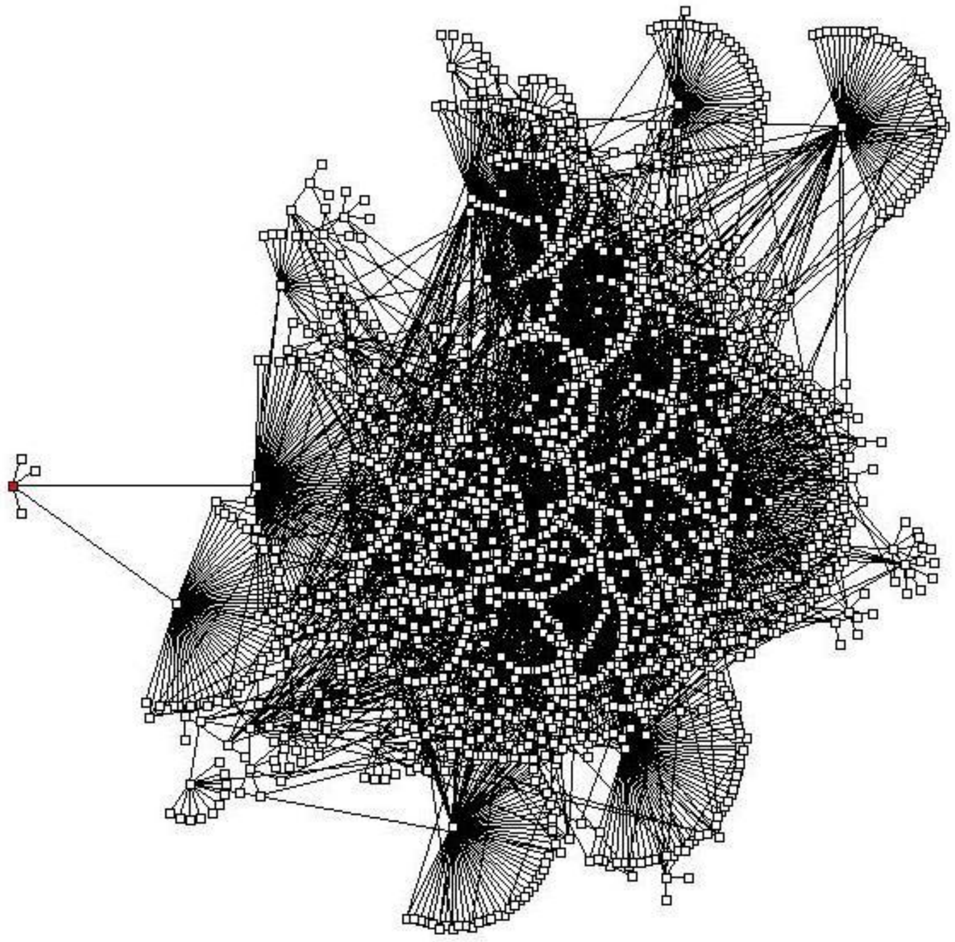
5

Graphs and Digraphs

Many problems are naturally modeled using graphs and digraphs, and data structures implementing the graph or the digraph ADT are commonly used throughout computer science. The subject of graph algorithms is a very active area of research today. Graphs and digraphs also play an important role in the Internet and other networks, such as communication, transportation, commodity flow, and so forth. The underlying structure of these networks is naturally modeled using a graph of a digraph. For example, in interstate highway system the nodes of a graph may represent cities (or junctions) and the edges may represent highways linking these cities. As a second example, the so-called *web digraph* has nodes corresponding to webpages, and a directed arc exists from web page *A* to webpage *B* if there is a hyperlink reference (href) to web page *B* in webpage *A*. As a third example, in parallel computing graphs serve as models for interconnection networks. As a fourth example, we may represent the underlying topology of networks, such as the peer-to-peer network Gnutella (see Figure 5.1). Besides network applications, graphs and digraphs serve as natural models for a host of other applications. To cite an example from computer science, the logical flow of a computer program written in a high-level language is naturally modeled by a program digraph (flow chart). Optimizing compilers then utilize various properties of this digraph, such as strongly connected components and vertex coloring, to help achieve the goal of translating the high-level code into machine code that exhibits optimal performance.



A snapshot of a portion of the Gnutella peer-to-peer network

Figure 5.1

In this chapter we provide an introduction to some basic graph theory concepts. We also discuss the two basic search strategies known as depth-first search and breadth-first search, and apply these strategies to such problems as topological sorting. and finding shortest paths in graphs.

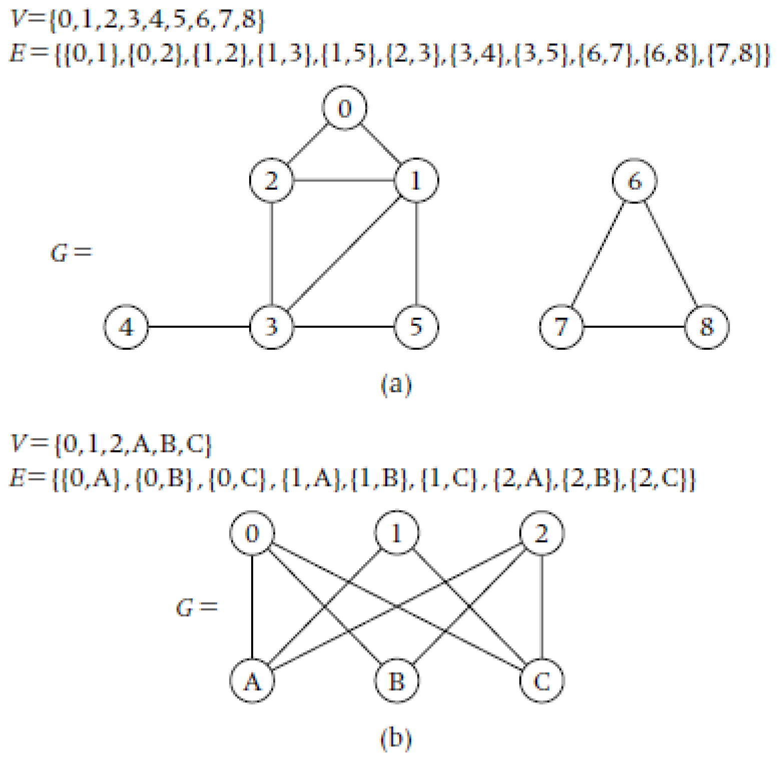
5.1 Graphs and Digraphs

In this section, we give a brief introduction to the theory of graphs and digraphs. We introduce some basic terminology, prove several elementary results, and present two standard representations of graphs and digraphs.

5.1.1 Graphs

Formally, a *graph* $G = (V,E)$ is a set $V = V(G)$ called *vertices* (or *nodes*), together with a set $E = E(G)$ called *edges*, such that an edge is an *unordered* pair of vertices. An edge $\{u,u\}$ is called a *loop*. Unless otherwise stated, we restrict our attention to graphs without loops. For simplicity, we sometimes denote the edge $\{u,v\}$ by uv. Given an edge $e = uv$ in a graph $G = (V,E)$, we refer to vertices u and v as the *end* vertices of e, and we say that e *joins* u and v. Two vertices u and v are *adjacent* if they are the two end vertices of an edge in the graph (that is, $uv \in E$). The set of all vertices adjacent to u is the *neighborhood of* u. A node is *isolated* if its neighborhood is empty. A vertex v and an edge e are *incident* if e contains vertex v, that is, if $e = vw$ for some vertex w. Two edges e and f are *adjacent* if they have an end vertex in common.

A graph can be represented pictorially by a drawing in the plane, where the vertices are represented by points in the plane and an edge $\{u,v\} \in E$ is represented by a continuous curve joining u and v (see Figure 5.2). Note from Figure 5.2*b* that such drawings allow the continuous curves to intersect at points (called *crossings*) other than vertices. It turns out that the graph in Figure 5.2b cannot be drawn in the plane without at least one crossing.


$V=\{0,1,2,3,4,5,6,7,8\}$
$E=\{\{0,1\},\{0,2\},\{1,2\},\{1,3\},\{1,5\},\{2,3\},\{3,4\},\{3,5\},\{6,7\},\{6,8\},\{7,8\}\}$

$V=\{0,1,2,A,B,C\}$
$E=\{\{0,A\},\{0,B\},\{0,C\},\{1,A\},\{1,B\},\{1,C\},\{2,A\},\{2,B\},\{2,C\}\}$

(a) A planar graph, and (b) a nonplanar

Figure 5.2

Graphs such as the one in Figure 5.2a, which can be drawn in the plane without crossings, are called *planar graphs*. Such a drawing is called an *embedding* of G in the plane. Planar graphs are particularly important in computer science. For example, they are useful in VLSI design and flow diagrams. If a graph is not planar, then we say that it is *nonplanar*.

Two graphs $G = (V,E)$ and $G' = (V',E')$ are *isomorphic* if there exists a bijective mapping $\beta: V \rightarrow V'$ from the vertex set V of G onto the vertex set V' of G' such that adjacency relationships are preserved; that is, $\{u,w\} \in E$ if, and only if, $\{\beta(u),\beta(w)\} \in E'$. The mapping β is called an *isomorphism*. Deciding whether two graphs are isomorphic is, in general, a difficult problem. A graph is *complete* if every pair of distinct vertices is joined by an edge. Clearly, any two complete graphs having the same number of vertices are isomorphic. A complete graph on n vertices is denoted by K_n.

We denote the number of vertices and edges by $n = n(G)$ and $m = m(G)$, respectively, so that $n = |V|$ and $m = |E|$. The *degree* of a vertex $v \in V$, denoted by $d(v)$, is the number of edges incident with v. Let $\delta = \delta(G)$ and $\Delta = \Delta(G)$ denote the minimum and maximum degrees, respectively, over all the vertices in G, so that $\delta \leq d(v) \leq \Delta$ for all $v \in V$.

The following useful formula due to Euler relates the number m of edges to the sum of the degrees of the vertices.

Proposition 5.1.1

The sum of the degrees over all the vertices of a graph G equals twice the number of edges; that is,

$$\sum_{v \in V} d(v) = 2m. \tag{5.1.1}$$

Proof

Every edge is incident with exactly two vertices. Therefore, when summing the degrees over all the vertices we count each edge exactly twice, once for each of its end vertices. ■

A graph is *r-regular* if every vertex has degree r. A useful corollary of Proposition 5.1.1 is the following result relating the number of vertices and edges of an r-regular graph.

Corollary 5.1.2

If G is an r-regular graph with n vertices and m edges, then

$$m = \frac{rn}{2}. \tag{5.1.2}$$ □

Another corollary of Proposition 5.1.1 is obtained by taking both sides of (5.1.1) **mod** 2.

