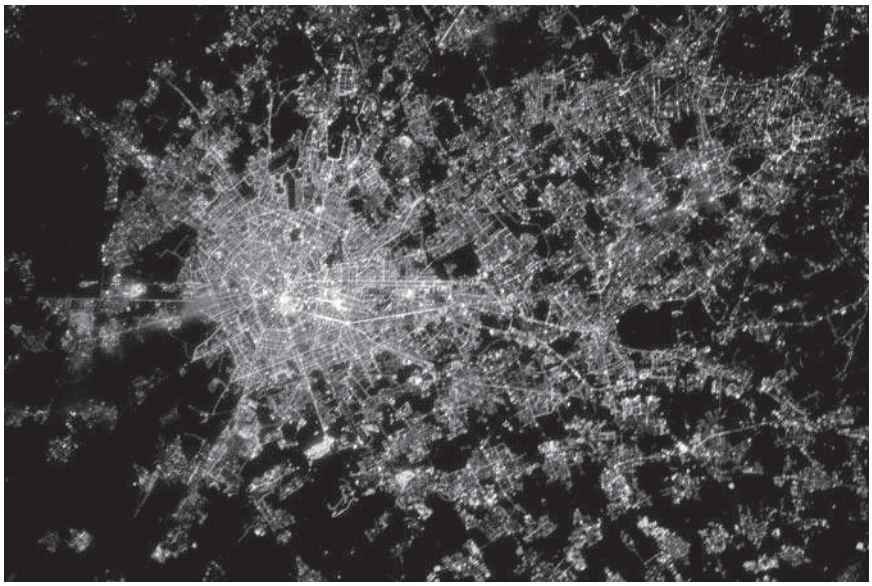
Chapter 13 Graphs and Traversals



The metropolitan area of Milan, Italy at night. Astronaut photograph ISS026-E-28829, 2011. U.S. government image. NASA-JSC.

Contents

Connectivity information is present in a multitude of different applications, including social networks, road networks, computer games and puzzles, and computer networks. In computer networks, for instance, computers are connected together using routers, switches, and (wired or radio) communication links. Information flows in the form of individual packets, from one computer to another, by tracing out paths that hop through intermediate connections. Alternatively, in computer puzzle and game applications, game positions are connected by the transitions that are allowed between them. In either case, however, we might be interested in paths that can be mapped out in the connectivity structures that are represented in these applications. For instance, a path in a puzzle application could be a solution that starts from an initial game position and leads to a goal position.

In some applications, we might not really care whether we find a shortest path from a source to a destination, whereas in others we might be interested in finding the shortest paths possible. For example, in interactive applications, such as in online games or video chat sessions, we would naturally want packets to flow in our computer network along shortest paths, since we want to minimize any delays that occur between one person's action and the reaction of communicating partner. In other applications, such as in solving a maze puzzle, there may be only a single path between a source and destination, with several dead ends that need to be ruled out in order find this path. Thus, we would like to design algorithms that can efficiently identify paths in connectivity structures.

In addition to these motivating applications, connectivity information can be defined by all kinds of relationships that exist between pairs of objects. Additional examples include mapping (in geographic information systems), transportation (in road and flight networks), and electrical engineering (in circuits). The topic we study in this chapter—***graphs***—is therefore focused on representations and algorithms for dealing efficiently with such relationships. That is, a graph is a set of objects, called "vertices," together with a collection of pairwise connections between them, which define "edges."

Because applications for graphs are so widespread and diverse, people have developed a great deal of terminology to describe different components and properties of graphs, which we also explore in this chapter. Fortunately, since most graph applications are relatively recent developments, this terminology is fairly intuitive.

We begin this chapter by discussing much of this terminology and some elementary properties of graphs. We also present ways of representing graphs. As highlighted above, traversals are important computations for graphs, both for shortest paths and for arbitrary paths, and we discuss in Section 13.2. We discuss directed graphs in Section 13.4, where pairwise relationships have a given direction, and connectivity problems become more complicated.

13.1 Graph Terminology and Representations

A ***graph*** G is a set, V, of ***vertices*** and a collection, E, of pairs of vertices from V, which are called ***edges***. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V. Incidentally, some people use different terminology for graphs and refer to what we call vertices as "nodes" and what we call edges as "arcs" or "ties."

Edges in a graph are either ***directed*** or ***undirected***. An edge (u, v) is said to be ***directed*** from u to v if the pair (u, v) is ordered, with u preceding v. An edge (u, v) is said to be ***undirected*** if the pair (u, v) is not ordered. Undirected edges are sometimes denoted with set notation, as $\{u, v\}$, but for simplicity we use the pair notation (u, v), noting that in the undirected case (u, v) is the same as (v, u). Graphs are typically visualized by drawing the vertices as circles or rectangles and the edges as segments or curves connecting pairs of these circles or rectangles.

Example 13.1: *We can visualize collaborations among the researchers of a certain discipline by constructing a graph whose vertices are associated with the researchers themselves, and whose edges connect pairs of vertices associated with researchers who have coauthored a paper or book. (See Figure 13.1.) Such edges are undirected because coauthorship is a* **symmetric relation***; that is, if A has coauthored something with B, then B necessarily has coauthored something with A.*

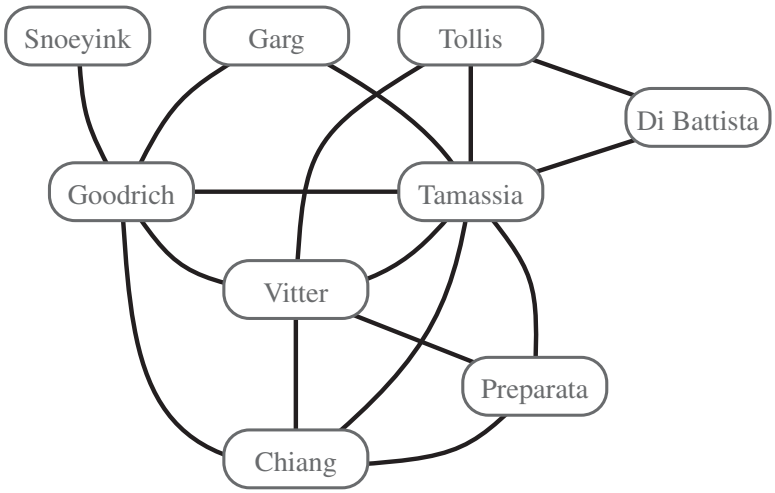


Figure 13.1: Graph of coauthorships among some authors.

13.1.1 Some Graph Terminology

If all the edges in a graph are undirected, then we say the graph is an ***undirected graph***. Likewise, a ***directed graph***, also called a ***digraph***, is a graph whose edges are all directed. A graph that has both directed and undirected edges is often called a ***mixed graph***. Note that an undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u, v) by the pair of directed edges (u, v) and (v, u). It is often useful, however, to keep undirected and mixed graphs represented as they are, for such graphs have several applications.

Example 13.2: *We can associate with an object-oriented program a graph whose vertices represent the classes defined in the program, and whose edges indicate inheritance between classes. There is an edge from a vertex v to a vertex u if the class for v extends the class for u. Such edges are directed because the inheritance relation only goes in one direction (that is, it is* **asymmetric***).*

Example 13.3: *A city map can be modeled by a graph whose vertices are intersections or dead ends, and whose edges are stretches of streets without intersections. This graph has both undirected edges, which correspond to stretches of two-way streets, and directed edges, which correspond to stretches of one-way streets. Thus, a graph modeling a city map is a mixed graph.*

Example 13.4: *Physical examples of graphs are present in the electrical wiring and plumbing networks of a building. Such networks can be modeled as graphs, where each connector, fixture, or outlet is viewed as a vertex, and each uninterrupted stretch of wire or pipe is viewed as an edge. Such graphs are actually components of much larger graphs, namely the local power and water distribution networks. Depending on the specific aspects of these graphs that we are interested in, we may consider their edges as undirected or directed, for, in principle, water can flow in a pipe and current can flow in a wire in either direction.*

The two vertices joined by an edge are called the ***end vertices*** of the edge. The end vertices of an edge are also known as the ***endpoints*** of that edge. If an edge is directed, its first endpoint is its ***origin*** and the other is the ***destination*** of the edge.

Two vertices are said to be ***adjacent*** if they are endpoints of the same edge. An edge is said to be ***incident*** on a vertex if the vertex is one of the edge's endpoints. The ***outgoing edges*** of a vertex are the directed edges whose origin is that vertex. The ***incoming edges*** of a vertex are the directed edges whose destination is that vertex. The ***degree*** of a vertex v, denoted $\deg(v)$, is the number of incident edges of v. The ***in-degree*** and ***out-degree*** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted $\text{indeg}(v)$ and $\text{outdeg}(v)$, respectively.

Example 13.5: *We can study air transportation by constructing a graph G, called a* **flight network***, whose vertices are associated with airports, and whose edges are associated with flights. (See Figure 13.2.) In graph G, the edges are directed because a given flight has a specific travel direction (from the origin airport to the*

destination airport). The endpoints of an edge e in G correspond respectively to the origin and destination for the flight corresponding to e. Two airports are adjacent in G if there is a flight that flies between them, and an edge e is incident upon a vertex v in G if the flight for e flies to or from the airport for v. The outgoing edges of a vertex v correspond to the out-bound flights from v's airport, and the incoming edges correspond to the in-bound flights to v's airport. Finally, the in-degree of a vertex v of G corresponds to the number of in-bound flights to v's airport, and the out-degree of a vertex v in G corresponds to the number of out-bound flights.

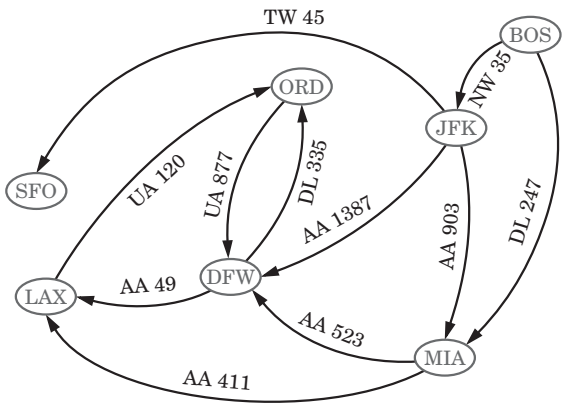


Figure 13.2: Example of a directed graph representing a flight network. The endpoints of edge UA 120 are LAX and ORD; hence, LAX and ORD are adjacent. The in-degree of DFW is 3, and the out-degree of DFW is 2.

The definition of a graph groups edges in a ***collection***, not a ***set***, thus allowing for two undirected edges to have the same end vertices, and for two directed edges to have the same origin and destination. Such edges are called ***parallel edges*** or ***multiple edges***. Parallel edges may exist in a flight network (Example 13.5), in which case multiple edges between the same pair of vertices could indicate different flights operating on the same route at different times of the day. Another special type of edge is one that connects a vertex to itself. In this case, we say that an edge (undirected or directed) is a ***self-loop*** if its two endpoints coincide. A self-loop may occur in a graph associated with a city map (Example 13.3), where it would correspond to a "circle" (a curving street that returns to its starting point).

With few exceptions, like those mentioned above, graphs do not have parallel edges or self-loops. Such graphs are said to be ***simple***. Thus, we can usually say that the edges of a simple graph are a ***set*** of vertex pairs (and not just a collection). Throughout this chapter, we shall assume that a graph is simple unless otherwise specified. This assumption simplifies the presentation of data structures and algorithms for graphs. Extending the results of this chapter to general graphs, with self-loops and/or parallel edges, is straightforward but tedious.

