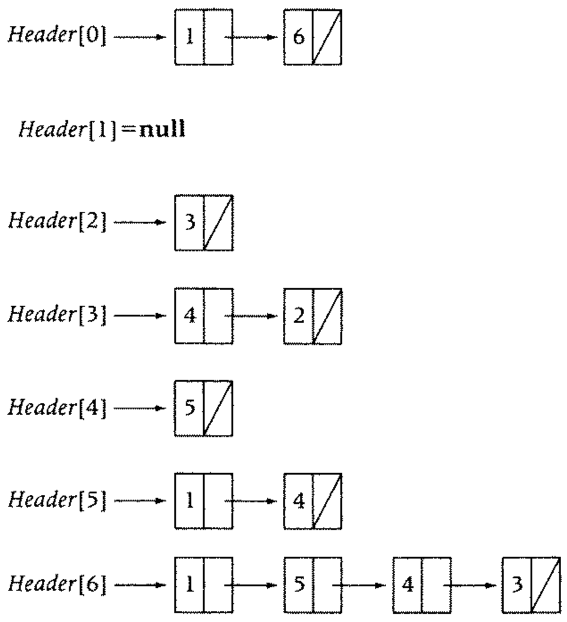
FIGURE 11.14
Continued



(b) Array of header nodes

11.2 Search and Traversal of Graphs and Digraphs

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.

11.2.1 Depth-First Search

We first give the pseudocode *DFS* for a depth-first search. For simplicity, we assume that G has n vertices labeled $0, 1, \ldots, n-1$; 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
   Visit(v)
   for each vertex u adjacent to v do
      if Mark[u] = 0 then DFS(G, u) endif
   endfor
end DFS
```

The **for** loop in the *DFS* pseudocode does 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). Because we have labeled the vertices $0, 1, \ldots, n-1$, 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 11.15 illustrates this convention.

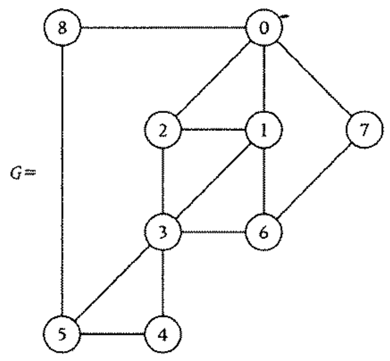


FIGURE 11.15

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

For the graph G in Figure 11.15, *DFS* visits all the vertices of G. For a general graph G, *DFS* starts at vertex v and 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. First, suppose 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
        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
        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 **.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
    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 11.16 illustrates the sequence of pushes, visits, and pops performed by *DFS* for the graph in Figure 11.15.

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*

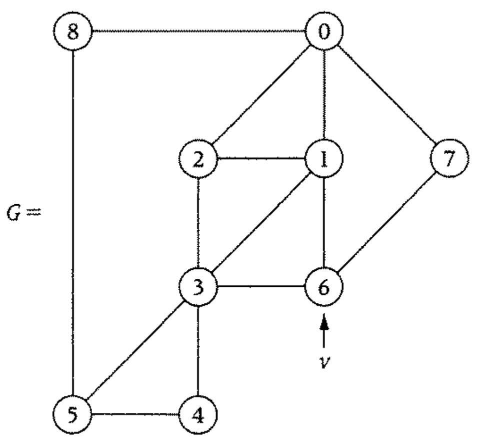
FIGURE 11.16

DFS with $v = 6$ and stack operations.

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.



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 $n + 2m \in O(n + m)$.

11.2.2 Depth-First Search Tree

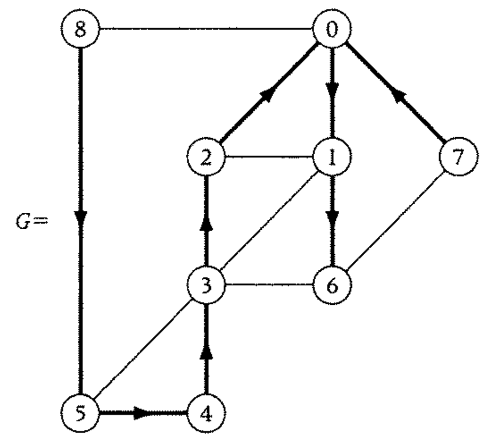
A depth-first search with starting vertex v determines a tree, called the *depth-first search tree* (*DFS-tree*) *rooted at v*. During a depth-first search, whenever we move from a vertex u to an adjacent unvisited vertex w, we add the edge uw to the tree. This tree is naturally implemented using the parent array representation (denoted by *DFSTreeParent*[0:n − 1]), where u is the parent of w. For example, in the nonrecursive procedure *DFS*, we merely need to add the statement *DFSTreeParent*[w] ← u before pushing u on the stack, and add the array *DFSTreeParent*[0:n − 1] as a third parameter. We will refer to this augmented procedure as *DFSTree*.

The DFS-tree rooted at vertex 6 of the graph in Figure 11.16 is illustrated in Figure 11.17a, and the associated array *DFSTreeParent* is given in Figure 11.17b.

The array *DFSTreeParent*[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 in Figure 11.17 from ver-

FIGURE 11.17

(a) The DFS-tree rooted at vertex 6 for the graph in Figure 11.16; (b) the associated array *DFSTreeParent*[0:8].



(a)

v	0	1	2	3	4	5	6	7
DFSTreeParent[v]	1	6	0	2	3	4	−1	0

(b)

tex 8 to vertex 6. By simply starting at vertex 8 and following the pointers in the array *DFSTreeParent*[0:n − 1], we obtain the path

> 8, *DFSTreeParent*[8] = 5, *DFSTreeParent*[5] = 4, *DFSTreeParent*[4] = 3, *DFSTreeParent*[3] = 2, *DFSTreeParent*[2] = 0, *DFSTreeParent*[0] = 1, *DFSTreeParent*[1] = 6.

11.2.3 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.

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

If we call the procedure *DFSTree* instead of *DFS*, then a forest of DFS-trees is 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* (*DFT-forest*) of G. In the procedure *DFTForest* that follows, the DFT-forest is implemented using an array *DFTForestParent*[0:n − 1].