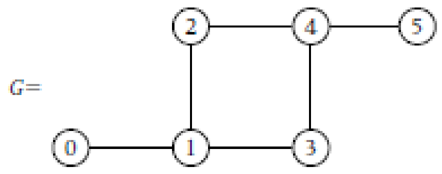
Corollary 5.1.3

For any graph G, the number of vertices of odd degree is even. □

A *path P of length p* ($p \geq 0$) joining vertices u and v is an alternating sequence of $p + 1$ vertices and p edges $u_0e_1u_1e_2 \ldots e_pu_p$ such that $u = u_0$, $v = u_p$, where e_i joins u_{i-1} and u_i, $i = 1,2, \ldots, p$. We call u_0 and u_p the *initial* and *terminal* vertices, respectively, and the remaining vertices in the path the *interior* vertices. Vertices in a path can be repeated, but edges must be distinct. Since the path P is completely defined by the sequence of vertices $u_0u_1 \ldots u_p$, we often use this shorter sequence to denote P. If $u = v$, then the path is called a *closed path* or *circuit*. We call a path *simple* if the interior vertices in the path are all distinct and are different from the initial and terminal vertices. A *simple circuit* or *cycle* is a simple closed path. A path that contains every edge in the graph exactly once is called an *Eulerian path*. A circuit that contains every edge exactly once is called an *Eulerian circuit* or *Eulerian tour*. A simple path that contains every vertex in the graph is called a *Hamiltonian path*. A cycle that contains every vertex in the graph is called a *Hamiltonian cycle*.

Two vertices $u, v \in V$ are *connected* if there exists a path (of length 0 when $u = v$) that joins them. The relation *u is connected to v* is an equivalence relation on V. Each equivalence class C of vertices, together with all incident edges, is called a (*connected*) *component* of G. When G has only one component, then G is *connected*; otherwise, G is *disconnected*.

The *distance* between u and v, denoted by $d(u,v)$, is the length of a path from u to v that has the shortest (minimum) length among all such paths. By convention, if u and v are not connected, then $d(u,v) = \infty$. The *diameter* of G is the maximum distance between any two vertices. In Figure 5.3 the distance between every pair of vertices and the diameter is given for a sample graph G. We show the distances using a 6-by-6 *distance matrix*, whose ijth entry is given by $d(i,j)$.



Distance matrix of G

$$\begin{bmatrix} 0 & 1 & 2 & 2 & 3 & 4 \\ 1 & 0 & 1 & 1 & 2 & 3 \\ 2 & 1 & 0 & 2 & 1 & 2 \\ 2 & 1 & 2 & 0 & 1 & 2 \\ 3 & 2 & 1 & 1 & 0 & 1 \\ 4 & 3 & 2 & 2 & 1 & 0 \end{bmatrix} \quad \text{Diameter } G = 4$$

Distances between vertices for a sample graph G on six vertices

Figure 5.3

A *tree* is a connected graph without cycles. A rooted tree is simply a tree with one vertex designated as the root. Every tree is planar, so that it can be drawn in the plane without crossings. The following propositions are not difficult, and their proofs are left as exercises.

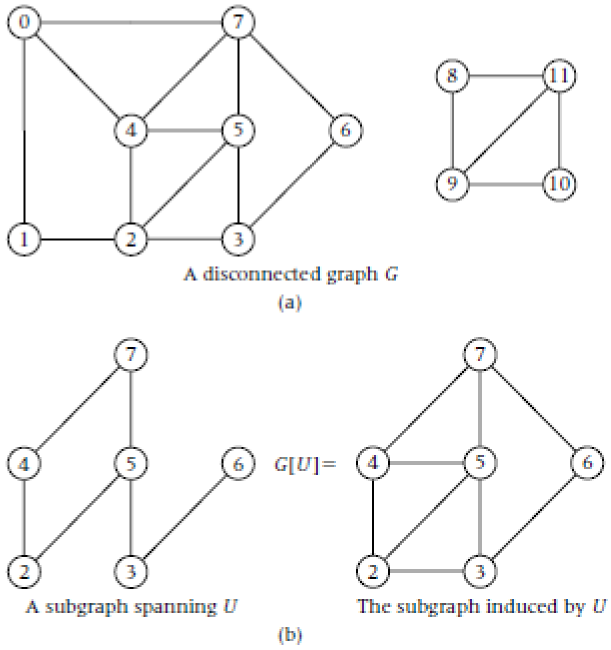
Proposition 5.1.4

A graph T is a tree if, and only if, there exists a unique path joining every pair of distinct vertices of T. □

Proposition 5.1.5

A connected graph T is a tree if, and only if, the number of edges of T is one less than the number of vertices. □

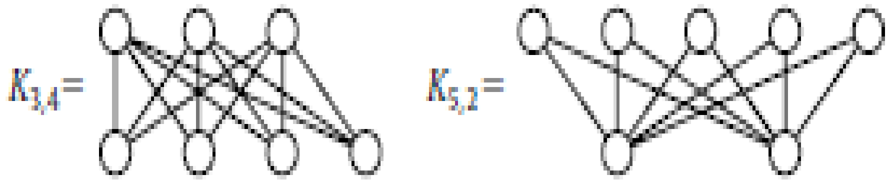
A *subgraph* H of G is a graph such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. A subgraph H that is a tree is a *subtree* of G, or simply a *tree* of G. A subgraph H of G is called a *spanning* subgraph if H contains all the vertices of G. When a subgraph H of G is a tree, we use the term *spanning tree* of G. Given a subset U of vertices of G, the subgraph $G[U]$ *induced by* U is the subgraph with vertex set U and edge set consisting of all edges in G having both end vertices in U. In Figure 5.4b we show a subgraph and an induced subgraph on the same set of vertices U. Note that a component of a graph G is a connected induced subgraph of G that is not contained in a strictly larger connected subgraph (see Figure 5.4a).



(a) The components of G are the subgraphs, $G[A]$ and $G[B]$, induced by the vertex sets $A = \{0,1,2,3,4,5,6,7\}$ and $B = \{8,9,10,11\}$, respectively; (b) a subgraph spanning $U = \{2,3,4,5,6,7\}$ and the induced subgraph $G[U]$

Figure 5.4

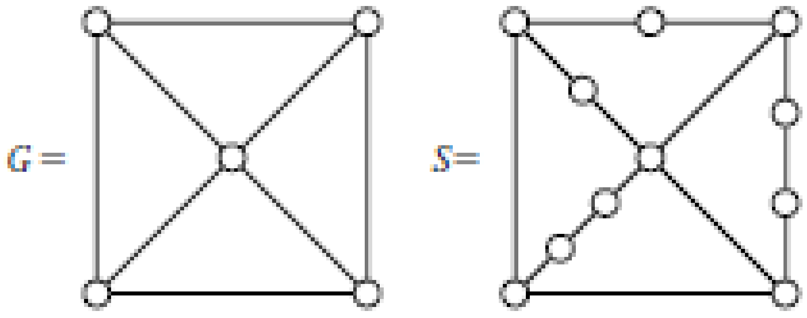
A graph is *bipartite* if there exists a bipartition of the vertex set V into two sets X and Y such that every edge has one end in X and the other in Y. The *complete bipartite* graph $K_{i,j}$ is the bipartite graph with $n = i + j$ vertices, i vertices belonging to X and j vertices belonging to Y, such that there is an edge joining every vertex $x \in X$ to every vertex $y \in Y$. The complete bipartite graph $K_{3,3}$ is shown in Figure 5.2b. Two other examples of complete bipartite graphs are given in Figure 5.5.



Two complete bipartite graphs

Figure 5.5

The graphs K_5 and $K_{3,3}$ are both nonplanar graphs. If you try drawing them in the plane without edge crossings, you will have trouble. In fact, K_5 and $K_{3,3}$ are in some sense the quintessential nonplanar graphs (see Theorem 5.1.6). A *subdivision* S of G is a graph obtained from G by replacing each edge e with a path joining the same two vertices as e (subdividing the edge e, as in Figure 5.6). Observe that G is a subdivision of itself (replace each edge with a path of length 1). Clearly, if G is nonplanar, then every subdivision of G is nonplanar. It is also clear that if G contains a nonplanar graph, then G is nonplanar. It follows that a graph is nonplanar if it contains a subgraph that is a subdivision of K_5 or $K_{3,3}$. The fact that the converse is also true is a surprising and deep result discovered by Kuratowski. We state Kuratowski's theorem without proof.



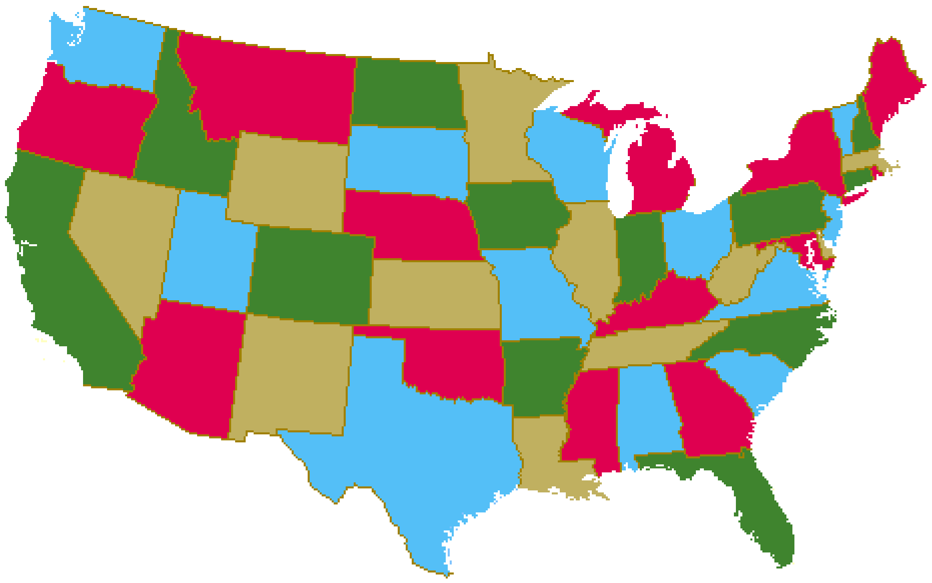
Subdividing a graph

Figure 5.6

Theorem 5.1.6 (Kuratowski's Theorem)

A graph G is nonplanar if, and only if, it contains a subgraph that is (isomorphic to) a subdivision of K_5 or $K_{3,3}$. □

The condition of being planar occurs in many computer applications, such as the design of VLSI circuits. Moreover, planar graphs became of interest long before the advent of computers. For example, in 1750 Euler gave his famous polyhedron formula relating the number of vertices, edges, and faces of a connected planar graph. As another example, a very famous mathematical question known as the four color conjecture can be modeled using planar graphs. For centuries map makers had known implicitly that they only needed four colors to color the countries in any map drawn on the globe so that no two neighboring countries (that is, countries sharing a common boundary consisting of more than an isolated point) get the same color (called a *proper* coloring). See Figure 5.7 for a four-coloring of the map of the 48 contiguous states of the U.S.A. That four colors suffice for any map on the globe was stated by Guthrie in 1856 as a formal conjecture, which resisted proof for over 100 years.