In the theorems that follow, we explore a few important properties of degrees and the number of edges in a graph. These properties relate the number of vertices and edges to each other and to the degrees of the vertices in a graph.

Theorem 13.6: *If G is a graph with m edges, then*

$$\sum_{v \in G} \mathit{deg}(v) = 2m.$$

Proof: An edge (u, v) is counted twice in the above summation: once by its endpoint u and once by its endpoint v. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges. ■

Theorem 13.7: *If G is a directed graph with m edges, then*

$$\sum_{v \in G} \mathit{indeg}(v) = \sum_{v \in G} \mathit{outdeg}(v) = m.$$

Proof: In a directed graph, an edge (u, v) contributes one unit to the out-degree of its origin u and one unit to the in-degree of its destination v. Thus, the total contribution of the edges to the out-degrees of the vertices is equal to the number of edges, and similarly for the in-degrees. ■

Theorem 13.8: *Let G be a simple graph with n vertices and m edges. If G is undirected, then $m \leq n(n-1)/2$, and if G is directed, then $m \leq n(n-1)$.*

Proof: Suppose that G is undirected. Since no two edges can have the same endpoints and there are no self-loops, the maximum degree of a vertex in G is $n-1$ in this case. Thus, by Theorem 13.6, $2m \leq n(n-1)$. Now suppose that G is directed. Since no two edges can have the same origin and destination, and there are no self-loops, the maximum in-degree of a vertex in G is $n-1$ in this case. Thus, by Theorem 13.7, $m \leq n(n-1)$. ■

Put another way, Theorem 13.8 states that a simple graph with n vertices has $O(n^2)$ edges.

A ***path*** in a graph is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex, such that each edge is incident to its predecessor and successor vertex. A ***cycle*** is a path with the same start and end vertices. We say that a path is ***simple*** if each vertex in the path is distinct, and we say that a cycle is ***simple*** if each vertex in the cycle is distinct, except for the first and last one. A ***directed path*** is a path such that all the edges are directed and are traversed along their direction. A ***directed cycle*** is defined similarly.

Example 13.9: *Given a graph G representing a city map (see Example 13.3), we can model a couple driving from their home to dinner at a recommended restaurant as traversing a path though G. If they know the way, and don't accidentally go*

through the same intersection twice, then they traverse a simple path in G. Likewise, we can model the entire trip the couple takes, from their home to the restaurant and back, as a cycle. If they go home from the restaurant in a completely different way than how they went, not even going through the same intersection twice, then their entire round trip is a simple cycle. Finally, if they travel along one-way streets for their entire trip, then we can model their night out as a directed cycle.

A ***subgraph*** of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G, respectively. A ***spanning subgraph*** of G is a subgraph of G that contains all the vertices of the graph G. A graph is ***connected*** if, for any two vertices, there is a path between them. If a graph G is not connected, its maximal connected subgraphs are called the ***connected components*** of G. A ***forest*** is a graph without cycles. A ***tree*** is a connected forest, that is, a connected graph without cycles.

Note that this definition of a tree is somewhat different from the one given in Section 2.3. Namely, in the context of graphs, a tree has no root. Whenever there is ambiguity, the trees of Section 2.3 should be called ***rooted trees***, while the trees of this chapter should be called ***free trees***. The connected components of a forest are (free) trees. A ***spanning tree*** of a graph is a spanning subgraph that is a (free) tree.

Example 13.10: *Perhaps the most talked about graph today is the Internet, which can be viewed as a graph whose vertices are computers and whose (undirected) edges are communication connections between pairs of computers on the Internet. The computers and the connections between them in a single domain, like* wiley.com, *form a subgraph of the Internet. If this subgraph is connected, then two users on computers in this domain can send e-mail to one another without having their information packets ever leave their domain. Suppose the edges of this subgraph form a spanning tree. This implies that, even if a single connection goes down (for example, because someone pulls a communication cable out of the back of a computer in this domain), then this subgraph will no longer be connected.*

There are a number of simple properties of trees, forests, and connected graphs.

Theorem 13.11: *Let G be an undirected graph with n vertices and m edges. Then we have the following:*

- *If G is connected, then $m \geq n - 1$.*
- *If G is a tree, then $m = n - 1$.*
- *If G is a forest, then $m \leq n - 1$.*

We leave the justification of this theorem as an exercise (C-13.1).

13.1.2 Operations on Graphs

In spite of their simplicity, graphs are a rich abstraction. This richness derives partly from the fact that graphs contain two kinds of objects—vertices and edges—and also because edges can be directed or undirected. There are, therefore, a number of operations that we can consider performing for a graph, G, including the following:

- Return the number, n, of vertices in G.
- Return the number, m, of edges in G.
- Return a set or list containing all n vertices in G.
- Return a set or list containing all m edges in G.
- Return some vertex, v, in G.
- Return the degree, deg(v), of a given vertex, v, in G.
- Return a set or list containing all the edges incident upon a given vertex, v, in G.
- Return a set or list containing all the vertices adjacent to a given vertex, v, in G.
- Return the two end vertices of an edge, e, in G; if e is directed, indicate which vertex is the origin of e and which is the destination of e.
- Return whether two given vertices, v and w, are adjacent in G.

When we allow for some or all the edges in a graph to be directed, then there are several additional methods we should consider including in the set of operations that could be supported by a graph, such as the following:

- Indicate whether a given edge, e, is directed in G.
- Return the in-degree of v, inDegree(v).
- Return a set or list containing all the incoming (or outgoing) edges incident upon a given vertex, v, in G.
- Return a set or list containing all the vertices adjacent to a given vertex, v, along incoming (or outgoing) edges in G.

We can also allow for update methods that add or delete edges and vertices, such as the following:

- Insert a new directed (or undirected) edge, e, between two given vertices, v and w, in G.
- Insert a new (isolated) vertex, v, in G.
- Remove a given edge, e, from G.
- Remove a given vertex, v, and all its incident edges from G.

In addition, we can allow any edge or vertex to store additional information, including numeric weights, Boolean values, or even pointers to general objects.

There are admittedly a lot of operations than one can perform with a graph, and the above list is not even exhaustive. The number of operations is to a certain extent unavoidable, however, since graphs are such rich structures.

13.1.3 Data Structures for Representing Graphs

There are two data structures that people often use to represent graphs, the ***adjacency list*** and the ***adjacency matrix***. In both of these representations, if vertices or edges are used to store data, then we assume there is some way of mapping vertices and edges to data that is associated with them. For example, we may use a lookup structure to store objects using vertex or edge names as keys, or we may represent edges or vertices as multiple-field objects and store the data associated with the edges or vertices in these fields. As we explain below, the main difference between these two graph representations are that we may get different time performances for various graph operations depending on how our graph is represented. Also, for a graph G with n vertices and m edges, an adjacency list representation uses $O(n + m)$ space, whereas an adjacency matrix representation uses $O(n^2)$ space.

The Adjacency List Structure

The ***adjacency list*** structure for a graph, G, includes the following components:

- A collection, V, of n vertices. This collection could be a set, list, or array, or it could even be defined implicitly as simply the integers from 1 to n. If vertices can "store" data, there also needs to be some way to map each vertex, v, to the data associated with v.
- A collection, E, of m edges, that is, pairs of vertices. This collection could be a set, list, or array, or it could even be defined implicitly by the pairs of vertices that are determined by adjacency lists. If edges can "store" data, there also needs to be some way to map each edge, e, to the data associated with e.
- For each vertex, v, in V, we store a list, called the ***adjacency list for*** v, that represents all the edges incident on v. This is implemented either as a list of references to each vertex, w, such that (v, w) is an edge in E, or it is implemented as a list of references to each edge, e, that is incident on v. If G is a directed graph, then the adjacency list for v is typically divided into two parts—one representing the incoming edges for v and one representing the outgoing edges for v.

We illustrate the adjacency list structure of a directed graph in Figure 13.3. For a vertex v, the space used by the adjacency list for v is proportional to the degree of v, that is, it is $O(\deg(v))$. Thus, by Theorem 13.6, the space requirement of the adjacency list structure for a graph, G, of n vertices and m edges is $O(n + m)$.

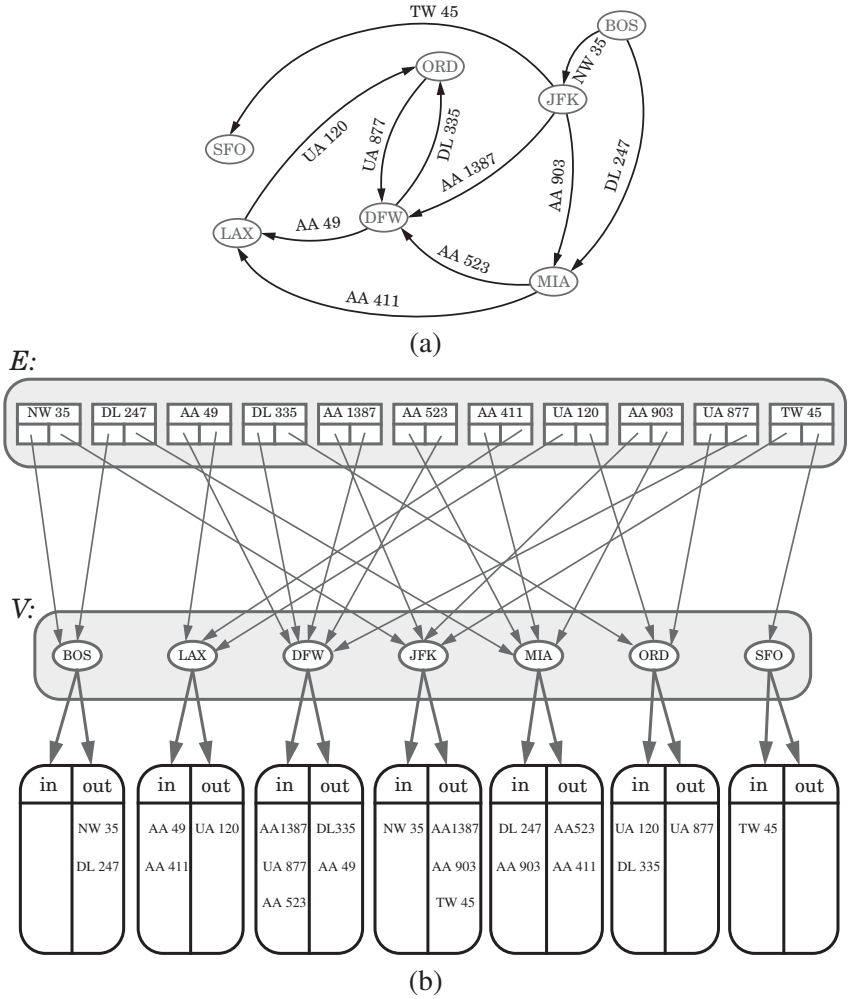


Figure 13.3: (a) A directed graph G; (b) a schematic representation of the adjacency list structure of G. In this example, we have a set of vertex objects and set of edge objects. Each edge object has pointers to its two end vertices and each vertex object has pointers to the two parts of its adjacency list, which store references to incident edges, one part for incoming edges and one for outgoing edges.

In addition, the adjacency list structure has the following performance properties:

- Returning the incident edges or adjacent vertices for a vertex, v, run in $O(\deg(v))$ time.
- Determining whether two vertices, u and v, are adjacent can be performed by inspecting either the adjacency list for u or that of v. By choosing the smaller of the two, we get $O(\min\{\deg(u), \deg(v)\})$ running time for this operation.

