) = 1$	and	$P_2(3, 2) = 1$
$P_3(4, 4) = 1$	because	$P_2(4, 3) = 1$	and	$P_2(3, 4) = 1$

The changes in P_4 occur similarly. The last matrix, P_4, is the required path matrix P of the graph G.

8.9 Consider the (undirected) weighted graph G in Fig. 8.24. Suppose the nodes are stored in memory in an array DATA as follows:

DATA: A, B, C, X, Y

Find the weight matrix $W = (w_{ij})$ of the graph G.

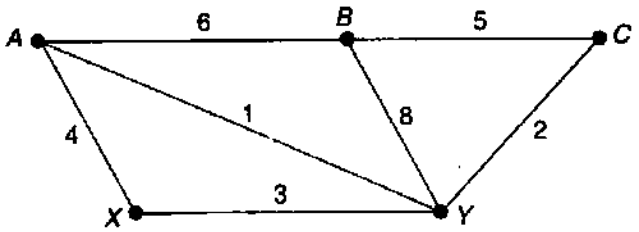


Fig. 8.24

Assuming v_1 = A, v_2 = B, v_3 = C, v_4 = X and v_5 = Y, we arrive at the following weight matrix W of G:

$$W = \begin{pmatrix} 0 & 6 & 0 & 4 & 1 \\ 6 & 0 & 5 & 0 & 8 \\ 0 & 5 & 0 & 0 & 2 \\ 4 & 0 & 0 & 0 & 3 \\ 1 & 8 & 2 & 3 & 0 \end{pmatrix}$$

Here w_{ij} denotes the weight of the edge from v_i to v_j. Since G is undirected, W is a symmetric matrix, that is, $w_{ij} = w_{ji}$.

8.10 Suppose G is a graph (undirected) which is cycle-free, that is, without cycles. Let $P = (P_{ij})$ be the path matrix of G.

(a) When can an edge $[v_i, v_j]$ be added to G so that G is still cycle-free?
(b) How does the path matrix P change when an edge $[v_i, v_j]$ is added to G?

(a) The edge $[v_i, v_j$ will form a cycle when it is added to G if and only if there already is a path between v_i and v_j. Hence the edge may be added to G when $P_{ij} = 0$.

(b) First set $p_{ij} = 1$, since the edge is a path from v_i to v_j. Also, set $p_{st} = 1$ if $p_{si} = 1$ and $p_{jt} = 1$. In other words, if there are both a path P_1 from v_s to v_i and a path P_2 from v_j to v_t, then P_1, $[v_i, v_j]$, P_2 will form a path from v_s to v_t.

8.11 A minimum spanning tree T of a weighted graph G is a spanning tree of G (see Problem 8.6) which has the minimum weight among all the spanning trees of G.

(a) Describe an algorithm to find a minimum spanning tree T of a weighted graph G.
(b) Find a minimum spanning tree T of the graph in Fig. 8.24.

(a) **Algorithm P8.11:** This algorithm finds a minimum spanning tree T of a weighted graph G.

1. Order all the edges of G according to increasing weights.
2. Initialize T to be a graph consisting of the same nodes as G and no edges.
3. Repeat the following M – 1 times, where M is the number of nodes in G:
   Add to T an edge E of G with minimum weight such that E does not form a cycle in T.
   [End of loop.]
4. Exit.

Step 3 may be implemented using the results of Solved Problem. 8.10. Problem 8.10(a) tells us which edge e may be added to T so that no cycle is formed—i.e., so that T is still cycle-free—and Problem 8.10(b) tells us how to keep track of the path matrix P of T as each edge e is added to T.

(b) Apply Algorithm P8.11 to obtain the minimum spanning tree T in Fig. 8.25. Although [A, X] has less weight than [B, C), we cannot add [A, X] to T, since it would form a cycle with [A, Y] and [Y, X].