A proper 4-coloring of the map of the USA (color Alaska and Hawaii as you like)

Figure 5.7

Given any map on the globe, there is a naturally associated planar graph whose vertices correspond to the countries, and where two vertices are adjacent if, and only if, the countries they represent are neighbors. It is easy to see that the graph associated with a map on the globe is planar. The map coloring problem then transforms to the equivalent problem of coloring the vertices of a planar graph using no more than four colors so that no two adjacent vertices get the same color. Certainly the transformed problem has a rather elegant mathematical formulation unencumbered by geometrically complex boundary curves associated with maps. Moreover, graphs can be input to a computer using simple data structures such as adjacency matrices, so proofs involving exhaustive case checking are sometimes possible. In fact, it was just that type of proof (together with deep mathematical insight) that was used by Appel and Haken in 1970 to settle the four color conjecture in the affirmative. The proof consisted in showing that two contradictory conditions hold for planar graphs that are not four-colorable: there is a certain finite set of planar graphs S (at least) one of which would have to occur as a subgraph of any planar graph G that is not four-colorable (S is an *unavoidable* set), and if a graph G contains one of these subgraphs, then there would be another graph G' on fewer vertices that is also not four-colorable (each graph in S is *reducible*).

Clearly, these two conditions rule out the existence of a planar graph that is not four-colorable, since if such things exist there would have to be one having a minimal number of vertices amongst all such graphs. Appel and Haken proved condition (1) mathematically, but used the computer to check that each graph in S had the reducibility property given in condition (2). Since then other proofs of the four color conjecture have been given along the same lines but using a smaller set of unavoidable reducible graphs. However, exhaustive computer checking of cases remains a component of all known proofs of the four color conjecture.

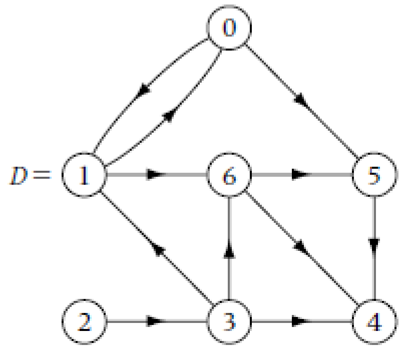
For an arbitrary graph *G* and an integer *c*, it is a difficult problem (in fact, it is NP-complete) to determine whether or not the vertices of G can be colored using no more than *c* colors such that no two adjacent vertices get the same color (called a proper coloring of *G*). The minimum number of colors needed to properly color the vertices of G is called the *chromatic number* of G. This invariant is discussed further in the exercises at the end of this chapter.

5.1.3 Digraphs

Graphs with directed edges are natural models for many physical phenomena. For example, the pairings of the players in a (match-play golf, table tennis, etc.) tournament can be represented by a graph, where the vertices correspond to the players and where two vertices *A* and *B* are joined with an edge if players *A* and *B* are to play each other. The winner of the match between each pair of players *A* and *B* can be recorded by "orienting" the edge joining *A* and *B*, so that it is directed from *A* to *B* if *A* defeats *B*. The resulting oriented graph is called a *directed graph* or *digraph*.

Formally, a *digraph D* is a set $V = V(D)$ of vertices together with a set $E = E(D)$ called *directed edges* (sometimes called *arcs*) such that a directed edge is an *ordered* pair of vertices. A digraph can be represented by a drawing in the plane in the same way that a graph can, except that we add an arrow to the curve representing each edge to indicate the orientation of that edge (see Figure 5.8).

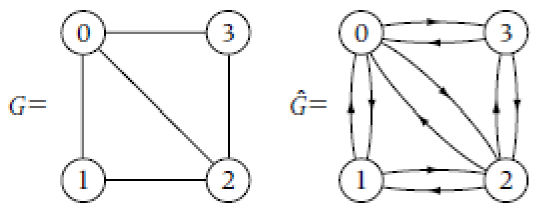
$V=\{0,1,2,3,4,5,6\}$
$E=\{\{0,1\},\{1,0\},\{0,5\},\{3,1\},\{1,6\},\{2,3\},\{3,4\},\{3,6\},\{6,4\},\{5,4\},\{6,5\}\}$



Drawing a digraph

Figure 5.8

Since an unordered pair $\{a,b\}$ is combinatorially equivalent to the two ordered pairs (a,b), (b,a), a digraph is actually a generalization of a graph. Associated with any given undirected graph $G = (V,E)$ is the combinatorial equivalent digraph $\hat{G} = (\hat{V}, \hat{E})$, where $V = \hat{V}$, and where the ordered pairs (u,v), (v,u) are in $\hat{E}$ if, and only if, the unordered pair $\{u,v\}$ is in E (see Figure 5.9).



Graph G and its combinatorial equivalent digraph $\hat{G}$

Figure 5.9

For simplicity, we sometimes denote the directed edge (u,v) by uv. We refer to u as the *tail* of e and v as the *head* of e. The *out-degree* of a vertex $v \in V$, denoted by $d_{\text{out}}(v)$ is the number of edges having tail v. Similarly, the *in-degree* of a vertex $v \in V$, denoted by $d_{\text{in}}(v)$, is the number of edges having head v. It is easily verified that

$$\sum_{v \in V} d_{\text{out}}(v) = \sum_{v \in V} d_{\text{in}}(v) = m,$$

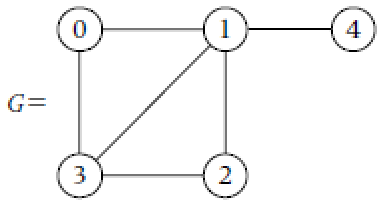
where m denotes the number of directed edges of D. The *out-neighborhood* (*in-neighborhood*) *of vertex u* is the set of all vertices v such that uv (vu) is a directed edge of D.

5.2 Implementing Graphs and Digraphs

Two standard implementations of a graph G are the adjacency matrix implementation and the adjacency lists implementation. For convenience, assume that the vertices of G are labeled 0,1, . . . , n– 1 (when referring to a vertex we do not distinguish between the label of the vertex and the vertex itself). The *adjacency matrix* of a graph G is the $n \times n$ symmetric matrix $A = (a_{ij})$ given by

$$a_{ij} = \begin{cases} 1 & \text{vertices } i \text{ and } j \text{ are adjacent in } G. \\ 0 & \text{otherwise,} \end{cases} \qquad i, j \in \{0,\ldots,n-1\}.$$

A sample graph G and its adjacency matrix are given in Figure 5.5.



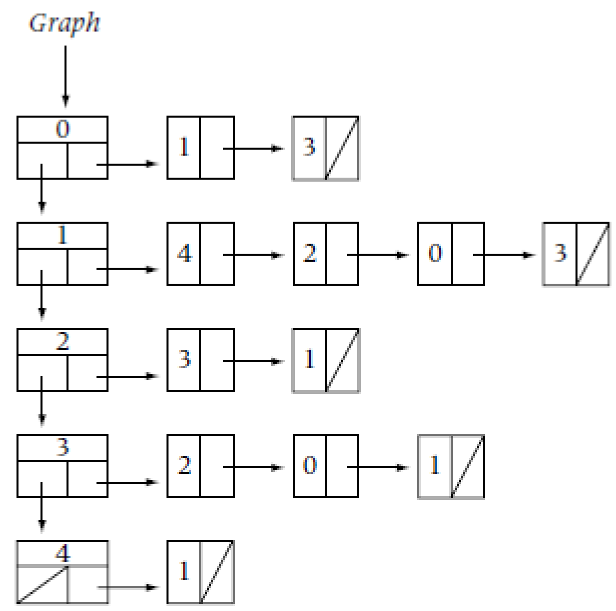
$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

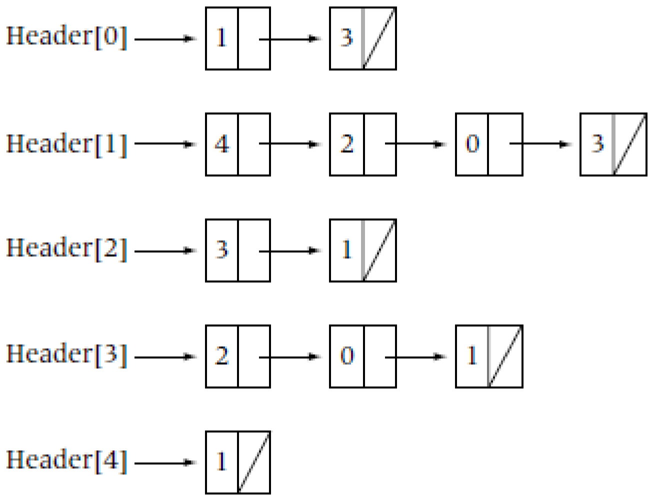
Adjacency matrix of a graph G

Figure 5.10

Implementing G using its adjacency matrix makes it easy to perform many standard operations on G, such as adding a new edge or deleting an existing edge. Note that the adjacency matrix of G allocates n^2 memory locations no matter how many edges are in the graph. Thus, implementing G using its adjacency matrix is inefficient if the number of edges of G is small relative to the number of vertices. For example, if G is a tree with n vertices, then G has only n – 1 edges. On the other hand, if $m \in \Theta(n^2)$, then the adjacency matrix representation is probably an efficient way to implement G.

Given a graph $G = (V,E)$, for $v \in V$, the *adjacency list of* v is the list consisting of all the vertices adjacent to v. The order in which the vertices are listed usually does not matter. The adjacency lists of G are generally implemented as linked lists, where pointers to the beginning of the linked lists are stored in *header* nodes. The header nodes can themselves be implemented using a linked list, or they can be stored in an array *Header*[0:n – 1]. Figures 5.11a and 5.11b illustrate the adjacency lists implementation using linked lists for the graph given in Figure 5.10, where the header nodes are implemented using a linked list and an array, respectively. We do not assume that the nodes in each adjacency list are maintained in increasing order of node labels.


(a) Linked list of header nodes


(b) Array of header nodes

Adjacency list implementations for the sample graph *G* given in Figure 5.10

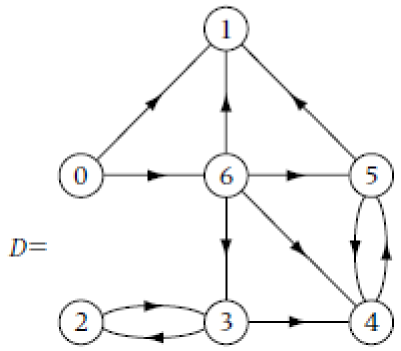
Figure 5.11

Each node of the adjacency list of vertex *i* corresponds to an edge {*i*,*j*} incident with *i*. In the adjacency lists implementation of a graph, where the header nodes belong to a linked list (see Figure 5.11*a*), it is convenient in practice to maintain a second pointer *p* in each list node defined as follows: If the list node belongs to the adjacency list of vertex *i* and corresponds to edge {*i*,*j*} then *p* points to the header node of adjacency list *j*.

Both the adjacency matrix and adjacency lists implementations of graphs generalize naturally to digraphs. The *adjacency matrix* of a digraph D, whose vertices are labeled 0,1, . . . , $n - 1$, is the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = \begin{cases} 1 & \text{if there is an arc from } i \text{ and } j, \\ 0 & \text{otherwise,} \end{cases} \qquad i, j \in \{0, \ldots, n-1\}.$$

The adjacency matrix of a sample digraph D is given in Figure 5.12.