The Adjacency Matrix Structure

In the ***adjacency matrix*** representation of a graph, G, we represent the edges in G using (a two-dimensional array) matrix, A. This representation allows us to determine adjacencies between pairs of vertices in constant time. As we shall see, the trade-off in achieving this speedup is that the space usage for representing a graph of n vertices is $O(n^2)$, even if the graph has few edges.

In the adjacency matrix representation, we number the vertices, $1, 2, \ldots, n$, and we view the edges as being pairs of such integers. Historically, the adjacency matrix was the first representation used for graphs, with the adjacency matrix being a Boolean $n \times n$ matrix, A, defined as follows:

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge in } G \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the adjacency matrix has a natural appeal as a mathematical structure (for example, an undirected graph has a symmetric adjacency matrix).

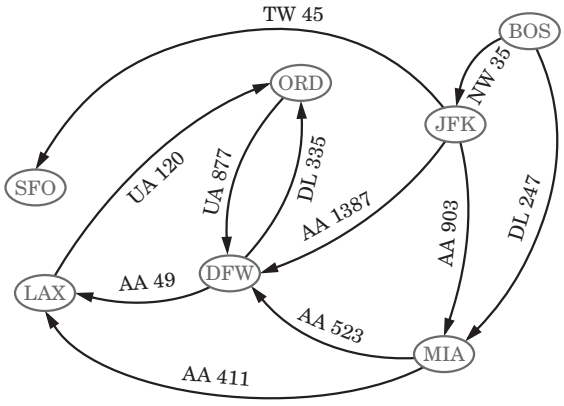
Modern instances of an adjacency matrix representation often update this historical perspective slightly to follow an object-oriented framework. In this case, we represent a graph, G, with an $n \times n$ array, A, such that $A[i,j]$ stores a reference to an edge object, e, if there is an edge $e = (i,j)$ in G. If there is no edge, (i,j) in G, then $A[i,j]$ is null.

In addition, if vertices or edges have some kind of data that is associated with them, then we would also need some way of mapping vertex numbers to vertex data and vertex pairs, (i,j), to associated edge data.

Using an adjacency matrix A, we can determine whether two vertices, v and w, are adjacent in $O(1)$ time. We can achieve this performance by accessing the vertices v and w to determine their respective indices i and j, and then testing whether the cell $A[i,j]$ is null or not. This performance achievement is traded off by an increase in the space usage, however, which is now $O(n^2)$, and in the running time of some other graph operations as well. For example, listing out the incident edges or adjacent vertices for a vertex v now requires that we examine an entire row or column of the array, A, representing the graph, which takes $O(n)$ time.

Deciding which representation to use for a particular graph, G, typically boils down to determining how ***dense*** G is. For instance, if G has close to a quadratic number of edges, then the adjacency matrix is often a good choice for representing G, but if G has close to a linear number of edges, then the adjacency list representation is probably superior. The graph algorithms we examine in this chapter tend to run most efficiently when acting upon a graph stored using an adjacency list representation.

We illustrate an example adjacency matrix in Figure 13.4.



(a)

0	1	2	3	4	5	6
BOS	DFW	JFK	LAX	MIA	ORD	SFO

(b)

	0	1	2	3	4	5	6
0	Ø	Ø	NW 35	Ø	DL 247	Ø	Ø
1	Ø	Ø	Ø	AA 49	Ø	DL 335	Ø
2	Ø	AA 1387	Ø	Ø	AA 903	Ø	TW 45
3	Ø	Ø	Ø	Ø	Ø	UA 120	Ø
4	Ø	AA 523	Ø	AA 411	Ø	Ø	Ø
5	Ø	UA 877	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

(c)

Figure 13.4: Schematic representation of an adjacency matrix structure. (a) A directed graph G; (b) a numbering of its vertices; (c) the adjacency matrix A for G, using a modern object-oriented viewpoint, where each cell, $A[i, j]$, holds a pointer to the edge object, $e = (i, j)$, or is null if there is no such edge in G.

13.2 Depth-First Search

In this section, we explore a fundamental kind of algorithmic operation that we might wish to perform on a graph—traversing the edges and the vertices of that graph. Specifically, a ***traversal*** is a systematic procedure for exploring a graph by examining all of its vertices and edges. For example, a web ***spider***, or ***crawler***, which is the data collecting part of a search engine, must explore a graph of hypertext documents by examining its vertices, which are the documents, and its edges, which are the hyperlinks between documents. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.

The first traversal algorithm we consider is ***depth-first search*** (DFS) in an undirected graph. Depth-first search is useful for performing a number of computations on graphs, including finding a path from one vertex to another, determining whether a graph is connected, and computing a spanning tree of a connected graph.

Traversing a Graph Using the Backtracking Technique

Depth-first search in an undirected graph G applies the ***backtracking*** technique and is analogous to wandering in a labyrinth with a string and a can of paint without getting lost. We begin at a specific starting vertex s in G, which we initialize by fixing one end of our string to s and painting s as "explored." The vertex s is now our "current" vertex—call our current vertex v. We then traverse G by considering an (arbitrary) edge (v, w) incident to the current vertex, v. If the edge (v, w) leads us to an already explored (that is, painted) vertex w, then we immediately backtrack to vertex v. If, on the other hand, (v, w) leads to an unexplored vertex, w, then we unroll our string, and go to w. We then paint w as "explored" and make it the current vertex, repeating the above computation. Eventually, we will get to a "dead end," that is, a current vertex, v, such that all the edges incident on v lead to vertices already explored. To get out of this impasse, we roll our string back up, backtracking along the edge that brought us to v, going back to a previously visited vertex, u. We then make u our current vertex and repeat the above computation for any edges incident upon u that we have not looked at before. If all of u's incident edges lead to visited vertices, then we again roll up our string and backtrack to the vertex we came from to get to u, and repeat the procedure at that vertex. Thus, we continue to backtrack along the path that we have traced so far until we find a vertex that has yet unexplored edges, at which point we take one such edge and continue the traversal. The process terminates when our backtracking leads us back to the start vertex, s, and there are no more unexplored edges incident on s. This simple process traverses the edges of G in an elegant, systematic way. (See Figure 13.5.)

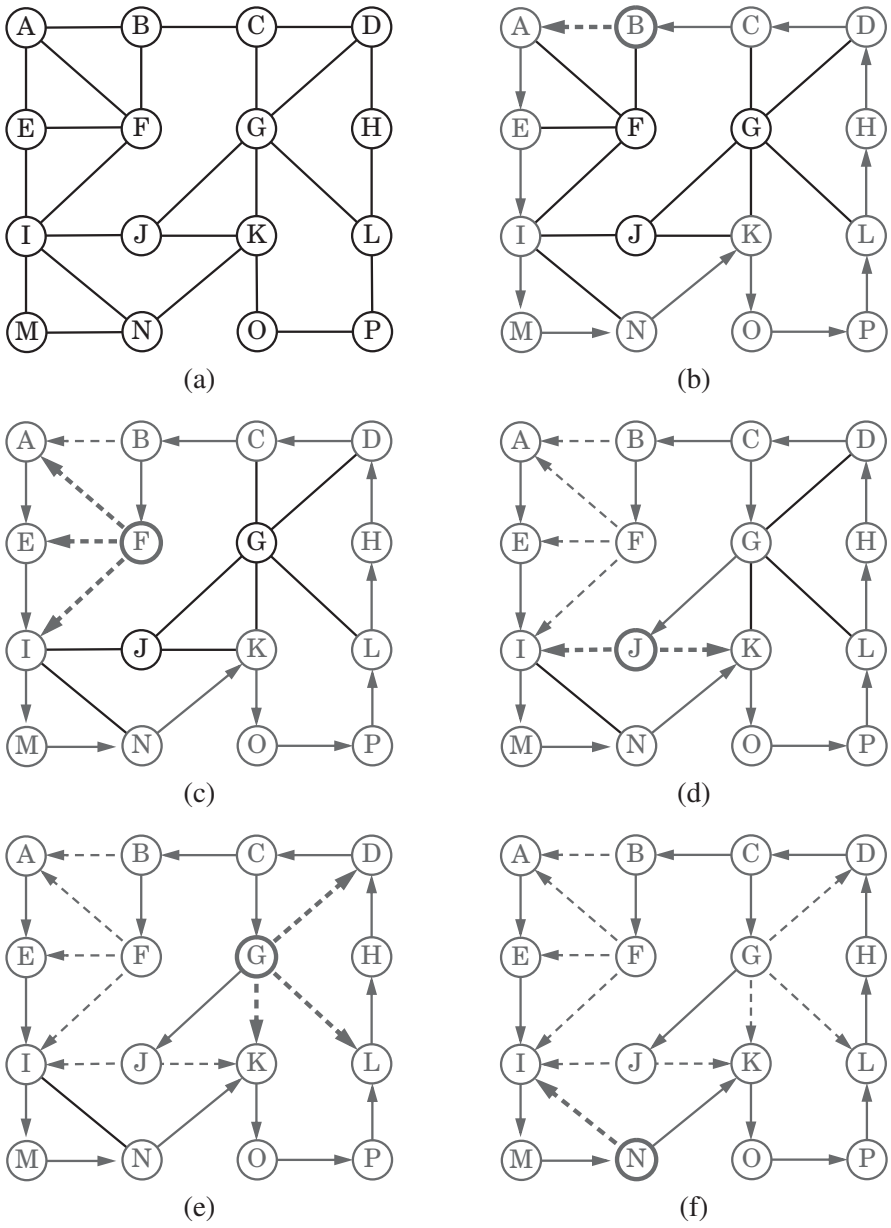


Figure 13.5: Example of depth-first search traversal on a graph starting at vertex A. Discovery edges are drawn with solid lines and back edges are drawn with dashed lines. The current vertex is drawn with a thick line: (a) input graph; (b) path of discovery edges traced from A until back edge (B,A) is hit; (c) reaching F, which is a dead end; (d) after backtracking to C, resuming with edge (C,G), and hitting another dead end, J; (e) after backtracking to G; (f) after backtracking to N.

Visualizing Depth-First Search

We can visualize a DFS traversal by orienting the edges along the direction in which they are explored during the traversal, distinguishing the edges used to discover new vertices, called ***discovery edges***, or ***tree edges***, from those that lead to already explored vertices, called ***back edges***. (See Figure 13.5f.) In the analogy above, discovery edges are the edges where we unroll our string when we traverse them, and back edges are the edges where we immediately return without unrolling any string. The discovery edges form a spanning tree of the connected component of the starting vertex s, called ***DFS tree***. We call the edges not in the DFS tree "back edges," because, assuming that the DFS tree is rooted at the start vertex, each such edge leads back from a vertex in this tree to one of its ancestors in the tree.

Recursive Depth-First Search

The pseudocode for a DFS traversal starting at a vertex v follows our analogy with string and paint based on the backtracking technique. We use recursion to implement this approach. We assume that we have a mechanism (similar to the painting analogy) to determine if a vertex or edge has been explored or not, and to label the edges as discovery edges or back edges.

A pseudocode description of recursive DFS is given in Algorithm 13.6.

Algorithm DFS(G, v):
Input: A graph G and a vertex v in G
Output: A labeling of the edges in the connected component of v as discovery edges and back edges, and the vertices in the connected component of v as explored

```
Label v as explored
for each edge, e, that is incident to v in G do
    if e is unexplored then
        Let w be the end vertex of e opposite from v
        if w is unexplored then
            Label e as a discovery edge
            DFS(G, w)
        else
            Label e as a back edge
```

Algorithm 13.6: A recursive description of the DFS algorithm for searching from a vertex, v.