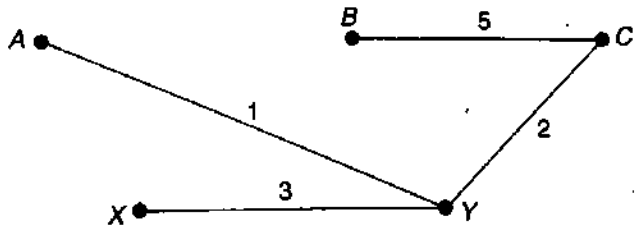


Fig. 8.25

8.12 Suppose a weighted graph G is maintained in memory by a node array DATA and a weight matrix W as follows:

DATA: X, Y, S, T

$$W = \begin{pmatrix} 0 & 0 & 3 & 0 \\ 5 & 0 & 1 & 7 \\ 2 & 0 & 0 & 4 \\ 0 & 6 & 8 & 0 \end{pmatrix}$$

Draw a picture of G.

The picture appears in Fig. 8.26. The nodes are labeled by the entries in DATA. That is, if $w_{ij} \neq 0$, then there is an edge from v_i to v_j with weight w_{ij}. (We assume v_1 = X, v_2 = Y, v_3 = S and v_4 = T, the order in which the nodes appear in the array DATA.)

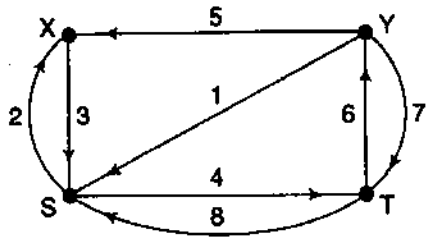


Fig. 8.26

Linked Representation of Graphs

8.13 A graph G is stored in memory as follows:

NODE	A	B		E		D	C	
NEXT	7	4	0	6	8	0	2	3
ADJ	1	2		5		7	9	
	1	2	3	4	5	6	7	8

START = 1, AVAILN = 5

DEST	2	6	4		6	7	4		4	6
LINK	10	3	6	0	0	0	0	4	0	0
	1	2	3	4	5	6	7	8	9	10

AVAILE = 8

Draw the graph G.

First find the neighbors of each NODE[K] by traversing its adjacency list, which has the pointer ADJ[K]. This yields:

A: 2(B) and 6(D) C: 4(E) E: 6(D)

B: 6(D), 4(E) and 7(C) D: 4(E)

Then draw the diagram as in Fig. 8.27.

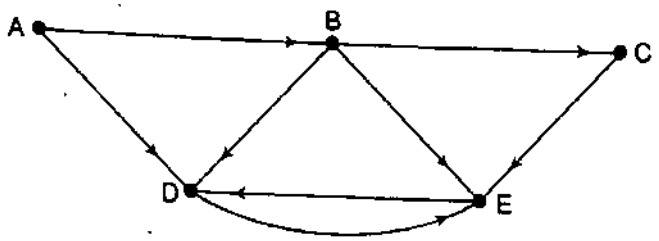


Fig. 8.27

8.14 Find the changes in the linked representation of the graph G in Problem 8.13 if the following operations occur: (a) Node F is added to G. (b) Edge (B, E) is deleted from G. (c) Edge (A, F) is added to G. Draw the resultant graph G.

(a) The node list is not sorted, so F is inserted at the beginning of the list, using the first available free node as follows:

START = 5
AVAILN = 8

	1	2	3	4	5	6	7	8
NODE	A	B		E	F	D	C	
NEXT	7	4	0	6	1	0	2	3
ADJ	1	2		5	0	7	9	

Observe that the edge list does not change.

(b) Delete LOC = 4 of node E from the adjacency list of node B as follows:

AVAILE = 3

	1	2	3	4	5	6	7	8	9	10
DEST	2	6			6	7	4		4	6
LINK	10	6	8	0	0	0	0	4	0	0

Observe that the node list does not change.