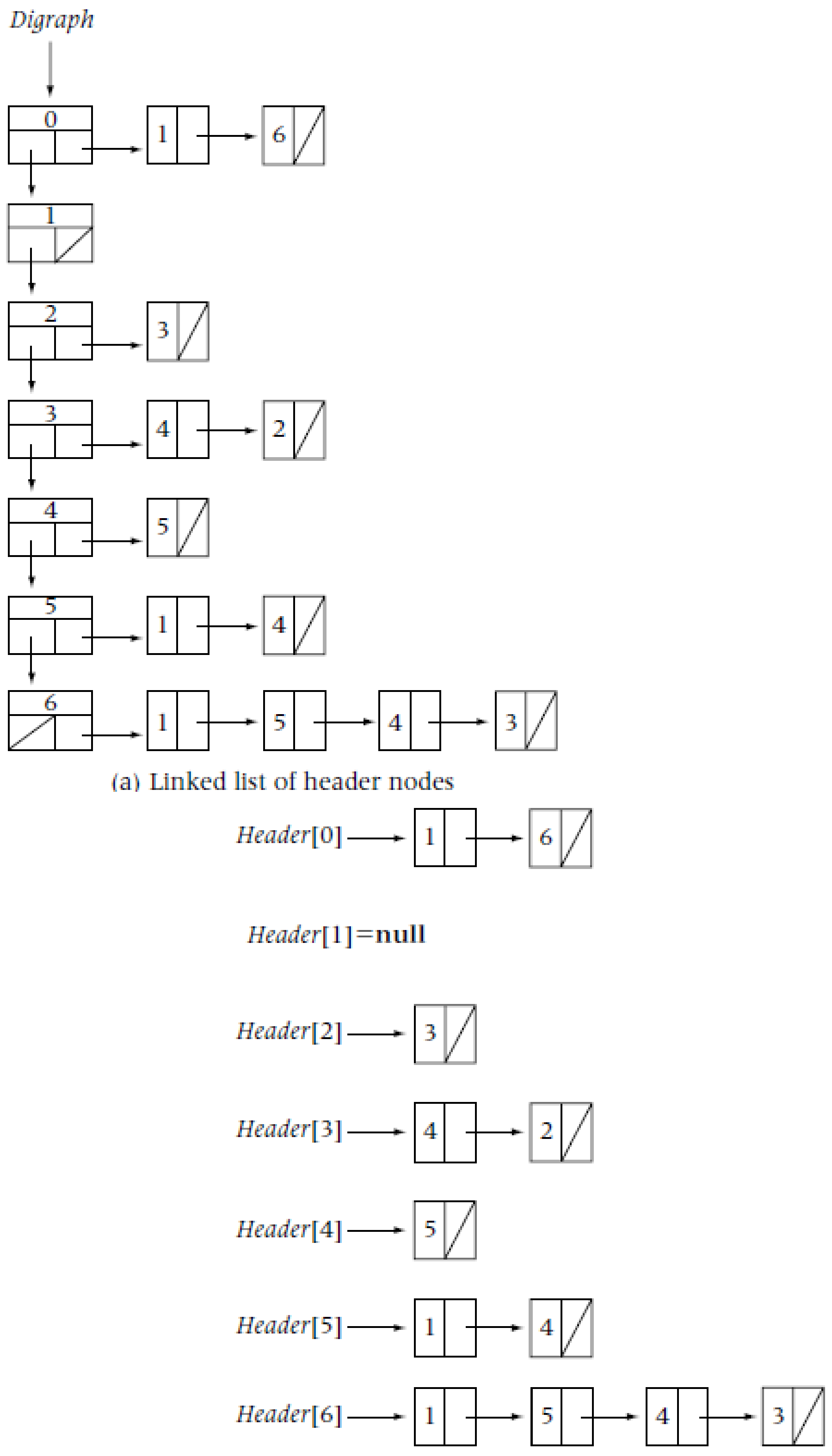


$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency matrix of a digraph D

Figure 5.12

In Figure 5.13a and 5.13b, the adjacency lists implementations using a linked list of header nodes and an array of header nodes, respectively, are illustrated for the sample digraph given in Figure 5.13.



(a) Linked list of header nodes

(b) Array of header nodes

Adjacency list implementations of the sample digraph *D* given in Figure 5.12
Figure 5.13

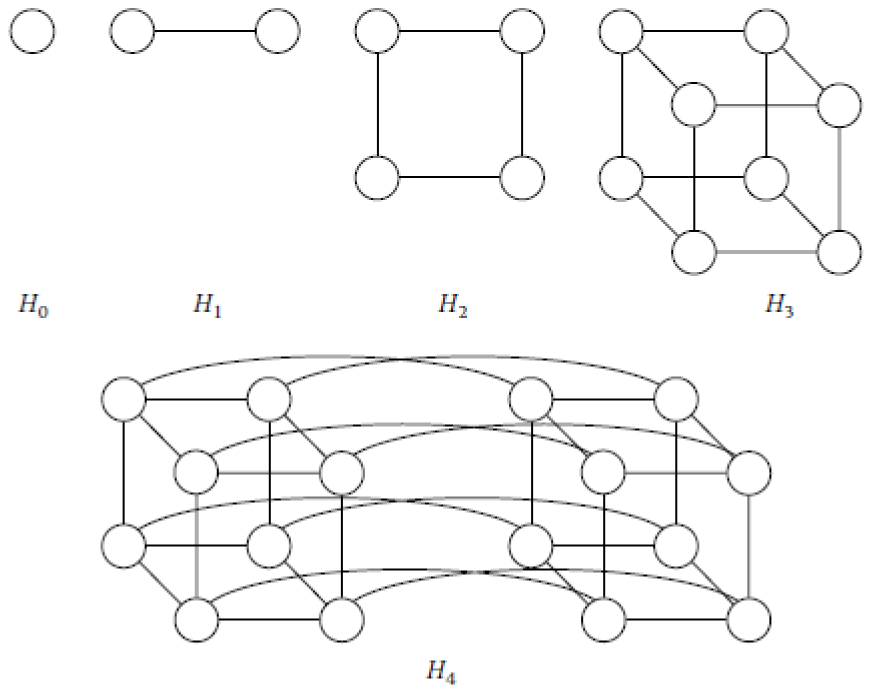
The adjacency matrix and adjacency lists representations of digraphs are generalizations of the adjacency matrix and adjacency lists representations of (undirected) graphs. To see this, consider any undirected graph G and its combinatorially equivalent digraph $\hat{G}$ (see Figure 5.9). Clearly, the adjacency matrix and adjacency lists of G and $\hat{G}$ are identical.

5.3 Graphs as Interconnection Networks

The underlying structure for an interconnection network model for parallel communication is a graph. The nodes of the graph correspond to the processors. Two nodes are joined with an edge (adjacent) whenever the corresponding two processors communicate directly with one another. In the interconnection network model, information is communicated between nonadjacent processors P and Q by relaying the information along a path in the network joining P and Q. The diameter is an upper bound on the number of communication steps needed to relay information between any two processors. Having small diameter is clearly a desirable property of an interconnection network. Another desirable property is having small maximum degree, since such interconnection network models are easier to build than those having large maximum degree.

The complete graph interconnection network K_n having n processors has diameter 1 (since the distance between every pair of vertices is 1), but is not practical to build since each vertex has very high degree ($n - 1$). At the other extreme is the one-dimensional mesh M_p, $p = n$, which has maximum degree 2 and diameter $n - 1$ (the distance between processor P_0 and processor P_{p-1} is $n - 1$). The two-dimensional $\mathrm{M}_{q,q}$ ($n = q^2$) has maximum degree 4 but has diameter $2(\sqrt{n} - 1)$, which is still quite large (the maximum distance $2(q - 1)$ occurs between processors $P_{0,0}$ and $P_{q-1,q-1}$).

There are some other interconnection networks based on graphs that achieve nice compromises between extreme values of degree and diameter. One such example is the hypercube interconnection network model. The k-dimensional hypercube H_k has 2^k nodes consisting of the set of all 0/1 k-tuples; that is, $V(H_k) = \{(x_1, . . . , x_k) \mid x_i \in \{0,1\}, i = 1, . . . , k\}$. Two nodes in $V(H_k)$ are joined with an edge of H_k if, and only if, they differed in exactly one component. We illustrate the hypercubes up to dimension 4 in Figure 11.8. Note that the hypercube of dimension k can be obtained by joining corresponding vertices in two copies of the hypercube of dimension $k - 1$.



Hypercubes H_i, $i = 0,1,2,3,4$

Figure 5.14

The hypercube H_k of dimension k is k-regular and has diameter $k = \log_2 n$ (see Exercise 5.x). Thus, the maximum degree and diameter of H_k are both logarithmic in the number of vertices n.

5.4 Search and Traversal of Graphs

The solutions to many important problems require an examination (visit) of the nodes of a graph. Two standard search techniques are *depth-first search* and *breadth-first search*.

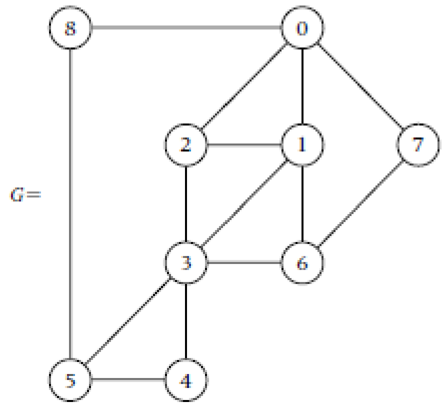
Depth-first search and breadth-first search differ in their exploring philosophies: Depth-first search always longs to see what's over the next hill (where pastures might be greener), whereas breadth-first search visits the immediate neighborhood thoroughly before moving on. After visiting a node, breadth-first search explores this node (visits all neighbors of the node that have not already been visited) before moving on. On the other hand, depth-first search immediately moves on to an unvisited neighbor, if one exists, after visiting a node. Whenever depth-first search is at an explored node, it "backtracks" until an unexplored node is encountered and then continues. This backtracking often returns to the same node many times before it is explored.

5.4.1 Depth-First Search

We first give the pseudocode *DFS* for depth-first search. For simplicity, we assume that G has n vertices labeled $0,1, . . . , n- 1$; we use the symbol v to simultaneously denote a vertex and its label. We maintain an auxiliary array $Mark[0:n- 1]$ to keep track of the nodes that have been visited.

```
procedure DFS(G,v) recursive
Input: G (a graph with n vertices and m edges)
          v (a vertex) //The array Mark[0:n– 1] is global and initialized to 0s
Output:   the depth-first search of G with starting vertex v
   Mark[v] ← 1
   call Visit(v)
   for each vertex u adjacent to v do
      if Mark[u] = 0 then call DFS(G,u) endif
   endfor
end DFS
```

The **for** loop in the *DFS* pseudocode did not explicitly describe the order in which the vertices are considered. This order is incidental to the nature of the search, and would, in general, depend on the particular implementation of the graph *G* (for example, adjacency matrix, adjacency lists, and so forth). Since we have labeled the vertices 0,1, . . . , n– 1, we assume that the vertices are accessed by the **for** loop in increasing order of their labels (thereby making the order of visiting the nodes independent of the implementation). The graph in Figure 5.15 illustrates this convention.



DFS with $v = 6$ visits vertices in the order 6,1,0,2,3,4,5,8,7

Figure 5.15

For the graph *G* in Figure 5.15, *DFS* visits all the vertices of *G*. For a general graph *G*, *DFS* with starting vertex *v* visits all the vertices in the (connected) component containing *v*.