Properties of the Depth-First Search Algorithm

There are a number of observations that we can make about the depth-first search algorithm, many of which derive from the way the DFS algorithm partitions the edges of the undirected graph G into two groups, the discovery edges and the back edges. For example, since back edges always connect a vertex v to a previously visited vertex u, each back edge implies a cycle in G, consisting of the discovery edges from u to v plus the back edge (v, u).

Theorem 13.12, which follows, identifies some other important properties of the depth-first search traversal method.

Theorem 13.12: *Let G be an undirected graph on which a DFS traversal starting at a vertex s has been performed. Then the traversal visits all the vertices in the connected component of s, and the discovery edges form a spanning tree of the connected component of s.*

Proof: Suppose, for the sake of a contradiction, there is at least one vertex v in s's connected component not visited. Let w be the first unvisited vertex on some path from s to v (we may have $v = w$). Since w is the first unvisited vertex on this path, it has a neighbor u that was visited. But when we visited u, we must have considered the edge (u, w); hence, it cannot be correct that w is unvisited. Therefore, there are no unvisited vertices in s's connected component. Since we only mark edges when we go to unvisited vertices, we will never form a cycle with discovery edges, that is, the discovery edges form a tree. Moreover, this is a spanning tree because the depth-first search visits each vertex in the connected component of s. ■

The depth-first search algorithm for searching all the vertices in a graph, G, is shown in Algorithm 13.7.

Algorithm DFS(G):
Input: A graph G
Output: A labeling of the vertices in each connected component of G as explored

```
Initially label each vertex in v as unexplored
for each vertex, v, in G do
    if v is unexplored then
        DFS(G, v)
```

Algorithm 13.7: The DFS algorithm for searching an entire graph, G. Each time we make a call, DFS(G, v), to the recursive depth-first algorithm for searching from a vertex, v, we traverse a different connected component of G.

Analyzing Depth-First Search

In terms of its running time, depth-first search is an efficient method for traversing a graph. Note that DFS is called exactly once on each vertex, and that every edge is examined exactly twice, once from each of its end vertices. Let m_s denote the number of edges in the connected component of vertex s. A DFS starting at s runs in $O(m_s)$ time provided the following conditions are satisfied:

- The graph is represented by a data structure that allows us to find all the incident edges for a vertex, v, in $O(\mathsf{degree}(v))$ time. Note that the adjacency list structure satisfies this property, the adjacency matrix structure does not.
- We need to have a way to "mark" a vertex or edge as explored, and to test if a vertex or edge has been explored in $O(1)$ time. One way to do such marking is to design vertex objects and edge objects so that they each contain a visited flag. Another way is to use an auxiliary hash table to store the explored vertices and edges, and have a Boolean explored value associated with the vertex and edge keys in this hash table. Yet another way is to design vertex and edge objects so that they each have a small hash table associated with them that can be used to add "decorator" fields to these objects.

By Theorem 13.12, we can use DFS to solve a number of interesting problems for an undirected graph, as shown in the following theorem.

Theorem 13.13: *Let G be a graph with n vertices and m edges represented with the adjacency list structure. A DFS traversal of G can be performed in $O(n + m)$ time. Also, there exist $O(n + m)$-time algorithms based on DFS for the following problems:*

- *Testing whether G is connected*
- *Computing a spanning forest of G*
- *Computing the connected components of G*
- *Computing a path between two vertices of G, or reporting that no such path exists*
- *Computing a cycle in G, or reporting that G has no cycles.*

The justification of Theorem 13.13 is based on algorithms that use the depth-first search algorithm or a slightly modified versions of the DFS algorithm. We explore the details of the proof of this theorem in several exercises.

13.3 Breadth-First Search

In this section, we consider the ***breadth-first search*** (BFS) traversal algorithm. Like DFS, BFS traverses a connected component of a graph, and in so doing, defines a useful spanning tree. Instead of searching recursively, however, BFS proceeds in rounds and subdivides the vertices into ***levels***, which represent the minimum number of edges from the start vertex to each vertex.

BFS starts at a given start vertex, s, which is at level 0 and defines the "anchor" for our string. In the first round, we explore all the vertices we can reach in one edge, marking each as explored. These vertices are placed into level 1. In the second round, we explore all the vertices that can be reached in two edges from the start vertex. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2, and so on. The BFS traversal terminates when every vertex has been visited. Pseudo-code for a BFS traversal starting at a vertex s is shown in Algorithm 13.8. We use auxiliary space to label edges, mark visited vertices, and store lists associated with levels. That is, the lists L_0, L_1, L_2, and so on, store the nodes that are in level 0, level 1, level 2, and so on.

Algorithm BFS(G, s):
Input: A graph G and a vertex s of G
Output: A labeling of the edges in the connected component of s as discovery edges and cross edges

```
Create an empty list, L_0
Mark s as explored and insert s into L_0
i ← 0
while L_i is not empty do
    create an empty list, L_{i+1}
    for each vertex, v, in L_i do
        for each edge, e = (v, w), incident on v in G do
            if edge e is unexplored then
                if vertex w is unexplored then
                    Label e as a discovery edge
                    Mark w as explored and insert w into L_{i+1}
                else
                    Label e as a cross edge
    i ← i + 1
```

Algorithm 13.8: BFS traversal of a graph.

We illustrate a BFS traversal in Figure 13.9.

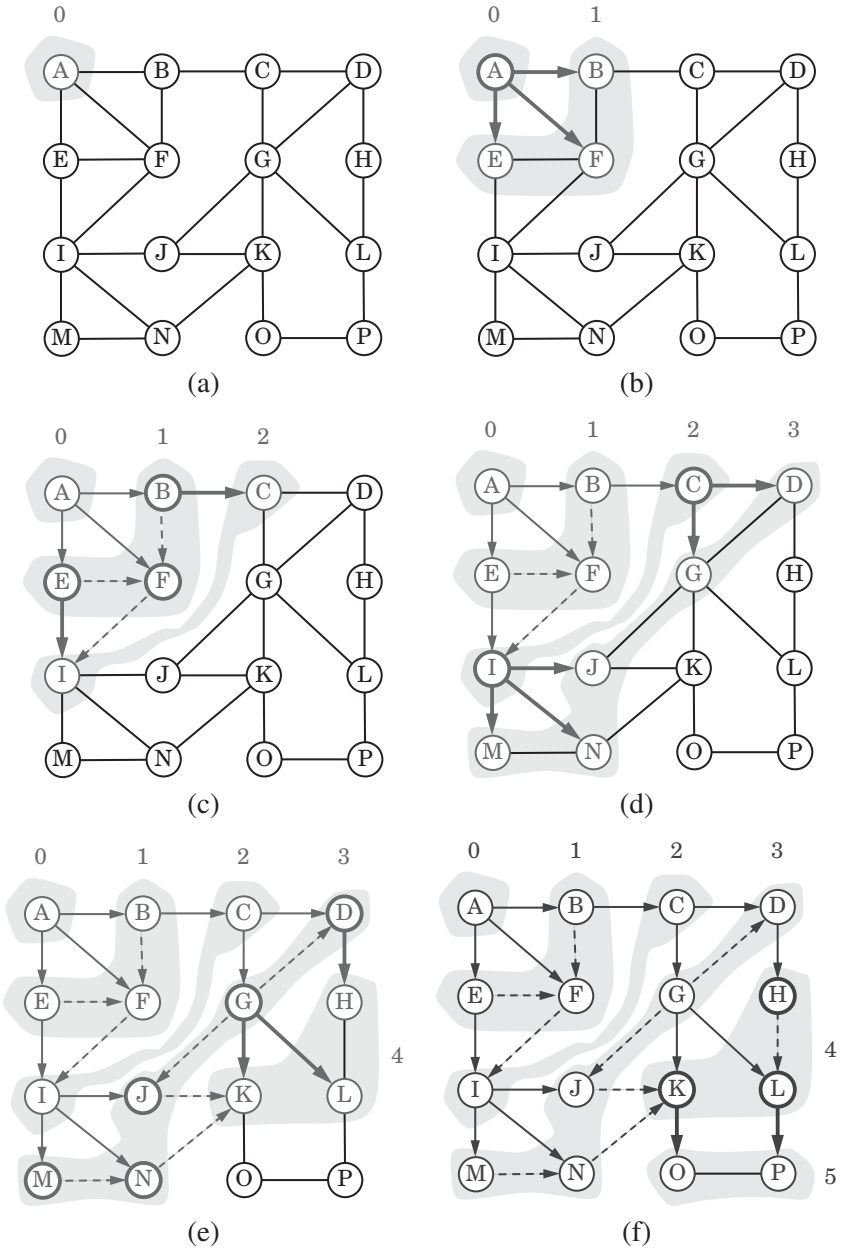


Figure 13.9: Example of breadth-first search traversal. The discovery edges are shown with solid lines and the cross edges are shown with dashed lines: (a) graph before the traversal; (b) discovery of level 1; (c) discovery of level 2; (d) discovery of level 3; (e) discovery of level 4; (f) discovery of level 5.

One of the nice properties of the BFS approach is that, in performing the BFS traversal, we can label each vertex by the length of a shortest path (in terms of the number of edges) from the start vertex s. In particular, if vertex v is placed into level i by a BFS starting at vertex s, then the length of a shortest path from s to v is i.

As with DFS, we can visualize the BFS traversal by orienting the edges along the direction in which they are explored during the traversal, and by distinguishing the edges used to discover new vertices, called ***discovery edges***, from those that lead to already visited vertices, called ***cross edges***. (See Figure 13.9f.) As with the DFS, the discovery edges form a spanning tree, which in this case we call the ***BFS tree***. We do not call the nontree edges "back edges" in this case, however, for none of them connects a vertex to one of its ancestors. Every nontree edge connects a vertex v to another vertex that is neither v's ancestor nor its descendant.

The BFS traversal algorithm has a number of interesting properties, some of which we state in the theorem that follows.

Theorem 13.14: *Let G be an undirected graph on which a BFS traversal starting at vertex s has been performed. Then we have the following:*

- *The traversal visits all the vertices in the connected component of s.*
- *The discovery edges form a spanning tree T of the connected component of s.*
- *For each vertex v at level i, the path of tree T between s and v has i edges, and any other path of G between s and v has at least i edges.*
- *If (u, v) is a cross edge, then the level numbers of u and v differ by at most 1.*

We leave the justification of this theorem as an exercise (C-13.17). The analysis of the running time of BFS is similar to the one of DFS.

Theorem 13.15: *Let G be a graph with n vertices and m edges represented with the adjacency list structure. A BFS traversal of G takes $O(n+m)$ time. Also, there exist $O(n + m)$-time algorithms based on BFS for the following problems:*

- *Testing whether G is connected*
- *Computing a spanning forest of G*
- *Computing the connected components of G*
- *Given a start vertex s of G, computing, for every vertex v of G, a path with the minimum number of edges between s and v, or reporting that no such path exists*
- *Computing a cycle in G, or reporting that G has no cycles.*

13.4 Directed Graphs

In this section, we consider issues that are specific to directed graphs. Recall that a directed graph, or ***digraph***, is a graph that has only directed edges.