(c) The location LOC = 5 of the node F is inserted at the beginning of the adjacency list of the node A, using the first available free edge. The changes are as follows:

ADJ[1] = 3
AVAILE = 8

	1	2	3	4	5	6	7	8	9	10
DEST	2	6	5		6	7	4		4	6
LINK	10	6	1	0	0	0	0	4	0	0

The only change in the node list is the ADJ[1] = 3. (Observe that the shading indicates the changes in the lists.) The updated graph G appears in Fig. 8.28.

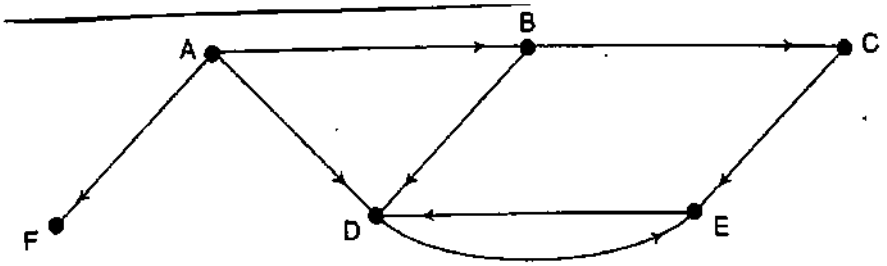


Fig. 8.28

8.15 Suppose a graph G is maintained in memory in the form

GRAPH(NODE, NEXT, ADJ, START, DEST, LINK)

Write a procedure which finds the indegree INDEG and the outdegree OUTDEG of each node of G.

First we traverse the node list, using the pointer PTR in order to initialize the arrays INDEG and OUTDEG to zero. Then we traverse the node list, using the pointer PTRA, and for each value of PTRA, we traverse the list of neighbors of NODE[PTRA], using the pointer PTRB. Each time an edge is encountered, PTRA gives the location of its initial node and DEST[PTRB] gives the location of its terminal node. Accordingly, each edge updates the arrays INDEG and OUTDEG as follows:

OUTDEG[PTRA] := OUTDEG[PTRA] + 1

and

INDEG[DEST[PTRB]] := INDEG[DEST[PTRB]] + 1

The formal procedure follows.

Procedure P8.15: DEGREE(NODE, NEXT, ADJ, START, DEST, LINK, INDEG, OUTDEG)
This procedure finds the indegree INDEG and outdegree OUTDEG of each node in the graph G in memory.

1. [Initialize arrays INDEG and OUTDEG.]
   (a) Set PTR := START.
   (b) Repeat while PTR ≠ NULL: [Traverses node list.]
      (i) Set INDEG[PTR] := 0 and OUTDEG[PTR] := 0.
      (ii) Set PTR := NEXT[PTR].
   [End of loop.]
2. Set PTRA := START.
3. Repeat Steps 4 to 6 while PTRA ≠ NULL: [Traverses node list.]
4. Set PTRB := ADJ[PTRA].
5. Repeat while PTRB ≠ NULL: [Traverses list of neighbors.]
   (a) Set OUTDEG[PTRA] := OUTDEG[PTRA] + 1 and
   INDEG[DEST[PTRB]] :=
   INDEG[DEST[PTRB]] + 1.
   (b) Set PTRB := LINK[PTRB].
   [End of inner loop using pointer PTRB.]
6. Set PTRA := NEXT[PTRA].
   [End of Step 3 outer loop using the pointer PTRA.]
7. Return.

8.16 Suppose G is a finite undirected graph. Then G consists of a finite number of disjoint connected components. Describe an algorithm which finds the number NCOMP of connected components of G. Furthermore, the algorithm should assign a component number COMP(N) to every node N in the same connected component of G such that the component numbers range from 1 to NCOMP.

The general idea of the algorithm is to use a breadth-first or depth-first search to find all nodes N reachable from a starting node A and to assign them the same component number. The algorithm follows.

Algorithm P8.16: Finds the connected components of an undirected graph G.