To further illustrate the **for** loop in *DFS*, we show how this loop can be written in the two standard ways to implement a graph. Suppose first that *G* is implemented using its adjacency matrix $A[0:n-1,0:n-1]$. We assume that the vertex *v* is labeled *i*. In this case the **for** loop becomes

```
for j ← 0 to n– 1 do
   if (A[i,j] = 1) .and. (Mark[j] = 0) then
      call DFS(G,j)
```

```
        endif
    endfor
```

On the other hand, suppose that *G* is implemented using adjacency lists. Recall that a typical version of this implementation has an array *Header*[0:*n*– 1] of header nodes, where *Header*[*v*] is a pointer to the adjacency list of *v*. A node in the adjacency list for *v* corresponding to vertex *u* contains a field *Vertex* containing the index (label) of *u*. It also contains a pointer *NextVertex* to the next vertex in the adjacency list. Under these assumptions, the **for** loop in *DFS* becomes

```
    p ← Header[v]
    while (p ≠ null) do
        if Mark[p→Vertex] = 0 then
            call DFS(G,p→Vertex)
        endif
        p ← p→NextVertex
    endwhile
```

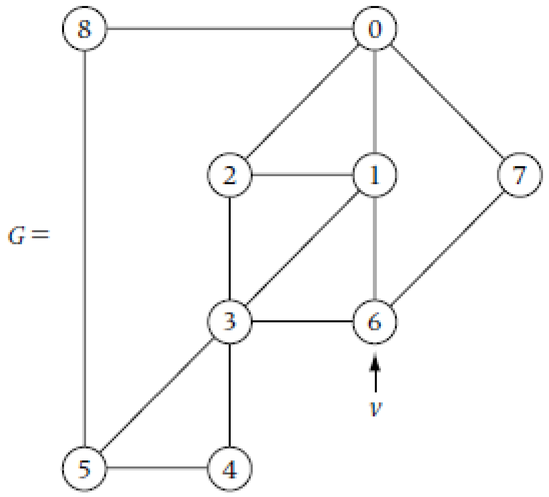
It is useful to write *DFS* as a nonrecursive procedure. For convenience, the nonrecursive version calls a procedure *Next* that determines the next unvisited node *w* adjacent to the node *u* just visited. If no such node *w* exists, then a Boolean parameter *found* is set to be **.false.**.

```
procedure DFS(G,v)
Input: G (a graph with n vertices and m edges)
            v (a vertex)
Output:     the depth-first search of G starting from vertex v
    S a stack initialized as empty
    Mark[0:n– 1] a 0/1 array initialized to 0s
    Mark[v] ← 1
    call Visit(v)
    u ← v
    Next(u,w,found)
    while found .or. (.not. Empty(S))
        if found then   //go deeper
            Push(S,u)
            Mark[w] ← 1
            Visit(w)
            u ← w
        else
            Pop(S,u)         //backtrack
        endif
        Next(u,w,found)
    endwhile
end DFS
```

Figure 5.16 illustrates the sequence of pushes, visits, and pops performed by *DFS* for the graph of Figure 5.15.

Visit 6	
Push 6	*Visit* 1
Push 1	*Visit* 0
Push 0	*Visit* 2
Push 2	*Visit* 3
Push 3	*Visit* 4
Push 4	*Visit* 5
Push 5	*Visit* 8
Pop	$(u = 5)$
Pop	$(u = 4)$
Pop	$(u = 3)$
Pop	$(u = 2)$
Pop	$(u = 0)$
Push 0	*Visit* 7
Pop	$(u = 0)$
Pop	$(u = 1)$
Pop	$(u = 6)$

Stack is now empty.



DFS with $v = 6$ with stack operations illustrated

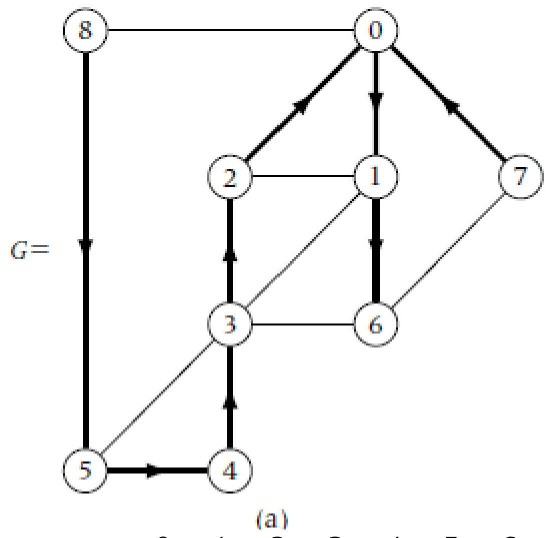
Figure 5.16

We analyze the complexity of *DFS* from the point of view of two basic operations: visiting a node and examining a node (by calls to *Next*) to see if it has been marked as visited. The worst-case complexity for both operations occurs when the graph is connected. If G is connected, then it is easily verified that every vertex is visited exactly once, so that we have a total of n node visits. Each edge uw in the graph gives rise to exactly two vertex examinations by *Next*, one with u as input parameter and one with w as input parameter. This shows that the worst-case complexity of *DFS* in terms of the number of vertices examined by *Next* is $2m$. Therefore, the total number of basic operations of the two types performed by *DFS* in the worst case is $n + 2m \in O(n + m)$.

5.4.2 Depth-First Search Tree

Depth-first search with starting vertex v determines a tree, called the *depth-first search tree* (*DFS-tree*) *rooted at v*. During a depth-first search, whenever we move from a vertex u to an adjacent unvisited vertex w, we add the edge uw to the tree. This tree is naturally implemented using the parent array representation *Parent*[0:n– 1] where u is the parent of w. For example, in the nonrecursive procedure *DFS*, we merely need to add the statement *Parent*[w] $\leftarrow u$ before pushing u on the stack, and add the array *Parent*[0:n– 1] as a third parameter.

The depth-first search tree rooted at vertex 6 of the graph of Figure 5.16 is illustrated in Figure 5.17a, and the associated array *Parent* is given in Figure 5.17b.



v	0	1	2	3	4	5	6	7	8
Parent[v]	1	6	0	2	3	4	-1	0	5

(b)

(a) The *DFS* tree rooted at vertex 6 for the graph given in Figure 5.16; (b) the associated array *Parent*[0:8]

Figure 5.17

The array *Parent*[0:*n*– 1] gives us the unique path from any given vertex *w* back to *v* in the depth-first search tree rooted at *v*. For example, suppose we wish to find the path in the depth-first search tree of Figure 5.17 from vertex 8 to vertex 6. By simply starting at vertex 7 and following the pointers in the array *Parent*[0:*n*– 1], we obtain the path

$$8,\ Parent[8]=5,\ Parent[5]=4,\ Parent[4]=3,\ Parent[3]=2,\ Parent[2]=0,$$
$$Parent[0]=1,\ Parent[1]=6.$$

5.4.3 Depth-First Traversal

If the graph *G* is connected, then *DFS* visits *all* the vertices of *G* so that *DFS* performs a *traversal* of *G*. For a general graph *G*, the following simple algorithm based on repeated calls to *DFS* with different starting vertices performs a traversal of *G*.

In the pseudocode that follow we utilize the version *DFS that keeps track of the DFS tree by maintain its parent array. In particular, we call DFS*(*G*,*v*,*Parent*[0:*n* – 1]) where *Parent*[0:*n* – 1] is an input /output parameter . This results in a forest of DFS-trees being generated (a DFS-tree rooted at *i* is generated for each *i* such that *i* is not visited when *v* = *i*). We refer to this forest as the *depth-first traversal forest* or *DFT-forest* of *G*.

```
procedure DFT(G,Parent[0:n – 1])
Input: G (a graph with n vertices and m edges)
Output:   Depth-first traversal
          Parent[0:n– 1] (an array implementing the DFT forest of G)
    Mark[0:n– 1] a 0/1 array initialized to 0s
    for v ← 0 to n – 1 do
        Parent[v] = -1
    endfor
    for v ← 0 to n – 1 do
        if Mark[v] = 0 then
            DFS(G,v,Parent[0:n – 1])
        endif
    endfor
end DFT
```

5.4.4 Breadth-First Search

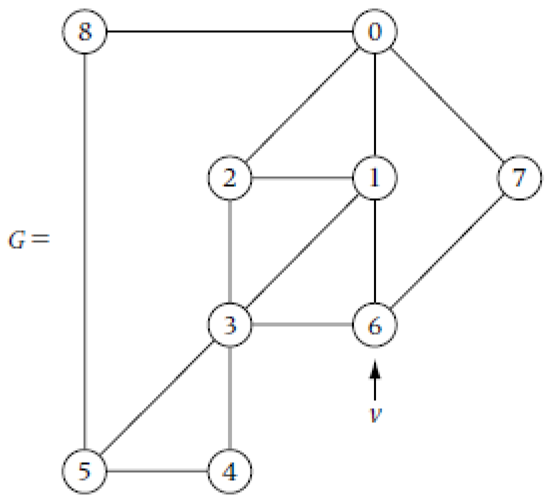
Recall that the strategy underlying breadth-first search is to visit all unvisited vertices adjacent to a given vertex before moving on. The breadth-first strategy is easily implemented by visiting these adjacent vertices and placing them on a queue. Then a vertex is removed from the queue, and the process repeated. Breadth-first search starts by inserting a vertex *v* onto an initially empty queue, and continues until the queue is empty.

```
procedure BFS(G,v)
Input: G (a graph with n vertices and m edges)
          v (vertex)  //the array Mark[0:n– 1] is global and initialized to 0s
Output:   the breadth-first search of G starting from vertex v
    Q   a queue initialized as empty
    call    Enqueue(Q,v)
    Mark[v] ← 1
    call Visit(v)
    while .not. Empty(Q) do
        Dequeue(Queue,u)
        for each vertex w adjacent to u do
            if Mark[w] = 0 then
                Enqueue(Q,w)
                Mark[w] ← 1
                Visit(w)
            endif
        endfor
    endwhile
end BFS
```

Clearly, *BFS* has the same O(*n* + *m*) complexity as *DFS*. Figure 5.18 illustrates *BFS* starting at vertex 6 for the same graph as given in Figure 5.16.

Enqueue 6	*Visit* 6
Dequeue	($u = 6$)
Enqueue 1	*Visit* 1
Enqueue 3	*Visit* 3
Enqueue 7	*Visit* 7
Dequeue	($u = 1$)
Enqueue 0	*Visit* 0
Enqueue 2	*Visit* 2
Dequeue	($u = 3$)
Enqueue 4	*Visit* 4
Enqueue 5	*Visit* 5
Dequeue	($u = 7$)
Dequeue	($u = 0$)
Enqueue 8	*Visit* 8
Dequeue	($u = 2$)
Dequeue	($u = 4$)
Dequeue	($u = 5$)
Dequeue	($u = 8$)

Queue is now empty.