A fundamental issue with directed graphs is the notion of ***reachability***, which deals with determining where we can get to in a directed graph. For example, in a graph whose vertices represent college courses and whose directed edges represent prerequisites, it is important to know which courses depend on a given other course as an explicit or implicit prerequisite. A traversal in a directed graph always goes along directed paths, that is, paths where all the edges are traversed according to their respective directions. Given vertices u and v of a digraph $\vec{G}$, we say that u ***reaches*** v (and v is ***reachable*** from u) if $\vec{G}$ has a directed path from u to v.

A digraph $\vec{G}$ is ***strongly connected*** if, for any two vertices u and v of $\vec{G}$, u reaches v and v reaches u. A ***directed cycle*** of $\vec{G}$ is a cycle where all the edges are traversed according to their respective directions. (Note that $\vec{G}$ may have a cycle consisting of two edges with opposite direction between the same pair of vertices.) A digraph $\vec{G}$ is ***acyclic*** if it has no directed cycles. (See Figure 13.10 for examples.)

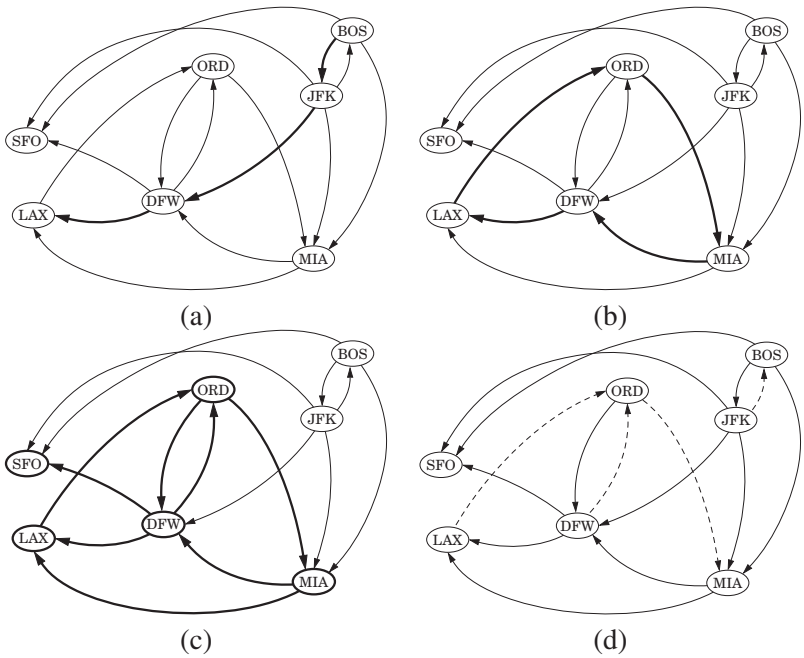


Figure 13.10: Examples of reachability in a digraph: (a) a directed path from BOS to LAX is drawn with thick lines; (b) a directed cycle (ORD, MIA, DFW, LAX, ORD) is drawn with thick lines; its vertices induce a strongly connected subgraph; (c) the subgraph of the vertices and edges reachable from ORD is shown with thick lines; (d) removing the dashed edges gives an acyclic digraph.

The ***transitive closure*** of a digraph, $\vec{G}$, is the digraph $\vec{G}^*$ such that the vertices of $\vec{G}^*$ are the same as the vertices of $\vec{G}$, and $\vec{G}^*$ has an edge (u, v), whenever $\vec{G}$ has a directed path from u to v. That is, we define $\vec{G}^*$ by starting with the digraph, $\vec{G}$, and adding in an extra edge (u, v), for each u and v such that v is reachable from u (and there isn't already an edge (u, v) in $\vec{G}$).

Interesting problems that deal with reachability in a digraph, $\vec{G}$, include finding all the vertices of $\vec{G}$ that are reachable from a given vertex s, determining whether $\vec{G}$ is strongly connected, determining whether $\vec{G}$ is acyclic, and computing the transitive closure $\vec{G}^*$ of $\vec{G}$.

13.4.1 Traversing a Digraph

As with undirected graphs, we can explore a digraph in a systematic way with slight modifications to the depth-first search (DFS) and breadth-first search (BFS) algorithms defined previously for undirected graphs (Sections 13.2 and 13.3). Such explorations can be used, for example, to answer reachability questions. The directed depth-first search and breadth-first search methods we develop in this section for performing such explorations are very similar to their undirected counterparts. In fact, the main difference is that the directed depth-first search and breadth-first search methods only traverse edges according to their respective directions. For instance, see Algorithm 13.11 for a possible extension of DFS to directed graphs (and see Exercise C-13.13 for another).

Algorithm DirectedDFS(G, v):
  Label v as active // Every vertex is initially unexplored
  for each outgoing edge, e, that is incident to v in G **do**
    if e is unexplored **then**
      Let w be the destination vertex for e
      if w is unexplored and not active **then**
        Label e as a discovery edge
        DirectedDFS(G, w)
      else if w is active **then**
        Label e as a back edge
      else
        Label e as a forward/cross edge
  Label v as explored

Algorithm 13.11: A recursive description of the DirectedDFS algorithm for searching from a vertex, v.

We illustrate a directed version of DFS starting at a vertex, BOS, in Figure 13.12.

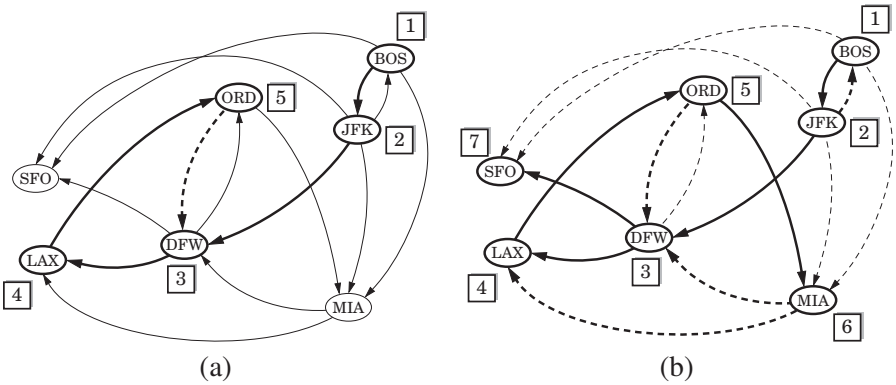


Figure 13.12: An example of a DFS in a digraph: (a) intermediate step, where, for the first time, an already visited vertex (DFW) is reached; (b) the completed DFS. The order in which the vertices are visited is indicated by a label next to each vertex. Discovery edges are shown with thick solid lines, back edges are shown with thick dashed lines, and forward edges are shown with thin dashed lines. For instance, (ORD,DFW) is a back edge and (DFW,ORD) is a forward edge. If there were an edge (SFO,LAX), it would be a cross edge.

A directed DFS on a digraph, $\vec{G}$, partitions the edges of $\vec{G}$ reachable from the starting vertex into ***discovery edges*** or ***tree edges***, which lead us to discover a new vertex, and ***nontree edges***, which take us to a previously visited vertex. The discovery edges form a tree rooted at the starting vertex, called the ***directed DFS tree***. Also, we can distinguish three kinds of nontree edges (see Figure 13.12b):

- ***back edges***, which connect a vertex to an ancestor in the DFS tree
- ***forward edges***, which connect a vertex to a descendant in the DFS tree
- ***cross edges***, which connect a vertex to a vertex that is neither its ancestor nor its descendant.

Theorem 13.16: *Let $\vec{G}$ be a digraph. Depth-first search on $\vec{G}$, starting at a vertex s, visits all the vertices of $\vec{G}$ that are reachable from s. Also, the DFS tree contains directed paths from s to every vertex reachable from s.*

Proof: Let V_s be the subset of vertices of $\vec{G}$ visited by DFS starting at vertex s. We want to show that V_s contains s and every vertex reachable from s belongs to V_s. Suppose, for the sake of a contradiction, that there is a vertex w reachable from s that is not in V_s. Consider a directed path from s to w, and let (u, v) be the first edge on such a path taking us out of V_s, that is, u is in V_s but v is not in V_s. When DFS reaches u, it explores all the outgoing edges of u, and thus must also reach vertex v via edge (u, v). Hence, v should be in V_s, and we have obtained a contradiction. Therefore, V_s must contain every vertex reachable from s. ■

Analyzing the running time of the directed DFS method is analogous to that for its undirected counterpart. A recursive call is made for each vertex exactly once, and each edge is traversed exactly once (along its direction). Hence, if the subgraph reachable from a vertex s has m_s edges, a directed DFS starting at s runs in $O(n_s + m_s)$ time, provided the digraph is represented with an adjacency list.

By Theorem 13.16, we can use DFS to find all the vertices reachable from a given vertex, and hence to find the transitive closure of $\vec{G}$. That is, we can perform a DFS, starting from each vertex v of $\vec{G}$, to see which vertices w are reachable from v, adding an edge (v, w) to the transitive closure for each such w. Likewise, by repeatedly traversing digraph $\vec{G}$ with a DFS, starting in turn at each vertex, we can easily test whether $\vec{G}$ is strongly connected. Therefore, $\vec{G}$ is strongly connected if each DFS visits all the vertices of $\vec{G}$.

Theorem 13.17: *Let $\vec{G}$ be a digraph with n vertices and m edges. The following problems can be solved by an algorithm that runs in $O(n(n + m))$ time:*

- *Computing, for each vertex v of $\vec{G}$, the subgraph reachable from v*
- *Testing whether $\vec{G}$ is strongly connected*
- *Computing the transitive closure $\vec{G}^*$ of $\vec{G}$.*

Testing for Strong Connectivity

Actually, we can determine if a directed graph $\vec{G}$ is strongly connected much faster than $O(n(n + m))$ time, just using two depth-first searches. We begin by performing a DFS of our directed graph $\vec{G}$ starting at an arbitrary vertex s. If there is any vertex of $\vec{G}$ that is not visited by this DFS, and is not reachable from s, then the graph is not strongly connected. So, if this first DFS visits each vertex of $\vec{G}$, then we reverse all the edges of $\vec{G}$ (using the reverseDirection method) and perform another DFS starting at s in this "reverse" graph. If every vertex of $\vec{G}$ is visited by this second DFS, then the graph is strongly connected, for each of the vertices visited in this DFS can reach s. Since this algorithm makes just two DFS traversals of $\vec{G}$, it runs in $O(n + m)$ time.

Directed Breadth-First Search

As with DFS, we can extend breadth-first search (BFS) to work for directed graphs. A pseudocode description is essentially the same as that shown in Algorithm 13.8. The algorithm still visits vertices level by level and partitions the set of edges into ***tree edges*** (or ***discovery edges***), which together form a directed ***breadth-first search*** tree rooted at the start vertex, and ***nontree edges***. Unlike the directed DFS method, however, the directed BFS method only leaves two kinds of nontree edges:

- ***back edges***, which connect a vertex to one of its ancestors
- ***cross edges***, which connect a vertex to another vertex that is neither its ancestor nor its descendant.

Modifying BFS for Directed Graphs

Thus, the only change needed in order to modify the pseudocode description of BFS shown in Algorithm 13.8 to work for directed graphs is to label each nontree edge as a back/cross edge. We explore how to distinguish between back and cross nontree edges with respect to a directed BFS tree in Exercise C-13.11. There are no forward edges, which is a fact we explore in an exercise (C-13.12).

13.4.2 Transitive Closure

In this section, we explore an alternative technique for computing the transitive closure of a digraph. That is, we describe a direct method for determining all pairs of vertices (v, w) in a directed graph such that w is reachable from v. Such information is useful, for example, in computer networks, for it allows us to immediately know if we can route a message from a node v to a node w, or whether it is appropriate to say "you can't get there from here" with respect to this message.