1. Initially set COMP(N) := 0 for every node N in G, and initially set L := 0.
2. Find a node A such that COMP(A) = 0. If no such node A exists, then:

   Set NCOMP := L, and Exit.

   Else:

   Set L := L + 1 and set COMP(A) := L.
3. Find all nodes N in G which are reachable from A (using a breadth-first search or a depth-first search) and set COMP(N) = L for each such node N.
4. Return to Step 2.

Linked Representation of Graphs

8.17 Suppose G is an undirected graph with m nodes $v_1, v_2, \ldots, v_m$ and n edges $e_1, e_2, \ldots, e_n$. The *incidence matrix* of G is the $m \times n$ matrix $M = (m_{ij})$ where

$$m_{ij} = \begin{cases} 1 & \text{if node } v_i \text{ belongs to edge } e_j \\ 0 & \text{otherwise} \end{cases}$$

Find the incidence matrix M of the graph G in Fig. 8.29.

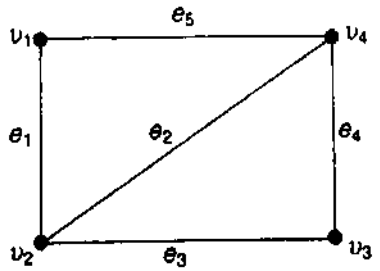


Fig. 8.29

Since G has 4 nodes and 5 edges, M is a 4×5 matrix. Set $m_{ij} = 1$ if v_i belongs to e_j. This yields the following matrix M:

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

8.18 Suppose u and v are distinct nodes in an undirected graph G. Prove:

(a) If there is a path P from u to v, then there is a simple path Q from u to v.
(b) If there are two distinct paths P_1 and P_2 from u to v, then G contains a cycle.

(a) Suppose $P = (v_0, v_1 \ldots, v_n)$ where $u = v_0$ and $v = v_n$. If $v_i = v_j$, then

$$P' = (v_0 \ldots, v_i, v_{j+1} \ldots, v_n)$$

is a path from u to v which is shorter than P. Repeating this process, we finally obtain a path Q from u to v whose nodes are distinct. Thus Q is a simple path from u to v.

(b) Let w be a node in P_1 and P_2 such that the next nodes in P_1 and P_2 are distinct. Let w' be the first node following w which lies on both P_1 and P_2. (See Fig. 8.30.) Then the subpaths of P_1 and P_2 between w and w' have no nodes in common except w and w'; hence these two subpaths form a cycle.

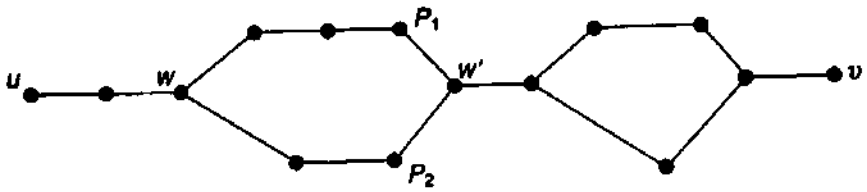


Fig. 8.30

8.19 Prove Proposition 8.2: Let A be the adjacency matrix of a graph G. Then $a_K(i, j)$, the ij entry in the matrix A^K, gives the number of paths of length K from v_i to v_j.

The proof is by induction on K. Note first that a path of length 1 from v_i to v_j is precisely an edge (v_i, v_j). By definition of the adjacency matrix A, $a_1(i, j) = a_{ij}$ gives the number of edges from v_i to v_j. Hence the proposition is true for $K = 1$.

Suppose $K > 1$. (Assume G has m nodes.) Since $A^K = A^{K-1} A$,

$$a_K(i, j) = \sum_{s=1}^{m} a_{K-1}(i, s) a_1(s, j)$$

By induction, $a_{K-1}(i, s)$ gives the number of paths of length $K - 1$ from v_i to v_s, and $a_1(s, j)$ gives the number of paths of length 1 from v_s to v_j. Thus $a_{K-1}(i, s)a_1(s, j)$ gives the number of paths of length K from v_i to v_j where v_s is the next-to-last node. Thus all the paths of length K from v_i to v_j can be obtained by summing up the $a_{K-1}(i, s)a_1(s, j)$ for all s. That is, $a_K(i, j)$ is the number of paths of length K from v_i to v_j. Thus the proposition is proved.