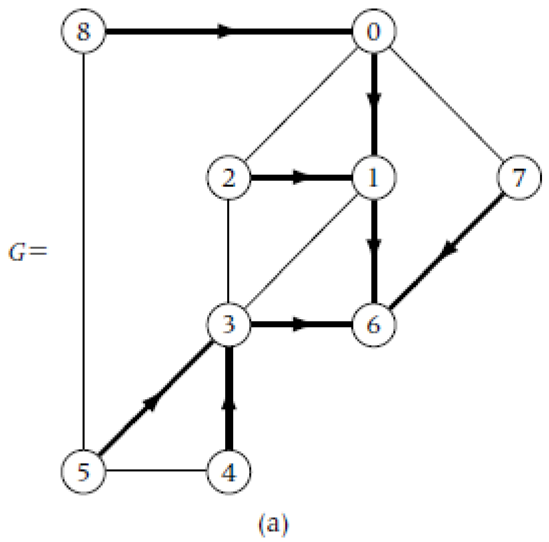


BFS with *v* = 6 visits vertices in the order 6,1,3,7,0,2,4,5,8

Figure 5.18

Breadth-first search with starting vertex *v* determines a tree spanning the component of the graph *G* containing *v*. As with *DFS*, a minor modification of the pseudocode for *BFS* generates this *breadth-first search tree* (*BFS-tree*) rooted at *v*. We maintain a parent array *Parent*[0:*n*– 1] and when enqueueing a non-visited vertex w adjacent to u, we define the parent of *w* to be *u*.

The breadth-first search tree rooted at vertex 6 and the associated array *Parent* for the graph in Figure 5.18 is illustrated in Figure 5.19.



v	0	1	2	3	4	5	6	7	8
Parent[v]	1	6	1	6	3	3	-1	6	0

(b)

BFS tree rooted at vertex 6 and its associated array *Parent*[0:8]

Figure 5.19

Just as was the case for depth-first search, breadth-first search can be used to determine whether or not the graph *G* is connected. Also, a breadth-first *traversal* of an arbitrary graph (connected or not) is obtained from the following algorithm.

```
procedure BFT(G)

Input: G (a graph with n vertices and m edges)
Output:   Breadth-first traversal
          Parent[0:n– 1] (an array implementing the BFT forest of G)
    Mark[0:n– 1] a 0/1 array initialized to -1's
    for v ← 0 to n– 1 do
        Parent[v] ← 0
    endfor
    for v ← 0 to n– 1 do
        if Mark[v] = -1 then
            BFS(G,v,Parent[0:n – 1])
        endif
    endfor
end BFT
```

Similar to depth-first traversal, breadth-first traversal generates a breadth-first search forest as implemented by the parent array *BFSForestParent*[0:*n*– 1].

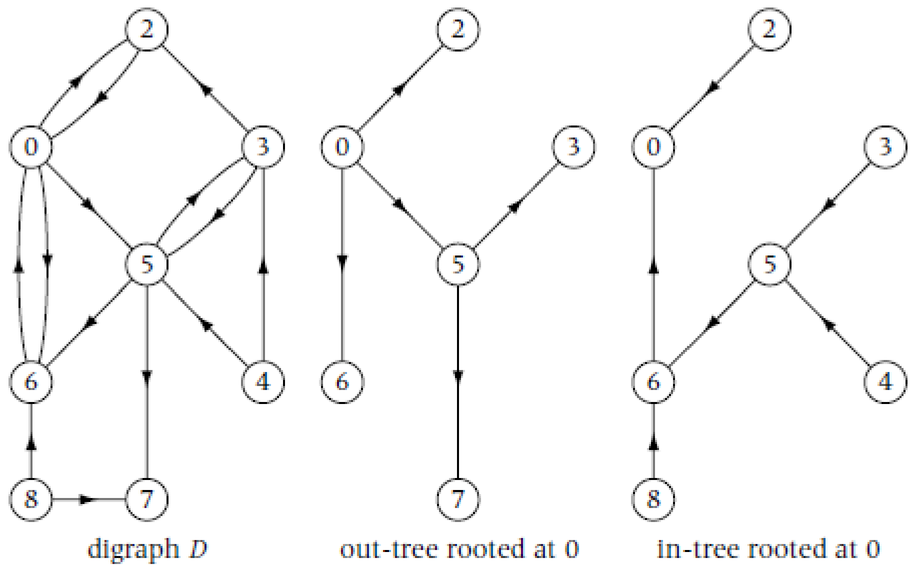
5.4.5 Breadth-First Search Tree and Shortest Paths

The breadth-first search tree starting at vertex v is actually a *shortest path tree* in the graph rooted at v; that is, it contains a shortest path from v to every vertex in the same component as v. We leave the proof of this shortest path property as an exercise. The shortest path property is in sharp contrast to the paths generated by depth-first search. Indeed, we have seen that paths generated by depth-first search with starting vertex v are often rather longer than shortest paths to v, a fact well illustrated by the complete graph K_n on n vertices. Given any $v \in V(K_n)$ breadth-first search generates (shortest) paths of length 1 from v to all other vertices. On the other hand, depth-first search applied to K_n generates a depth-first search tree that is a path of length $n - 1$.

Clearly, any algorithm that finds a shortest path tree must visit each vertex and examine each edge. Thus, a lower bound on the complexity of the shortest path problem is $\Omega(n + m)$. Since the algorithm *BFSTree* has worst-case complexity $O(n + m)$, it is an (order) optimal algorithm.

5.4.6 Searching and Traversing Digraphs

Each search and traversal technique for a graph G generalizes naturally to a digraph D. It is convenient to define both an in-version and an out-version of these searches and traversals (the out-version is equivalent to the in-version in the digraph with the edge orientations reversed). A *directed path from u to v* is a sequence of vertices $u = w_0w_1 \dots w_p = v$, such that w_iw_{i+1} is a directed edge of D, $i = 0, \dots, p - 1$. For convenience, we sometimes refer to a directed path simply as a path. Given a vertex r in a digraph D, an *out-tree T rooted at r* is a minimal subdigraph that contains a directed path from r to any other vertex v in T. An *in-tree rooted at r* is defined analogously. See the example shown in Figure 5.20.



An in-tree and an out-tree of a digraph D

Figure 5.20

Corresponding to the algorithm *DFS* for graphs we have the two algorithms indirected depth-first search *DFSIn* and out-directed depth-first search *DFSOut*. Analogously, we have algorithms *BFSIn*, *BFSOut* for breadth-first search, and similar algorithms for breadth-first traversals of digraphs.

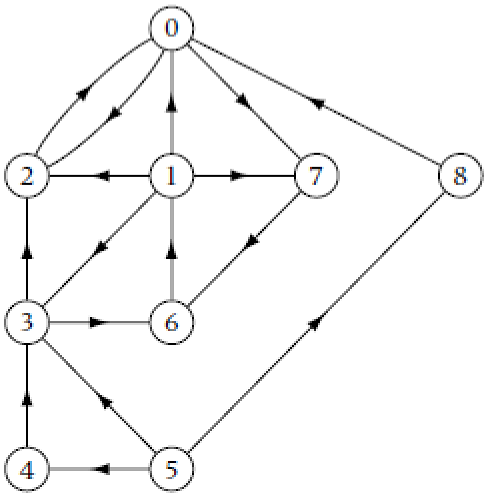
Figure 5.21a illustrates the sequence of pushes, visits, and pops performed by *DFSOut* and *DFSIn* for a sample digraph *D* starting at vertex 1, and 5.21b gives the DFS out-tree and DFS in-tree rooted at 1. Figure 5.22a illustrates the sequence of enqueues, visits, and dequeues performed by *BFSOut* and *BFSIn* for the digraph *D* of Figure 5.21. Figure 5.22b gives the BFS out-tree and BFS in-tree rooted at 1.

Visit 1	
Push 1	*Visit* 0
Push 0	*Visit* 2
Pop	$(u = 0)$
Push 0	*Visit* 7
Push 7	*Visit* 6
Pop	$(u = 7)$
Pop	$(u = 0)$
Pop	$(u = 1)$
Push 1	*Visit* 3
Pop	$(u = 3)$
Pop	$(u = 1)$
Stack is now empty.	

DFSOut starting at 1

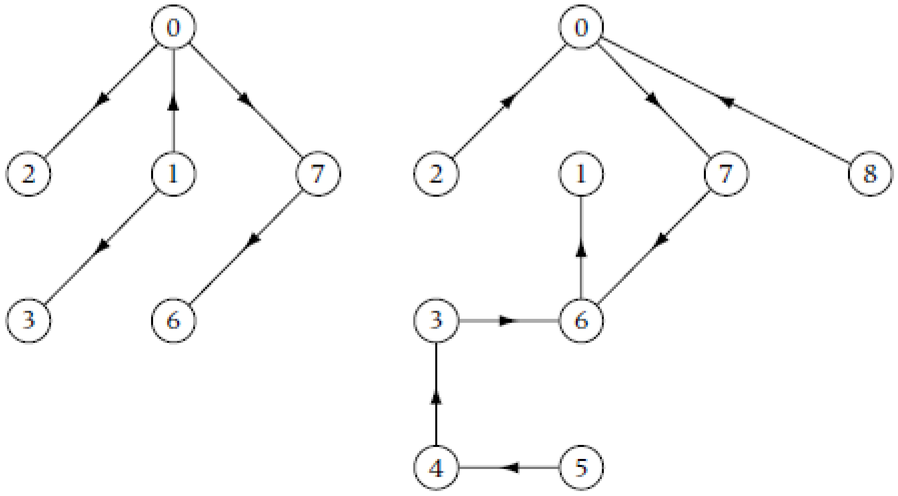
Visit 1	
Push 1	*Visit* 6
Push 6	*Visit* 3
Push 3	*Visit* 4
Push 4	*Visit* 5
Pop	$(u = 4)$
Pop	$(u = 3)$
Pop	$(u = 6)$
Push 6	*Visit* 7
Push 7	*Visit* 0
Push 0	*Visit* 2
Pop	$(u = 0)$
Push 0	*Visit* 8
Pop	$(u = 0)$
Pop	$(u = 7)$
Pop	$(u = 6)$
Pop	$(u = 1)$
Stack is now empty.	

DFSIn starting at 1



digraph *D*

(a)



DFS out-tree rooted at 1

DFS in-tree rooted at 1

v	0	1	2	3	4	5	6	7	8
ParentOutTree[*v*]	1	-1	0	1	-1	-1	7	0	-1
ParentInTree[*v*]	7	-1	0	6	3	4	1	6	0

(b)

(a) *DFSOut* and *DFSIn* with input *D* and starting vertex *v* = 1 visits vertices in the order 1,0,2,7,6,3 and 1,6,3,4,5,7,0,2,8, respectively; (b) DFS out-tree and in-tree rooted at 1 and their associated parent arrays *ParentOutTree*[0:8] and *ParentInTree*[0:8].

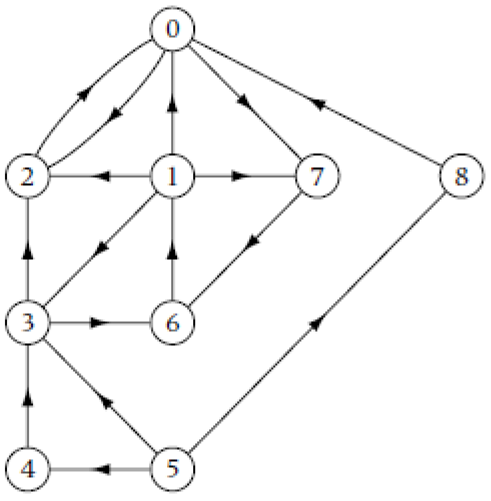
Figure 5.21

Enqueue 1	*Visit* 1
Dequeue	$(u = 1)$
Enqueue 0	*Visit* 0
Enqueue 2	*Visit* 2
Enqueue 3	*Visit* 3
Enqueue 7	*Visit* 7
Dequeue	$(u = 0)$
Dequeue	$(u = 2)$
Dequeue	$(u = 3)$
Enqueue 6	*Visit* 6
Dequeue	$(u = 7)$
Dequeue	$(u = 6)$

Queue is now empty.