The Floyd-Warshall Algorithm

Let $\vec{G}$ be a digraph with n vertices and m edges. We compute the transitive closure of $\vec{G}$ in a series of rounds. We initialize $\vec{G}_0 = \vec{G}$. We also arbitrarily number the vertices of $\vec{G}$ as

$$v_1, v_2, \ldots, v_n.$$

We then begin the computation of the rounds, beginning with round 1. In a generic round k, we construct digraph $\vec{G}_k$ starting with $\vec{G}_k = \vec{G}_{k-1}$ and adding to $\vec{G}_k$ the directed edge (v_i, v_j) if digraph $\vec{G}_{k-1}$ contains both the edges (v_i, v_k) and (v_k, v_j). In this way, we will enforce a simple rule embodied in the lemma that follows.

Lemma 13.18: *For $i = 1, \ldots, n$, digraph $\vec{G}_k$ has an edge (v_i, v_j) if and only if digraph $\vec{G}$ has a directed path from v_i to v_j, whose intermediate vertices (if any) are in the set $\{v_1, \ldots, v_k\}$. In particular, $\vec{G}_n$ is equal to $\vec{G}^*$, the transitive closure of $\vec{G}$.*

This lemma suggests a simple ***dynamic programming*** algorithm (Chapter 12) for computing the transitive closure of $\vec{G}$, which is known as the ***Floyd-Warshall algorithm***. Pseudo-code for this method is given in Algorithm 13.13.

Algorithm FloydWarshall($\vec{G}$):
Input: A digraph $\vec{G}$ with n vertices
Output: The transitive closure $\vec{G}^*$ of $\vec{G}$
Let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\vec{G}$
$\vec{G}_0 \leftarrow \vec{G}$
for $k \leftarrow 1$ **to** n **do**
  $\vec{G}_k \leftarrow \vec{G}_{k-1}$
  for $i \leftarrow 1$ **to** n, $i \neq k$ **do**
    for $j \leftarrow 1$ **to** n, $j \neq i, k$ **do**
      if both edges (v_i, v_k) and (v_k, v_j) are in $\vec{G}_{k-1}$ **then**
        if $\vec{G}_k$ does not contain directed edge (v_i, v_j) **then**
          add directed edge (v_i, v_j) to $\vec{G}_k$
return $\vec{G}_n$

Algorithm 13.13: The Floyd-Warshall algorithm. This dynamic programming algorithm computes the transitive closure $\vec{G}^*$ of G by incrementally computing a series of digraphs $\vec{G}_0, \vec{G}_1, \ldots, \vec{G}_n$, for $k = 1, \ldots, n$.

Analysis of the Floyd-Warshall Algorithm

The running time of the Floyd-Warshall algorithm is easy to analyze. The main loop is executed n times and the inner loop considers each of $O(n^2)$ pairs of vertices, performing a constant-time computation for each pair. If we use a data structure, such as the adjacency matrix structure, that supports methods areAdjacent and insertDirectedEdge in $O(1)$ time, we have that the total running time is $O(n^3)$. Thus, we have the following.

Theorem 13.19: *Let $\vec{G}$ be a digraph with n vertices represented by the adjacency matrix structure. The Floyd-Warshall algorithm computes the transitive closure $\vec{G}^*$ of $\vec{G}$ in $O(n^3)$ time.*

Let us now compare the running time of the Floyd-Warshall algorithm with that of the more complicated algorithm of Theorem 13.17, which repeatedly performs a DFS n times, starting at each vertex. If the digraph is represented by an adjacency matrix structure, then a DFS traversal takes $O(n^2)$ time (we explore the reason for this in an exercise). Thus, running DFS n times takes $O(n^3)$ time, which is no better than a single execution of the Floyd-Warshall algorithm.

If the digraph is represented by an adjacency list structure, then running the DFS algorithm n times would take $O(n(n + m))$ time. Even so, if the graph is ***dense***, that is, if it has $\Theta(n^2)$ edges, then this approach still runs in $O(n^3)$ time.

Thus, the only case where the algorithm of Theorem 13.17 is better than the Floyd-Warshall algorithm is when the graph is not dense and is represented using an adjacency list structure.

We illustrate an example run of the Floyd-Warshall algorithm in Figure 13.14.

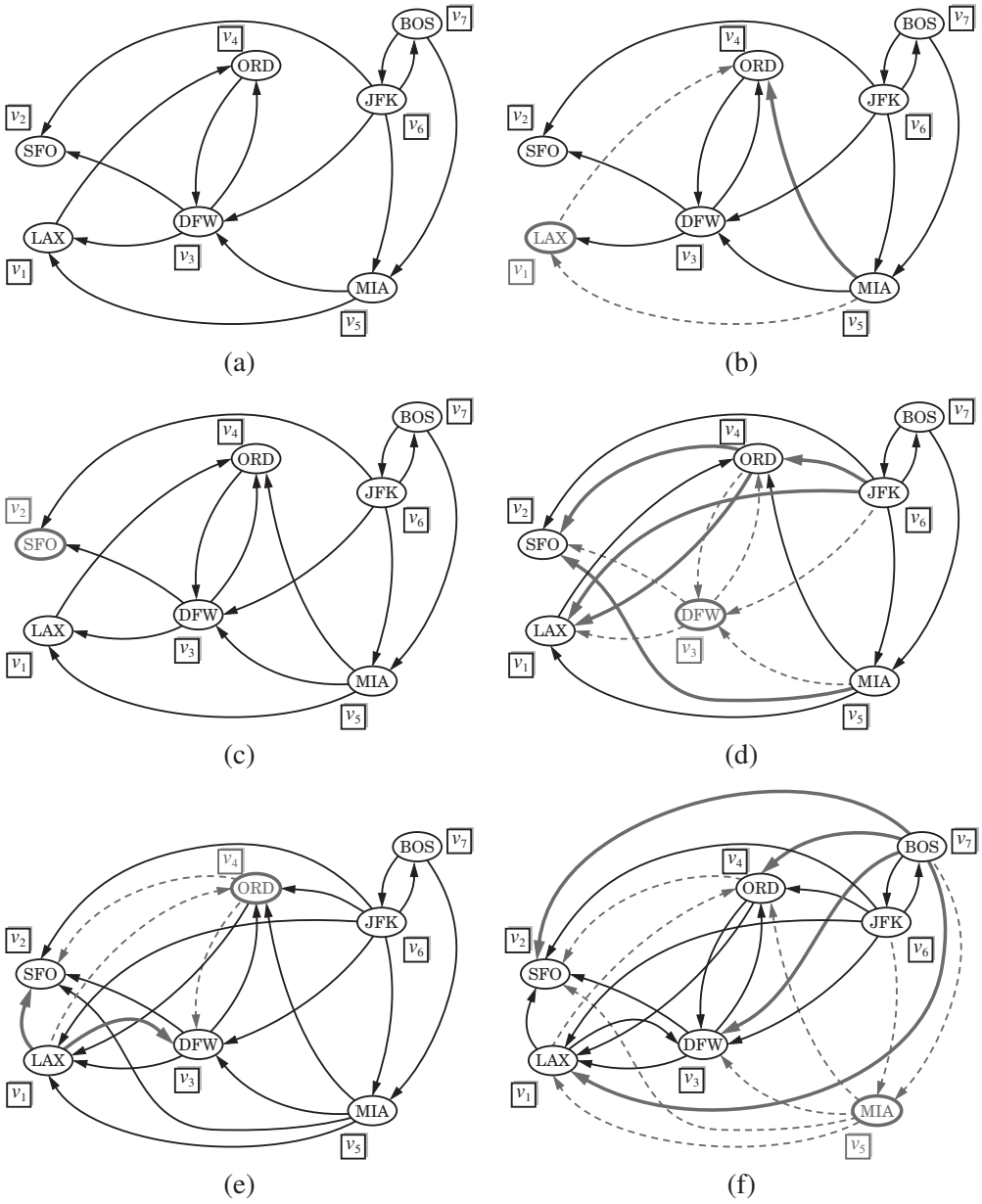


Figure 13.14: Sequence of digraphs computed by the Floyd-Warshall algorithm: (a) initial digraph $\vec{G} = \vec{G}_0$ and numbering of the vertices; (b) digraph $\vec{G}_1$; (c) $\vec{G}_2$; (d) $\vec{G}_3$; (e) $\vec{G}_4$; (f) $\vec{G}_5$. Note that $\vec{G}_5 = \vec{G}_6 = \vec{G}_7$. If digraph $\vec{G}_{k-1}$ has the edges (v_i, v_k) and (v_k, v_j), but not the edge (v_i, v_j), in the drawing of digraph $\vec{G}_k$, we show edges (v_i, v_k) and (v_k, v_j) with dashed thin lines, and edge (v_i, v_j) with a solid thick line.

13.4.3 Directed DFS and Garbage Collection

In some languages, like C and C++, the memory space for objects must be explicitly allocated and deallocated by the programmer. This memory-allocation duty is often overlooked by beginning programmers, and, when done incorrectly, it can even be the source of frustrating programming errors for experienced programmers. Thus, the designers of other languages, like Java, place the burden of memory management on the runtime environment. A Java programmer does not have to explicitly deallocate the memory for some object when its life is over. Instead, a ***garbage collector*** mechanism deallocates the memory for such objects.

In Java, memory for most objects is allocated from a pool of memory called the "memory heap" (not to be confused with the heap data structure). In addition, a running program stores the space for its instance variables in its method stack (Section 2.1.1). Since instance variables in a method stack can refer to objects in the memory heap, all the variables and objects in a method stack is called a ***root object***. All those objects that can be reached by following object references that start from a root object are called ***live objects***. The live objects are the active objects currently being used by the running program; these objects should ***not*** be deallocated. For example, a running Java program may store, in a variable, a reference to a sequence S that is implemented using a doubly linked list. The reference variable to S is a root object, while the object for S is a live object, as are all the node objects that are referenced from this object and all the elements that are referenced from these node objects.

From time to time, the Java virtual machine (JVM) may notice that available space in the memory heap is becoming scarce. At such times, the JVM can elect to reclaim the space that is being used for objects that are no longer live. This reclamation process is known as ***garbage collection***. There are several different algorithms for garbage collection, but one of the most used is the ***mark-sweep algorithm***.

The Mark-Sweep Algorithm

In the mark-sweep garbage collection algorithm, we associate a "mark" bit with each object that identifies if that object is live or not. When we determine at some point that garbage collection is needed, we suspend all other running threads and clear all of the mark bits of objects currently allocated in the memory heap. We then trace through the Java stacks of the currently running threads and we mark all of the (root) objects in these stacks as "live." We must then determine all of the other live objects—the ones that are reachable from the root objects. To do this efficiently, we should use the directed-graph version of the depth-first search traversal. In this case, each object in the memory heap is viewed as a vertex in a directed graph, and the reference from one object to another is viewed as an edge. By performing a directed DFS from each root object, we can correctly identify and mark each live object. This process is known as the "mark" phase. Once this process has

completed, we then scan through the memory heap and reclaim any space that is being used for an object that has not been marked. This scanning process is known as the "sweep" phase, and when it completes, we resume running the suspended threads. Thus, the mark-sweep garbage collection algorithm will reclaim unused space in time proportional to the number of live objects and their references plus the size of the memory heap.