8.20 Suppose G is a finite undirected graph without cycles. Prove each of the following:

(a) If G has at least one edge, then G has a node v with degree 1.
(b) If G is connected—so that G is a tree—and if G has m nodes, then G has $m - 1$ edges.

(c) If G has m nodes and $m - 1$ edges, then G is a tree.

(a) Let $P = (v_0, v_1 \ldots, v_n)$ be a simple path of maximum length. Suppose $\deg(v_0) \neq 1$, and assume $[u, v_0]$ is an edge and $u \neq v_1$. If $u = v_i$ for $i > 1$, then $C = (v_i, v_0, \ldots, v_i)$ is a cycle. If $u \neq v_i$, then $P' = (u, v_0, \ldots, v_n)$ is a simple path with length greater than P. Each case leads to a contradiction. Hence $\deg(v_0) = 1$.

(b) The proof is by induction on m. Suppose $m = 1$. Then G consists of an isolated node and G has $m - 1 = 0$ edges. Hence the result is true for $m = 1$. Suppose $m > 1$. Then G has a node v such that $\deg(v) = 1$. Delete v and its only edge $[v, v']$ from the graph G to obtain the graph G'. Then G is still connected and G is a tree with $m - 1$ nodes. By induction, G' has $m - 2$ edges. Hence G has $m - 1$ edges. Thus the result is true.

(c) Let $T_1, T_2, \ldots, T_s$ denote the connected components of G. Then each T_i is a tree. Hence each T_i has one more node than edges. Hence G has s more nodes than edges. But G has only one more node than edges. Hence $s = 1$ and G is a tree.

SUPPLEMENTARY PROBLEMS

Graph Terminology

8.1 Consider the undirected graph G in Fig. 8.31. Find (*a*) all simple paths from node A to node H, (*b*) the diameter of G and (*c*) the degree of each node.

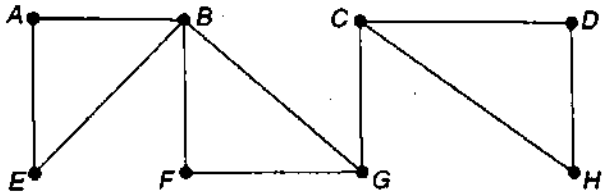


Fig. 8.31

8.2 Which of the multigraphs in Fig. 8.32 are (*a*) connected, (*b*) loop-free (i.e., without loops) and (*c*) graphs?

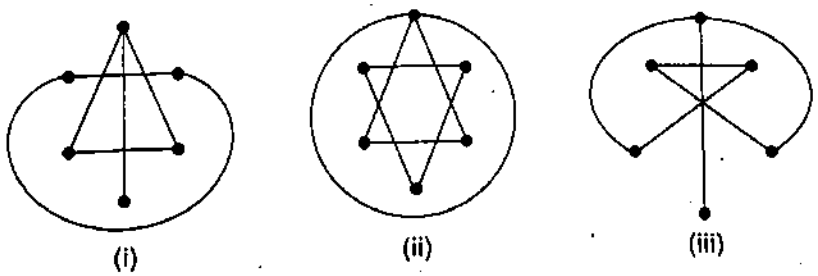


Fig. 8.32

8.3 Consider the directed graph G in Fig. 8.33. (*a*) Find the indegree and outdegree of each node. (*b*) Find the number of simple paths from v_1 to v_4. (*c*) Are there any sources or sinks?