```
procedure DFTForest(G, DFTForestParent[0:n − 1])
Input:   G (a graph with n vertices and m edges)
Output:  DFTForestParent[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
        DFTForestParent[v] ← 0
    endfor
    for v ← 0 to n − 1 do
        if Mark[v] = 0 then
            DFSTree(G, v, DFTForestParent[0:n − 1])
        endif
    endfor
end DFTForest
```

11.2.4 Breadth-First Search

Recall that the strategy underlying the 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. A 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
    Enqueue(Q, v)
    Mark[v] ← 1
    Visit(v)
    while .not. Empty(Q) do
        Dequeue(Q, 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 11.18 illustrates *BFS* starting at vertex 6 for the same graph as in Figure 11.16.

A 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*. When enqueueing a nonvisited vertex *w* adjacent to *u*, we define the parent of *w* to be *u*. We denote the modified procedure by *BFSTree* and the parent array implementation by *BFSTreeParent*[0:n − 1].

The breadth-first search tree rooted at vertex 6 and the associated array *TreeBFS* for the graph in Figure 11.18 is illustrated in Figure 11.19.

FIGURE 11.18

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

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.

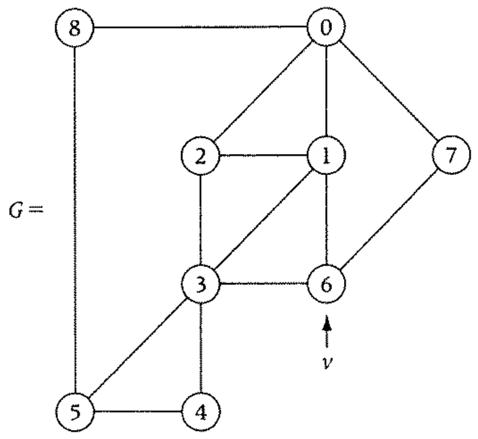
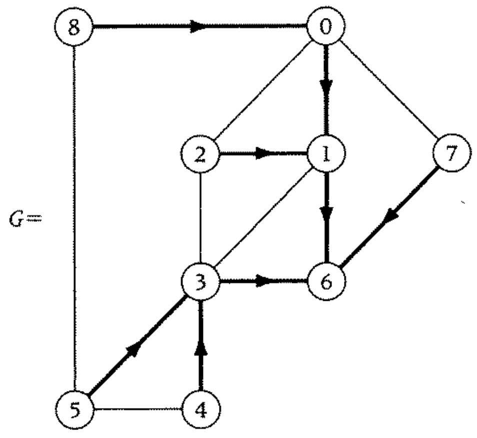


FIGURE 11.19

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



(a)

v	0	1	2	3	4	5	6	7
BFSTreeParent[*v*]	1	6	1	6	3	3	−1	6

(b)

Like a depth-first search, a breadth-first search can be used to determine whether 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:   the breadth-first traversal of G
    Mark[0:n − 1] a 0/1 array initialized to 0s
    for v ← 0 to n − 1 do
        if Mark[v] = 0 then
            BFS(G, v)
        endif
    endfor
end BFT
```