Performing DFS In-place

The mark-sweep algorithm correctly reclaims unused space in the memory heap, but there is an important issue we must face during the mark phase. Since we are reclaiming memory space at a time when available memory is scarce, we must take care not to use extra space during the garbage collection itself. The trouble is that the DFS algorithm, in the recursive way we have described it, can use space proportional to the number of vertices in the graph. In the case of garbage collection, the vertices in our graph are the objects in the memory heap; hence, we don't have this much memory to use. So our only alternative is to find a way to perform DFS in-place rather than recursively, that is, we must perform DFS using only a constant amount of additional storage.

The main idea for performing DFS in-place is to simulate the recursion stack using the edges of the graph (which in the case of garbage collection correspond to object references). Whenever we traverse an edge from a visited vertex v to a new vertex w, we change the edge (v, w) stored in v's adjacency list to point back to v's parent in the DFS tree. When we return back to v (simulating the return from the "recursive" call at w), we can now switch the edge we modified to point back to w. Of course, we need to have some way of identifying which edge we need to change back. One possibility is to number the references going out of v as 1, 2, and so on, and store, in addition to the mark bit (which we are using for the "visited" tag in our DFS), a count identifier that tells us which edges we have modified.

Using a count identifier of course requires an extra word of storage per object. This extra word can be avoided in some implementations, however. For example, many implementations of the Java virtual machine represent an object as a composition of a reference with a type identifier (which indicates if this object is an Integer or some other type) and as a reference to the other objects or data fields for this object. Since the type reference is always supposed to be the first element of the composition in such implementations, we can use this reference to "mark" the edge we changed when leaving an object v and going to some object w. We simply swap the reference at v that refers to the type of v with the reference at v that refers to w. When we return to v, we can quickly identify the edge (v, w) we changed, because it will be the first reference in the composition for v, and the position of the reference to v's type will tell us the place where this edge belongs in v's adjacency list. Thus, whether we use this edge-swapping trick or a count identifier, we can implement DFS in-place without affecting its asymptotic running time.

13.4.4 Directed Acyclic Graphs

A directed graph without cycles is referred to as a ***directed acyclic graph***, or ***dag***, for short. Applications of such graphs include the following:

- Inheritance between C++ classes or Java interfaces
- Prerequisites between courses of a degree program
- Scheduling constraints between the tasks of a project.

Example 13.20: *In order to manage a large project, it is convenient to break it up into a collection of smaller tasks. The tasks, however, are rarely independent, because scheduling constraints exist between them. (For example, in a house building project, the task of ordering nails obviously precedes the task of nailing shingles to the roof deck.) Clearly, scheduling constraints cannot have circularities, because a circularity would make the project impossible. (For example, in order to get a job you need to have work experience, but in order to get work experience you need to have a job.) The scheduling constraints impose restrictions on the order in which the tasks can be executed. Namely, if a constraint says that task a must be completed before task b is started, then a must precede b in the order of execution of the tasks. Thus, if we model a feasible set of tasks as vertices of a directed graph, and we place a directed edge from v to w whenever the task for v must be executed before the task for w, then we define a directed acyclic graph.*

The above example motivates the following definition. Let $\vec{G}$ be a digraph with n vertices. A ***topological ordering*** of $\vec{G}$ is an ordering $(v_1, v_2, \ldots, v_n)$ of the vertices of $\vec{G}$ such that for every edge (v_i, v_j) of $\vec{G}$, $i < j$. That is, a topological ordering is an ordering such that any directed path in $\vec{G}$ traverses vertices in increasing order. (See Figure 13.15.) Note that a digraph may have more than one topological ordering.

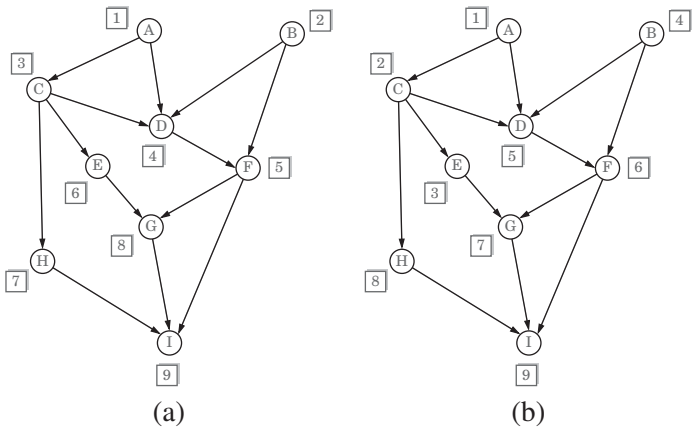


Figure 13.15: Two topological orderings of the same acyclic digraph.

Theorem 13.21: *A digraph has a topological ordering if and only if it is acyclic.*

Proof: The necessity (the "only if" part of the statement) is easy to demonstrate. Suppose $\vec{G}$ is topologically ordered. Assume, for the sake of a contradiction, that $\vec{G}$ has a cycle consisting of edges $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots, (v_{i_{k-1}}, v_{i_0})$. Because of the topological ordering, we must have $i_0 < i_1 < \cdots < i_{k-1} < i_0$, which is clearly impossible. Thus, $\vec{G}$ must be acyclic.

We now argue sufficiency (the "if" part). Suppose $\vec{G}$ is acyclic. We describe an algorithm to build a topological ordering for $\vec{G}$. Since $\vec{G}$ is acyclic, $\vec{G}$ must have a vertex with no incoming edges (that is, with in-degree 0). Let v_1 be such a vertex. Indeed, if v_1 did not exist, then in tracing a directed path from an arbitrary start vertex we would eventually encounter a previously visited vertex, thus contradicting the acyclicity of $\vec{G}$. If we remove v_1 from $\vec{G}$, together with its outgoing edges, the resulting digraph is still acyclic. Hence, the resulting digraph also has a vertex with no incoming edges, and we let v_2 be such a vertex. By repeating this process until $\vec{G}$ becomes empty, we obtain an ordering $v_1, \ldots, v_n$ of the vertices of $\vec{G}$. Because of the above construction, if (v_i, v_j) is an edge of $\vec{G}$, then v_i must be deleted before v_j can be deleted, and thus $i < j$. Thus, $v_1, \ldots, v_n$ is a topological ordering. ■

The above proof suggests Algorithm 13.16, called ***topological sorting***.

Algorithm TopologicalSort($\vec{G}$):
Input: A digraph $\vec{G}$ with n vertices.
Output: A topological ordering $v_1, \ldots, v_n$ of $\vec{G}$ or $\vec{G}$ has a cycle.

```
Let S be an initially empty stack
for each vertex u of G⃗ do
    incounter(u) ← indeg(u)
    if incounter(u) = 0 then
        S.push(u)
i ← 1
while S is not empty do
    u ← S.pop()
    number u as the i-th vertex v_i
    i ← i + 1
    for each edge e ∈ G⃗.outIncidentEdges(u) do
        w ← G⃗.opposite(u, e)
        incounter(w) ← incounter(w) − 1
        if incounter(w) = 0 then
            S.push(w)
if i > n then
    return v_1, ···, v_n
return "digraph G⃗ has a directed cycle"
```

Algorithm 13.16: Topological sorting algorithm.

Theorem 13.22: *Let $\vec{G}$ be a digraph with n vertices and m edges. The topological sorting algorithm runs in $O(n+m)$ time using $O(n)$ auxiliary space, and either computes a topological ordering of $\vec{G}$ or fails to number some vertices, which indicates that $\vec{G}$ has a directed cycle.*

Proof: The initial computation of in-degrees and setup of the incounter variables can be done with a simple traversal of the graph, which takes $O(n+m)$ time. We use an extra field in graph nodes or we use the decorator pattern, described in the next section, to associate counter attributes with the vertices. Say that a vertex u is ***visited*** by the topological sorting algorithm when u is removed from the stack S. A vertex u can be visited only when incounter$(u) = 0$, which implies that all its predecessors (vertices with outgoing edges into u) were previously visited. As a consequence, any vertex that is on a directed cycle will never be visited, and any other vertex will be visited exactly once. The algorithm traverses all the outgoing edges of each visited vertex once, so its running time is proportional to the number of outgoing edges of the visited vertices. Therefore, the algorithm runs in $O(n+m)$ time. Regarding the space usage, observe that the stack S and the incounter variables attached to the vertices use $O(n)$ space.

As a side effect, the algorithm also tests whether the input digraph $\vec{G}$ is acyclic. Indeed, if the algorithm terminates without ordering all the vertices, then the subgraph of the vertices that have not been ordered must contain a directed cycle. (See Figure 13.17.) ■

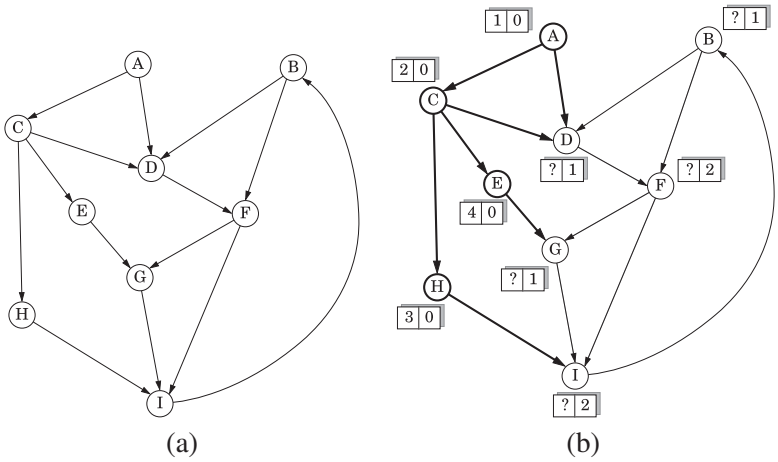


Figure 13.17: Detecting a directed cycle: (a) input digraph; (b) after algorithm TopologicalSort (Algorithm 13.16) terminates, the subgraph of the vertices with undefined number contains a directed cycle.

We visualize the topological sorting algorithm in Figure 13.18.