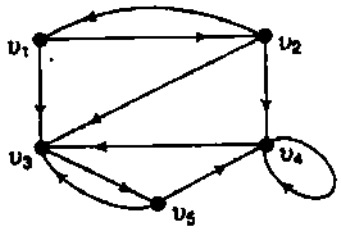


Fig. 8.33

8.4 Draw all (nonsimilar) trees with 5 or fewer nodes. (There are eight such trees.)

8.5 Find the number of spanning trees of the graph G in Fig. 8.34.

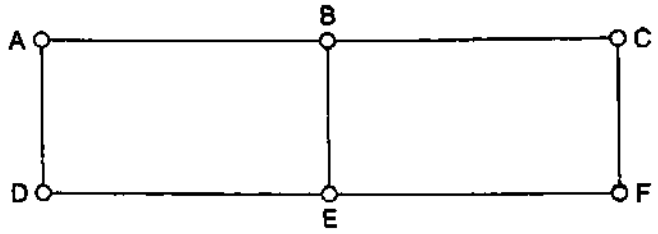


Fig. 8.34

Sequential Representation of Graphs; Weighted Graphs

8.6 Consider the graph G in Fig. 8.35. Suppose the nodes are stored in memory in an array DATA as follows:

DATA: X, Y, Z, S, T

(a) Find the adjacency matrix A of G. (b) Find the path matrix P or G. (c) Is G strongly connected?

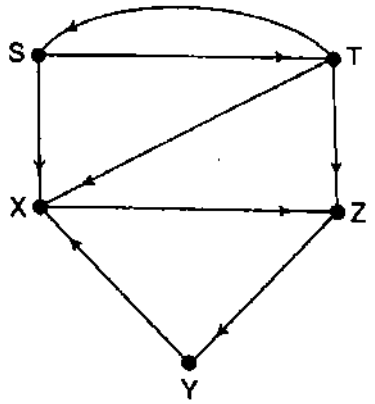


Fig. 8.35

8.7 Consider the weighted graph G in Fig. 8.36. Suppose the nodes are stored in an array DATA as follows:

DATA: X, Y, S, T

(a) Find the weight matrix W of G. (b) Find the matrix Q of shortest paths using Warshall's Algorithm 8.2.

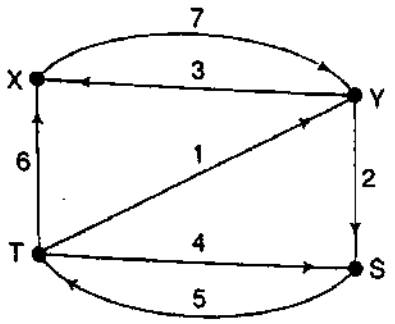


Fig. 8.36

8.8 Find a minimum spanning tree of the graph G in Fig. 8.37.

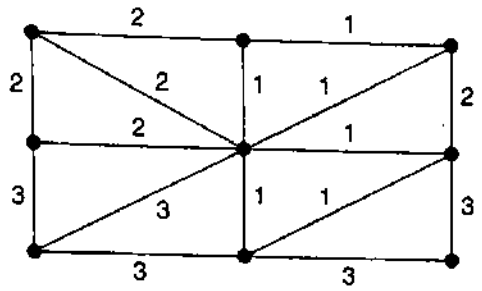


Fig. 8.37

8.9 The following is the incidence matrix M of an undirected graph G:

$$M = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

(Note that G has 5 nodes and 8 edges.) Draw G and find its adjacency matrix A.

8.10 The following is the adjacency matrix A of an undirected graph G:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(Note that G has 5 nodes.) Draw G and find its incidence matrix M.

Linked Representation of Graphs

8.11 Suppose a graph G is stored in memory as follows:

	1	2	3	4	5	6	7	8
NODE	A		C	E		D		B
NEXT	4	0	8	0	7	3	2	1
ADJ	6		1	10		2		9

START = 6, AVAILN = 5

	1	2	3	4	5	6	7	8	9	10
DEST	8	8		1	4	3	3		6	3
LINK	5	7	8	0	0	0	0	0	4	0

AVAILE = 3

Draw the graph G.