Similar to a depth-first traversal, a breadth-first traversal generates a breadth-first search forest as implemented by the parent array *BFSForest-Parent*$[0{:}n - 1]$.

11.2.5 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 a depth-first search. Indeed, we have seen that paths generated by a depth-first search with starting vertex v are often much longer than shortest paths to v, a fact well illustrated by the complete graph K_n on n vertices. Given any $v \in V(K_n)$, a breadth-first search generates shortest paths of length 1 from v to all other vertices. On the other hand, a depth-first search applied to K_n generates a depth-first search tree that is a path of length $n - 1$.

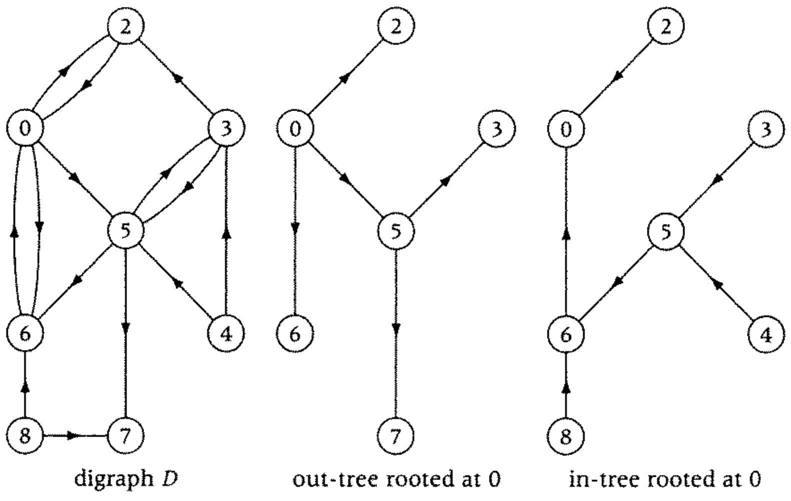
Clearly, any algorithm that finds a shortest-path tree must visit each vertex and examine each edge. Thus, a lower bound on the complexity of the shortest-path problem is $\Omega(n + m)$. Since the algorithm *BFSTree* has worst-case complexity $O(n + m)$, it is an (order) optimal algorithm.

11.2.6 Searching and Traversing Digraphs

Each search and traversal technique for a graph G generalizes naturally to a digraph D. It is convenient to define in-versions and out-versions of these searches and traversals (the out-version is equivalent to the in-version in

FIGURE 11.20

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



the digraph with the edge orientations reversed). A *directed path from u to v* is a sequence of vertices $u = w_0 w_1 \dots w_p = v$, such that $w_i w_{i+1}$ 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. Figure 11.20 illustrates an example.