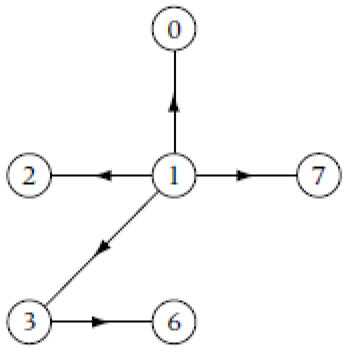
Enqueue 1	*Visit* 1
Dequeue	$(u = 1)$
Enqueue 6	*Visit* 6
Dequeue	$(u = 6)$
Enqueue 3	*Visit* 3
Enqueue 7	*Visit* 7
Dequeue	$(u = 3)$
Enqueue 4	*Visit* 4
Enqueue 5	*Visit* 5
Dequeue	$(u = 7)$
Enqueue 0	*Visit* 0
Dequeue	$(u = 4)$
Dequeue	$(u = 5)$
Dequeue	$(u = 0)$
Enqueue 2	*Visit* 2
Enqueue 8	*Visit* 8
Dequeue	$(u = 2)$
Dequeue	$(u = 8)$

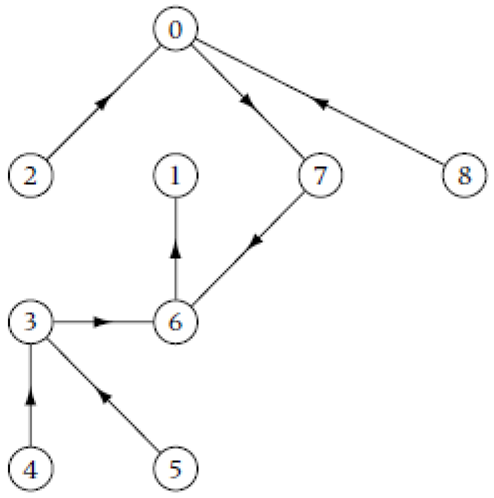
Queue is now empty.



BFSOut starting at 1 *BFSIn* starting at 1 digraph *D*

(a)





BFS out-tree rooted at 1 BFS in-tree rooted at 1

v	0	1	2	3	4	5	6	7	8
ParentOutTree[v]	1	-1	1	1	-1	-1	3	1	-1
ParentInTree[v]	7	-1	0	6	3	3	1	6	0

(b)

(a) *BFSOut* and *BFSIn* with input *D* and starting vertex $v = 1$ visits vertices in the order 1,0,2,3,7,6 and 1,6,3,7,4,5,0,2,8 respectively; (b) BFS out-tree and in-tree rooted at 1 and their associated associated parent arrays *ParentOutTree*[0:8] and *ParentInTree*[0:8].

Figure 5.22

5.4 Topological Sorting

Suppose that there are *n* tasks to be performed and that certain tasks must be performed before others. For example, if we are building a house, the task of pouring the foundation must precede the task of laying down the first floor. However, another pair of tasks might not need to be done in a particular order, such as painting the kitchen and painting the bathroom. The problem is to obtain a linear ordering of the tasks in such a way that if task *u* must be done before task *v*, then *u* occurs before *v* in the linear ordering.

The dependencies of the tasks can be naturally modeled using a directed acyclic graph (*dag*) *D*—that is, a directed graph without any directed cycles. The vertices of *D* correspond to the tasks, and a directed edge from *u* to *v* is in *D* if, and only if, task *u* must precede task *v*. A *topological sorting of D* is a listing of the vertices such that if *uv* is an edge of *D*, then *u* precedes *v* in the list. A *topological-sort labeling* of *D* is a labeling of the vertices in *D* with the labels 0, . . . , $n - 1$ such that for any edge *uv* in *D*, the label of *u* is smaller than the label of *v*.

A topological-sort labeling is obtained by performing an out-directed depth-first traversal, where we keep track of the order in which vertices become explored. A vertex becomes *explored* when the traversal accesses *u* having visited all vertices in the out-neighborhood of *u* (we assume here that a vertex is visited when it is first accessed by the traversal). The following proposition tells us immediately that a topological-sort labeling can be obtained by the reverse order in which the vertices become explored. That is, a vertex *u* is given topological-sort label $n - i$ if *u* is the i^{th} vertex to become explored, $i = 1, . . . , n$.

Proposition 5.3.1. Suppose *uv* is an arc in the dag *D*. Then *v* is explored before *u* in any depth-first out traversal of *D*.

Proof

We break the proof down into two cases.
Case 1. *v* is visited before *u* in the traversal.
In this case, note that there can not be a (directed) path from *v* to *u* in *D*, otherwise *D* would contain a cycle. Hence, after visiting *v*, a depth-first search is done and *u* is not visited in this part of the traversal. In other words, *v* will become explored before *u* is visited in the traversal.
Case 2. *u* is visited before *v* in the traversal.
In this case, when u is visited, there are unvisited vertices adjacent to *u* (namely, *v*), so that *u* will be pushed on the stack (either the explicit stack in a nonrecursive encoding, or the implicit activation record stack in a recursive encoding), and a depth-first search from *u* will commence as part of the traversal. Eventually, *v* will be visited in the depth-first search proceeding from *u*. Now when any vertex *w* is visited, if there are no unvisited vertices adjacent to *w*, then *w* becomes immediately explored. On the other hand, if there are unvisited vertices adjacent to *w*, then *w* is pushed on the stack. Note that when a vertex *w* is popped from the stack, either *w* is immediately pushed again if there is an unvisited vertex adjacent to *w*, or *w* never is pushed again, that is, *w* becomes explored. This implies that anything on the stack when a vertex *w* is visited remains on the stack until *w* is explored, i.e., all vertices on the stack when *w* is visited are explored after *w*. Since *u* is on the stack when *v* is visited, it follows that *v* is again explored before *u*. Note that in case (2) we did not use the fact that *D* contains no cycles.
■

The following Key Fact follows immediately from Proposition 5.3.1, and is the basis of the procedure *Topological* that computes the topological-sort ordering.

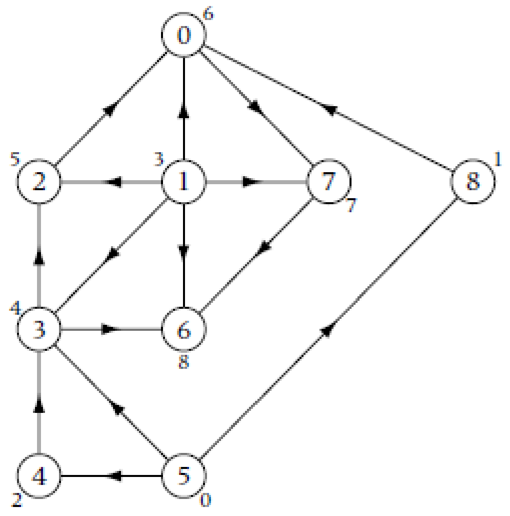
Key Fact

Given a dag *D*, a topological-sort ordering results from the reverse order in which the vertices are explored.

The following procedure *TopologicalSort* computes the topological-sort ordering resulting from the reverse order in which the vertices are explored.

```
procedure:   TopologicalSort(D,TopList[0:n– 1])
Input: D (dag with vertex set V = {0, . . . , n – 1} and edge set E)
Output:   TopList[0:n– 1] (array containing a topologically sorted list of the vertices in D)
    Mark[0:n– 1]   a 0/1 array initialized to 0s
    Counter ← n – 1
    for v ← 0 to  do
        if Mark[v] ← 0 then
            DFSOutTopLabel(D,v)
        endif
    endfor
    procedure DFSOutTopLabel(D,v) recursive
        Mark[v] ← 1
        for each w ∈ V such that vw ∈ E do
            if Mark[w] = 0 then
                DFSOutTopLabel(D,w)
            endif
        endfor
        TopList[Counter] ← v
        Counter ← Counter – 1
    end DFSOutTopLabel
end TopologicalSort
```

The array *TopList*[0:*n*– 1] output by the procedure *TopologicalSort* is illustrated in Figure 5.23 for a sample dag *D*. In Figure 5.23 we show the position in the list *TopList* outside each vertex.



v	0	1	2	3	4	5	6	7	8
TopList[v]	5	8	4	1	3	2	0	7	6

A dag D and the array *TopList*[0:8] output by *TopologicalSort*

Figure 5.23

5.6 Closing Remarks

In this chapter we saw how breadth-first search yields shortest paths in (unweighted) graphs and digraphs. The study of shortest path algorithms has a long history, and there are many such algorithms to be found in the literature. In Chapter 6 we will present algorithms for finding shortest paths in weighted graphs and digraphs based on the greedy method and in Chapter 8 we will present algorithms for all-pairs shortest paths based on dynamic programming.

Depth-first search has a host of applications. We saw how a labeling generated by a depth-first traversal yielded a topological sort of a dag. Other vertex labelings generated by depth-first search are the basis of many classical graph algorithms. Depth-first search has a number of useful properties, such as a parenthesis structure associated with the first and last access of nodes (see Exercise 5.31).

In Chapters 11 and 12 we discuss applications of graphs to the Internet and routing and connectivity of networks

Exercises

Section 5.1 Graphs and Digraphs

5.1 a) Show that the average degree α of a vertex in a graph G is given by $\alpha = 2m / n$.
b) Show that $\delta \leq 2m/n \leq \Delta$ (where δ and Δ denote the minimum and maximum degrees, respectively, of a vertex in G).

5.2 Give an example of each of the following types of graph.
a) a 2-regular graph with diameter 8
b) a 3-regular graph with diameter 5

5.3 Suppose a 6-regular graph G has 20 more edges than vertices. How many vertices and edges does G have?

5.4 Does there exist a 5-regular graph with 44 edges? Explain.

5.5 a) Using Euler's degree formula and Proposition 5.1.5 prove Proposition 4.2.3, which states that the number of interior vertices in a 2-tree is one less than the number of leaf nodes.

b) Repeat part (a) for *k*-ary trees.