8.12 Find the changes in the linked representation of the graph G in Supplementary Problem 8.11 if edge (C, E) is deleted and edge (D, E) is inserted.

8.13 Find the changes in the linked representation of the graph G in Supplementary Problem 8.11 if a node F and the edges (E, F) and (F, D) are inserted into G.

8.14 Find the changes in the linked representation of the graph G in Supplementary Problem 8.11 if the node B is deleted from G.

Supplementary Problems 8.15 to 8.18 refer to a graph G which is maintained in memory by a linked representation:

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

8.15 Write a procedure to supplement each of the following:

(a) Print the list of successors of a given node ND.
(b) Print the list of predecessors of a given node ND.

8.16 Write a procedure which determines whether or not G is an undirected graph.

8.17 Write a procedure which finds the number M of nodes of G and then finds the $M \times M$ adjacency matrix A of G. (The nodes are ordered according to their order in the node list of G.)

8.18 Write a procedure which determines whether there are any sources or sinks in G.

Supplementary Problems 8.19 to 8.20 refer to a weighted graph G which is stored in memory using a linked representation as follows:

GRAPH(NODE, NEXT, ADJ, START, AVAILN, WEIGHT, DEST, LINK, AVAILE)

8.19 Write a procedure which finds the shortest path from a given node NA to a given node NB.

8.20 Write a procedure which finds the longest simple path from a given node NA to a given node NB.

PROGRAMMING PROBLEMS

8.1 Suppose a graph G is input by means of an integer M, representing the nodes 1, 2, ..., M, and a list of N ordered pairs of the integers, representing the edges of G. Write a procedure for each of the following:

(a) To find the M × M adjacency matrix A of the graph G.
(b) To use the adjacency matrix A and Warshall's algorithm to find the path matrix P of the graph G.

Test the above using the following data:

(i) M = 5; N = 8; (3, 4), (5, 3), (2, 4), (1, 5), (3, 2), (4, 2), (3, 1), (5, 1),
(ii) M = 6; N = 10; (1, 6), (2, 1), (2, 3), (3, 5), (4, 5), (4, 2), (2, 6), (5, 3), (4, 3), (6, 4)

8.2 Suppose a weighted graph G is input by means of an integer M, representing the nodes 1, 2, ..., M, and a list of N ordered triplets (a_i, b_i, w_i) of integers such that the pair (a_i, b_i) is an edge of G and w_i is its weight. Write a procedure for each of the following:

(a) To find the M × M weight matrix W of the graph G.
(b) To use the weight matrix W and Warshall's Algorithm 8.2 to find the matrix Q of shortest paths between the nodes.

Test the above using the following data:

(i) M = 4; N = 7; (1, 2, 5), (2, 4, 2), (3, 2, 3), (1, 1, 7), (4, 1, 4), (4, 3, 1). (Compare with Example 8.4.)
(ii) M = 5; N = 8; (3, 5, 3), (4, 1, 2), (5, 2, 2), (1, 5, 5), (1, 3, 1), (2, 4, 1), (3, 4, 4), (5, 4, 4).

8.3 Suppose an empty graph G is stored in memory using the linked representation

GRAPH(NODE, NEXT, ADJ, START, AVAILN, DEST, LINK, AVAILE)

Assume NODE has space for 8 nodes and DEST has space for 12 edges. Write a program which executes the following operations on G:

(a) Inputs nodes A, B, C and D
(b) Inputs edges (A, B), (A, C), (C, B), (D, A), (B, D) and (C, D)
(c) Inputs nodes E and F
(d) Inputs edges (B, E), (F, E), (D, F) and (F, B)
(e) Deletes edges (D, A) and (B, D)
(f) Deletes node A

Programming Problems 8.4 to 8.5 refer to the data in Fig. 8.38, where the cities are stored as a binary search tree.