Corresponding to the algorithm *DFS* for graphs, we have the two algorithms *DFSIn,* an in-directed depth-first search, and *DFSOut,* an out-directed depth-first search. Similarly, corresponding to *DFSTree,* we have the algorithms *DFSInTree* and *DFSOutTree;* and corresponding to *DFTForest,* we have *DFTInForest* and *DFTOutForest.* Analogously, we have algorithms *BFSIn, BFSOut* and *BFSInTree, BFSOutTree,* respectively, for a breadth-first search and similar algorithms for breadth-first traversals of digraphs.

Figure 11.21a illustrates the sequence of pushes, visits, and pops performed by *DFSOut* and *DFSIn* for a sample digraph D starting at vertex 1; and Figure 11.20b gives the DFS out-tree and DFS in-tree rooted at 1. Figure 11.21a illustrates the sequence of enqueues, visits, and dequeues performed by *BFSOut* and *BFSIn* for the digraph D of Figure 11.21. Figure 11.22b gives the BFS out-tree and BFS in-tree rooted at 1.

FIGURE 11.21

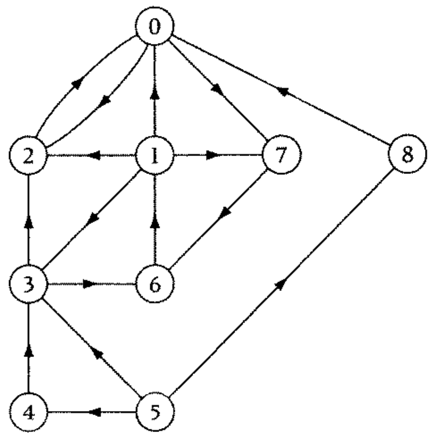
(a) *DFSOut* and *DFSIn* with input *D* and starting vertex $v = 1$ visit vertices in the order 1, 0, 2, 7, 6, 3 and 1, 6, 3, 4, 5, 7, 0, 2, 8, respectively; (b) DFS out-tree and in-tree rooted at 1 and their associated arrays.

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

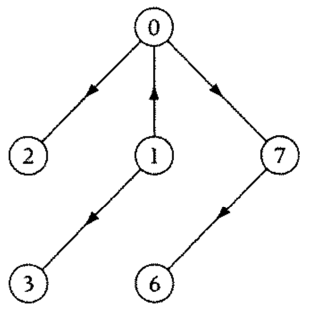
Visit 1	
Push 1	*Visit* 6
Push 6	*Visit* 3
Push 3	*Visit* 4
Push 4	*Visit* 5
Pop	$(u = 4)$
Pop	$(u = 3)$
Pop	$(u = 6)$
Push 6	*Visit* 7
Push 7	*Visit* 0
Push 0	*Visit* 2
Pop	$(u = 0)$
Push 0	*Visit* 8
Pop	$(u = 0)$
Pop	$(u = 7)$
Pop	$(u = 6)$
Pop	$(u = 1)$
Stack is now empty.	

DFSIn starting at 1

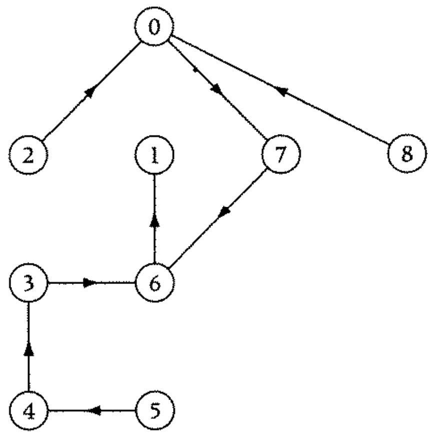


digraph *D*

(a)



DFS out-tree rooted at 1



DFS in-tree rooted at 1

v	0	1	2	3	4	5	6	7	8
DFSOutTreeParent[*v*]	1	−1	0	1	−1	−1	7	0	−1
DFSInTreeParent[*v*]	7	−1	0	6	3	4	1	6	0

(b)