5.6 Prove that a graph is bipartite if it contains no odd cycle (i.e., cycle of odd length).

5.7 Prove Proposition 5.1.4.

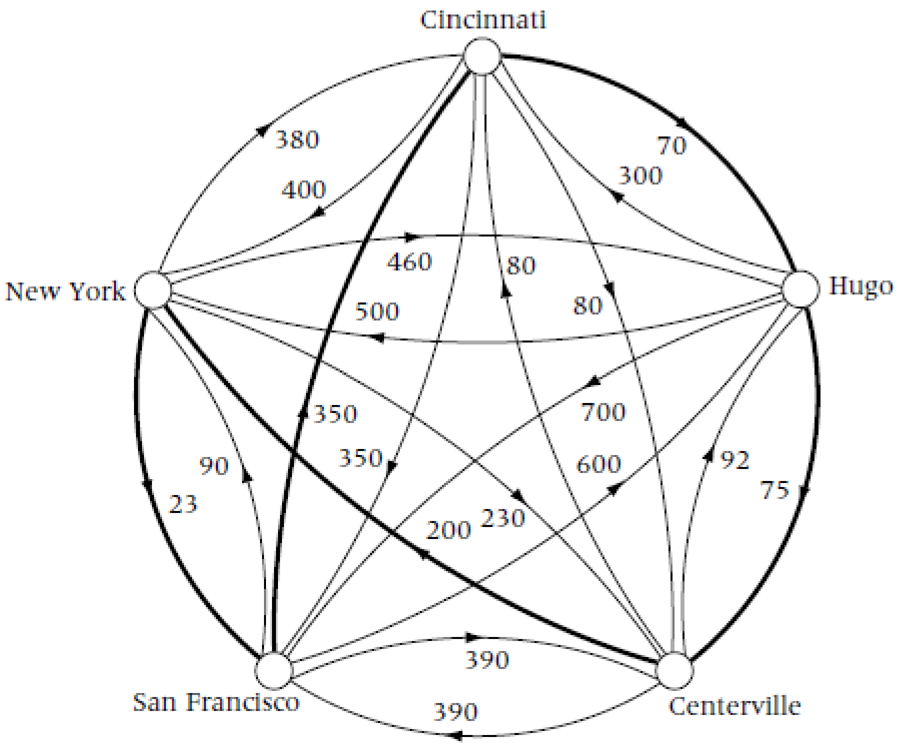
5.8 Prove Proposition 5.1.5.

5.9 Show that for a digraph D having m directed edges, we have

$$\sum_{v\in V} d_{\text{out}}(v) = \sum_{v\in V} d_{\text{in}}(v) = m.$$

5.10 Design an algorithm for finding a proper coloring of a graph G using $\Delta + 1$ colors (where Δ denotes the maximum degree of a vertex).

5.11 Suppose that a salesperson must visit n cities and that there is a cost associated with traveling from one city to another. Starting from a home city, the salesperson wishes to visit each of the other $n - 1$ cities once and return home in such an order that the total cost incurred is a minimum among all such tours (see Figure 5.24). The Traveling Salesman problem (TSP) is to determine such a minimum cost tour.



A minimum length tour of five cities amongst the 24 possible tours

Figure 5.24

The most straightforward algorithm is an *exhaustive search* that enumerates all $(n - 1)!$ tours and keeps track of the shortest tour generated. In this exercise you will design a dynamic programming algorithm for TSP having worst-case complexity in $\Theta(n2^n)$. While this is certainly an improvement over $(n - 1)!$, this exponential complexity is disappointing. However, TSP belongs to a class of problems (*NP*-hard) for which it is believed that no polynomial-time algorithm exists for any problem in the class.

TSP is equivalent to finding a minimum cost Hamiltonian cycle in a weighted digraph D. We may assume without loss of generality that D is the complete digraph $\hat{K}_n = (V, \hat{E})$, where $|V| = n$ and $\hat{E}$ consists of all pairs of distinct vertices (u,v), $u,v \in V$. Given a weighted digraph $D = (V,E)$, we merely extend the cost weighting c to $\hat{K}_n$ by setting $c(uv) = \infty$ for every (u,v) not in E. TSP for the (undirected) complete graph K_n with cost weighting c is a special case of the Traveling Salesman problem for the digraph $\hat{K}_n$: For each edge e of K_n assign both of the directed edges e^- and e^+ of $\hat{K}_n$ corresponding to e the weight $c(e)$.

Consider a Hamiltonian cycle H of $\hat{K}_n$, where without loss of generality we assume that vertex 0 is the initial and terminal vertex of H. Let i denote the first vertex visited by H after leaving vertex 0, and let H_i denote the subpath of H from vertex i to vertex 0, which passes through each vertex of $\hat{K}_n$ once (such a path is called a *Hamiltonian path*). If H is a minimum cost Hamiltonian cycle, then H_i must be a shortest path from i to 0 whose interior vertices consist of the set $V - \{0,i\}$. (A shortest path P joining two vertices i and j is a path that minimizes *Cost*(P), where *Cost*(P) is the sum of the costs over all the edges of P.) Now consider any subset U of V. For i a vertex not in U, let *MinCost*(i,U) denote the cost of a shortest (minimum cost) path P from i to 0 whose interior vertices consist of the set U. Note that since P is a shortest path, P must pass through each vertex of U exactly once. Note also that *MinCost*(0,$V - \{0\}$) is the minimum cost of a Hamiltonian cycle, that is, the cost of an optimal salesman tour.

a) Derive the following recurrence relation for *MinCost*(i,U)

$$MinCost(i,U) = \min_{j \in U}\{c(i,j) + MinCost(j, U - \{j\})\},$$
$$\textbf{init. cond. } MinCost(i,\varnothing) = c(i,0),\ 1 \le i \le n-1$$

b) Give pseudocode for an algorithm *TravellingSalesman* based on the recurrence relation in part (a).

c) Verify that the worst-case complexity of your algorithm *TravellingSalesman* in part (b) is given by $W(n) = (n-1)2^{n-2} \in \Theta(n2^n)$.

5.12 a) Prove that a graph G has an Eulerian path from a to b if, and only if, G is connected, every vertex different from a and b has even degree, and a and b have odd degree.

b) Prove that a graph G has an Eulerian cycle if, and only if, every vertex has even degree.

c) State and prove a necessary and sufficient condition for the existence of Eulerian paths and cycles in digraphs.

5.2 Implementing Graphs and Digraphs

5.13 Give pseudocode for the standard ADT operations of adding and deleting an edge for a graph G for each of the following implementations.

a. adjacency matrix
b. adjacency lists with array of header nodes
c. adjacency lists with a linked list of header nodes

5.14 In the adjacency lists implementation of a graph, where the header nodes belong to a linked list (see Figure 5.13a), it is convenient in practice to maintain a second pointer *p* in each list node defined as follows: If the list node belongs to the adjacency list of vertex *i* and corresponds to edge (*i*,*j*), then *p* points to the header node of adjacency list *j*. Redo Exercise 5.24c with this enhanced implementation.

5.3 Graphs as Interconnection Networks

5.15 Prove by induction that the hypercube H_k of dimension k has
a) 2^k vertices
b) $k2^{k-1}$ edges

5.16 Consider the hypercube H_k of dimension k on $n = 2^k$ vertices.
a) Show that H_k is k-regular.
b) Show that H_k has diameter $k = \log_2 n$.

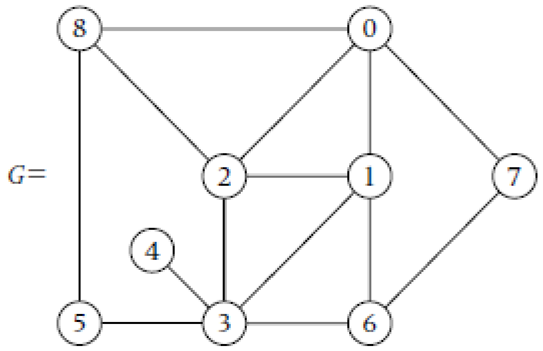
5.17 Give an alternate proof of the result in part (b) of Exercise 5.14 using part (a) of that exercise and Corollary 5.1.2.

5.18 Prove by induction that the k-dimensional hypercube H_k has exactly $C(k,j)2^{k-j}$ different sub-hypercubes of dimension j, $j = 0, \ldots, k$.

5.19 Design an $O(\log n)$ algorithm for summing n numbers on the k-dimensional hypercube H_k, where $n = 2^k$.

Section 5.4 Search and Traversal of Graphs and Digraphs

5.20 Consider the following graph.



a) Starting at vertex 3, list the vertices in the order in which they are visited by breadth-first search, and give the array *Parent*[0:8] that gives a parent implementation of the breadth-first search spanning tree.

b) Repeat part (a) for depth-first search, giving the parent array for the depth-first search tree *Parent*[0:8].

5.21 a) Prove that a depth-first search with starting vertex *v* of a graph *G* visits every vertex of *G* if, and only if, *G* is connected.

b) More generally, prove that a depth-first search with starting vertex *v* of a graph *G* visits every vertex of the component of *G* containing *v*.

c) Repeat (a) and (b) for breadth-first search.

5.22 Give complete pseudocode for *DFS* for the following implementations of a graph *G*, where the array *Current*[0:*n*– 1] is used by the procedure *Next*:
a) adjacency matrix
b) adjacency lists with an array of header nodes
c) adjacency lists with a linked list of header nodes

5.23 Repeat Exercise 5.28 for *BFS*.

5.24 Prove that the breadth-first search out-tree rooted at a vertex *v* is a *shortest path* out-tree rooted at *v* in the digraph; that is, it contains a shortest path from *v* to every vertex accessible from *v*.

5.25 In a depth-first search, show that if we output a left parenthesis when a node is accessed for the first time and a right parenthesis when a node is accessed for the last time (that is, when backtracking from the node), the resulting parenthesization is proper; this is each left parenthesis is properly matched with a corresponding right parenthesis.

5.26 Consider an (out-directed) depth-first traversal of a directed graph *D*. A *back edge* is an edge *vu* not belonging to the depth-first search forest *F*, such that *u* and *v* belong to the same tree in *F* and *u* is an ancestor of *v*. Show that a directed graph *D* is a dag if, and only if, there are no back edges.

5.27 a) Design an algorithm for determining whether a digraph *D* is a dag.
b) Modify your algorithm in (a) to output a directed cycle if *D* is not a dag.

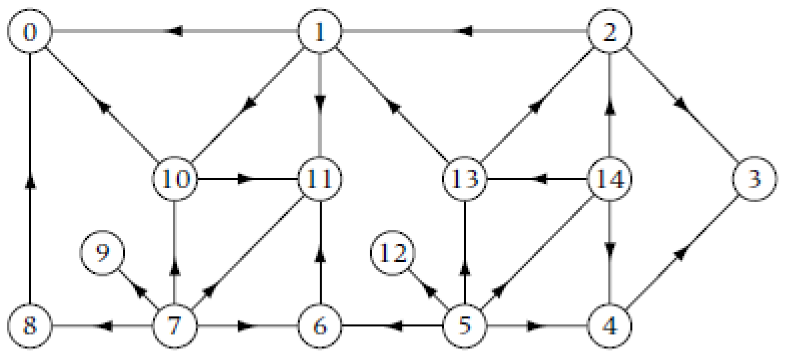
5.28 Design and analyze an algorithm that determines whether a graph is bipartite and identifies the bipartition of the vertices if it is bipartite for the following implementations of the graph.
a) adjacency matrix
b) adjacency lists

5.29 Design and analyze an algorithm that computes the diameter of a connected graph *G*.

5.30 The *girth* of a graph *G* is the length of the smallest cycle in *G*. Design and analyze an algorithm for determining the girth of *G*.

Section 5.5 Topological Sorting

5.31 Show the topological-sort labeling generated by *TopologicalSort* for the following dag.


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14