	CITY	LEFT	RIGHT	ADJ
1	Atlanta	0	2	12
2	Boston	0	0	1
3	Houston	0	0	14
4	New York	3	8	4
5		6		
6		0		
7	Washington	0	0	10
8	Philadelphia	0	7	6
9	Denver	10	4	8
10	Chicago	1	0	2

START = 9, AVAILN = 5

	NUMBER	PRICE	ORIG	DEST	LINK
1	201	80	2	10	3
2	202	80	10	2	0
3	301	50	2	4	0
4	302	50	4	2	5
5	303	40	4	8	7
6	304	40	8	4	9
7	305	120	4	9	0
8	306	120	9	4	13
9	401	40	8	7	0
10	402	40	7	8	11
11	403	80	7	1	0
12	404	80	1	7	16
13	501	80	9	3	15
14	502	80	3	9	0
15	503	140	9	1	0
16	504	140	1	9	0
17					18
18					19
19					20
20					0

NUM = 16, AVAILE = 17

Fig. 8.38

8.4 Write a procedure with input CITYA and CITYB which finds the flight number and cost of the flight from city A to city B, if a flight exists. Test the procedure using (*a*) CITYA = Chicago, CITYB = Boston; (*b*) CITYA = Washington, CITYB = Denver; and (*c*) CITYA = New York, CITYB = Philadelphia.

8.5 Write a procedure with input CITYA and CITYB which finds the way to fly from city A to city B with a minimum number of stops, and also finds its cost. Test the procedure using (*a*) CITYA = Boston, CITYB = Houston; (*b*) CITYA = Denver, CITYB = Washington; and (*c*) CITYA = New York, CITYB = Atlanta.

8.6 Write a procedure with input CITYA and CITYB which finds the cheapest way to fly from city A to city B and also finds the cost. Test the procedure using the data in Programming Problem 8.5. (Compare the results.)

8.7 Write a procedure which deletes a record from the file given the flight number NUMB. Test the program using (*a*) NUMB = 503 and NUMB = 504 and (*b*) NUMB = 303 and NUMB = 304.

8.8 Write a procedure which inputs a record of the form

(NUMBNEW, PRICENEW, ORIGNEW, DESTNEW)

Test the procedure using the following data:

(a) NUMBNEW = 505, PRICENEW = 80, ORIGNEW = Chicago, DESTNEW = Denver
NUMBNEW = 506, PRICENEW = 80, ORIGNEW = Denver, DESTNEW = Chicago
(b) NUMBNEW = 601, PRICENEW = 70, ORIGNEW = Atlanta, DESTNEW = Miami
NUMBNEW = 602, PRICENEW = 70, ORIGNEW = Miami, DESTNEW = Atlanta

(Note that a new city may have to be inserted into the binary search tree of cities.)

8.9 Translate the topological sort algorithm into a program which sorts a graph G. Assume G is input by its set V of nodes and its set E of edges. Test the program using the nodes A, B, C, D, X, Y, Z, S and T and the edges

(a) (A, Z), (S, Z), (X, D), (B, T), (C, B), (Y, X), (Z, X), (S, C) and (Z, B)
(b) (A, Z), (D, Y), (A, X), (Y, B), (S, Y), (C, T), (X, S), (B, A), (C, S) and (X, T)
(c) (A, C), (B, Z), (Y, A), (Z, X), (D, Z), (A, S), (B, T), (Z, Y), (T, Y) and (X, A)

8.10 Write a program which finds the number of connected components of an unordered graph G and also assigns a component number to each of its nodes. Assume G is input by its set V of nodes and its set E of (undirected) edges. Test the program using the nodes A, B, C, D, X, Y, Z, S and T and the edges:

(a) [A, X], [B, T], [Y, C], [S, Z], [D, T], [A, S], [Z, A], [D, B] and [X, S]
(b) [Z, C], [D, B], [A, X], [S, C], [D, T], [X, S], [Y, B], [T, B] and [S, Z]