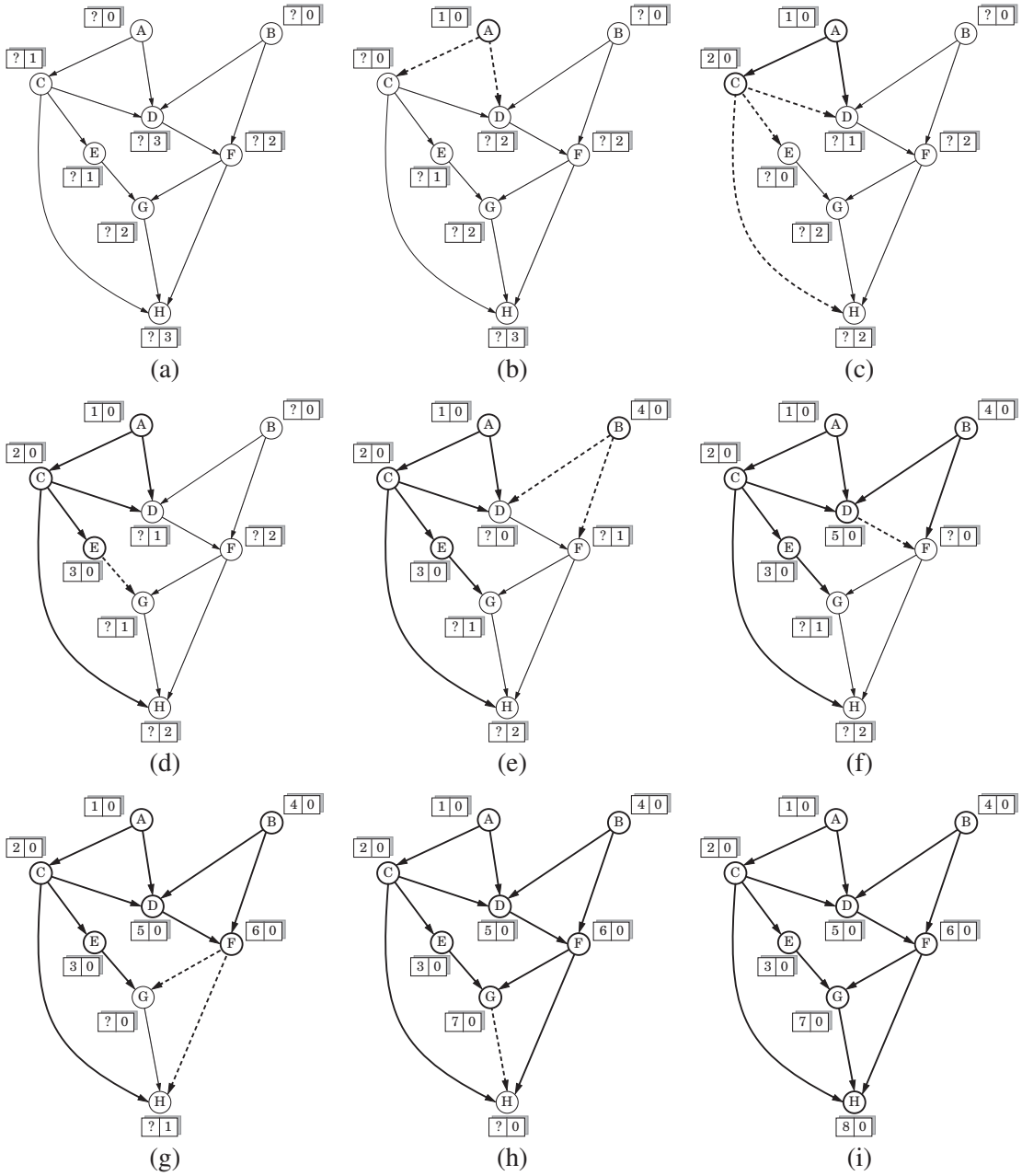


Figure 13.18: Example of a run of algorithm TopologicalSort (Algorithm 13.16): (a) initial configuration; (b–i) after each while-loop iteration. The vertex labels give the vertex number and the current incounter value. The edges traversed in previous iterations are drawn with thick solid lines. The edges traversed in the current iteration are drawn with thick dashed lines.

13.5 Biconnected Components

Let G be a connected undirected graph. A ***separation edge*** of G is an edge whose removal disconnects G. A ***separation vertex*** is a vertex whose removal disconnects G. Separation edges and vertices correspond to single points of failure in a network; hence, we often wish to identify them. A connected graph G is ***biconnected*** if, for any two vertices u and v of G, there are two disjoint paths between u and v, that is, two paths sharing no common edges or vertices, except u and v. A ***biconnected component*** of G is a subgraph satisfying one of the following (see Figure 13.19):

- A subgraph of G that is biconnected and for which adding any additional vertices or edges of G would force it to stop being biconnected
- A single edge of G consisting of a separation edge and its endpoints.

If G is biconnected, it has one biconnected component: G itself. If G has no cycles, on the other hand, then each edge of G is a biconnected component. Biconnected components are important in computer networks, where vertices represent routers and edges represent connections, for even if a router in a biconnected component fails, messages can still be routed in that component using the remaining routers.

As stated in the following lemma, whose proof is left as an exercise (C-13.5), biconnectivity is equivalent to the absence of separation vertices and edges.

Lemma 13.23: *Let G be a connected graph. The following are equivalent:*

1. *G is biconnected.*
2. *For any two vertices of G, there is a simple cycle containing them.*
3. *G does not have separation vertices or separation edges.*

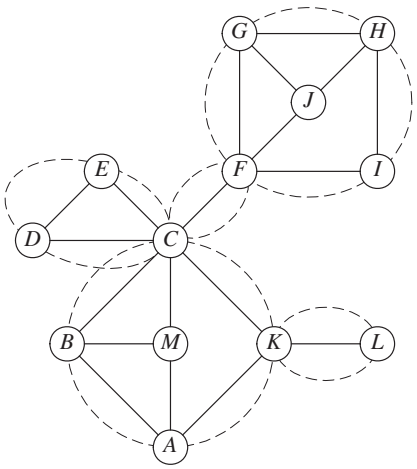


Figure 13.19: Biconnected components, shown circled with dashed lines. C, F, and K are separation vertices; (C, F) and (K, L) are separation edges.

Equivalence Classes and the Linked Relation

Any time we have a collection C of objects, we can define a Boolean relation, $R(x, y)$, for each pair x and y in C. That is, $R(x, y)$ is defined for each x and y in C as being either true or false. The relation R is an ***equivalence relation*** if it has the following properties:

- ***Reflexive Property***: $R(x, x)$ is true for each x in C.
- ***Symmetric Property***: $R(x, y) = R(y, x)$, for each pair x and y in C.
- ***Transitive Property***: If $R(x, y)$ is true and $R(y, z)$ is true, then $R(x, z)$ is true, for every x, y, and z in C.

For example, the usual "equals" operator ($=$) is an equivalence relation for any set of numbers. The ***equivalence class*** for any object x in C is the set of all objects y, such that $R(x, y)$ is true. Note that any equivalence relation R for a set C partitions the set C into disjoint subsets that consist of the equivalence classes of the objects in C.

We can define an interesting ***link relation*** on the edges of a graph G. We say two edges e and f of G are ***linked*** if $e = f$ or G has a simple cycle containing both e and f. The following lemma gives fundamental properties of the link relation.

Lemma 13.24: *Let G be a connected graph. Then,*

1. *The link relation forms an equivalence relation on the edges of G.*
2. *A biconnected component of G is the subgraph induced by an equivalence class of linked edges.*
3. *An edge e of G is a separation edge if and only if e forms a single-element equivalence class of linked edges.*
4. *A vertex v of G is a separation vertex if and only if v has incident edges in at least two distinct equivalence classes of linked edges.*

Proof: It is readily seen that the link relation is reflexive and symmetric. To show that it is transitive, suppose that edges f and g are linked, and edges g and h are linked. If $f = g$ or $g = h$, then $f = h$ or there is a simple cycle containing f and h; hence, f and h are linked. Suppose, then, that f, g, and h are distinct. That is, there is a simple cycle C_{fg} through f and g, and there is a simple cycle C_{gh} through g and h. Consider the graph obtained by the union of cycles C_{fg} and C_{gh}. While this graph may not be a simple cycle itself (although we could have $C_{fg} = C_{gh}$), it contains a simple cycle C_{fh} through f and h. Thus, f and h are linked. Therefore, the link relation is an equivalence relation.

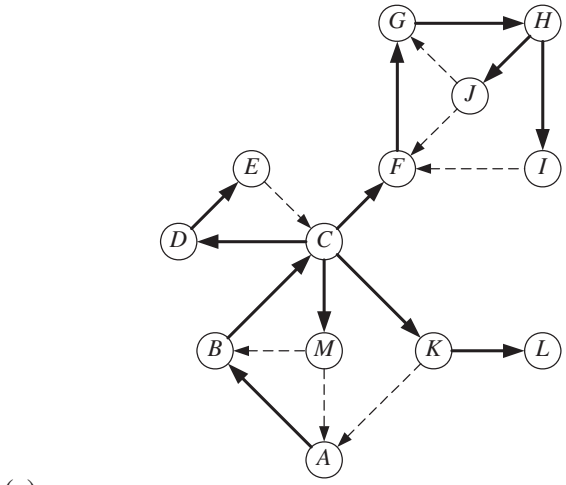
The correspondence between equivalence classes of the link relation and biconnected components of G is a consequence of Lemma 13.23. ■

A Linked Approach to Computing Biconnected Components via DFS

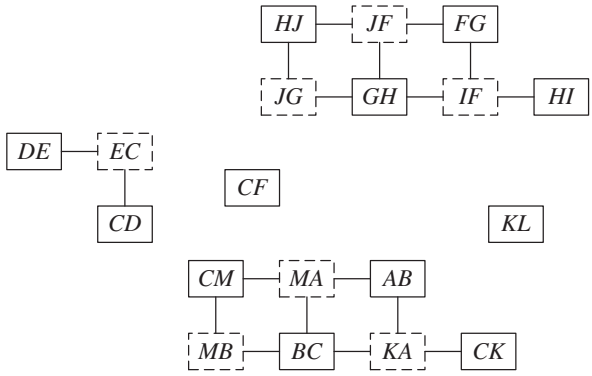
Since the equivalence classes of the link relation on the edges of G are the same as the biconnected components, by Lemma 13.24, to construct the biconnected components of G we need only compute the equivalence classes of the link relation among G's edges. To perform this computation, let us begin with a DFS traversal of G, and construct an ***auxiliary graph*** B as follows (see Figure 13.20):

- The vertices of B are the edges of G.
- For every back edge e of G, let $f_1, \ldots, f_k$ be the discovery edges of G that form a cycle with e. Graph B contains the edges $(e, f_1), \ldots, (e, f_k)$.

Since there are $m - n + 1$ back edges and each cycle induced by a back edge has at most $O(n)$ edges, the graph B has at most $O(nm)$ edges.



(a)



(b)

Figure 13.20: Auxiliary graph used to compute link components: (a) graph G on which a DFS traversal has been performed (the back edges are drawn with dashed lines); (b) auxiliary graph associated with G and its DFS traversal.

An $O(nm)$-Time Algorithm

From Figure 13.20, it appears that each connected component in B corresponds to an equivalence class in the link relation for the graph G. After all, we included an edge (e, f) in B for each back edge e found on the cycle containing f that was induced by e and the DFS spanning tree.

The following lemma, whose proof is left as an exercise (C-13.7), establishes a strong relationship between the graph B and the equivalence classes in the link relation on G's components of G, where, for brevity, we call the equivalence classes in the link relation the ***link components*** of G.

Lemma 13.25: *The connected components of the auxiliary graph B correspond to the link components of the graph G that induced B.*

Lemma 13.25 yields the following $O(nm)$-time algorithm for computing all the link components of a graph G with n vertices and m edges:

1. Perform a DFS traversal T on G.
2. Compute the auxiliary graph B by identifying the cycles of G induced by each back edge with respect to T.
3. Compute the connected components of B, for example, by performing a DFS traversal of the auxiliary graph B.
4. For each connected component of B, output the vertices of B (which are edges of G) as a link component of G.

From the identification of the link components in G, we can then determine the biconnected components, separation vertices, and separation edges of the graph G in linear time. Namely, after the edges of G have been partitioned into equivalence classes with respect to the link relation, the biconnected components, separation vertices, and separation edges of G can be identified in $O(n + m)$ time, using the simple rules listed in Lemma 13.24. Unfortunately, constructing the auxiliary graph B can take as much as $O(nm)$ time; hence, the bottleneck computation in this algorithm is the construction of B.

But note that we don't actually need all of the auxiliary graph B in order to find the biconnected components of G. We only need to identify the connected components in B. Thus, it would actually be sufficient if we were to simply compute a spanning tree for each connected component in B, that is, a spanning forest for B. Since the connected components in a spanning forest for B are the same as in the graph B itself, we don't actually need all the edges of B—just enough of them to construct a spanning forest of B.

Therefore, let us concentrate on how we can apply this more efficient spanning-forest approach to compute the equivalence classes of the edges of G with respect to the link relation.

A Linear-Time Algorithm

As outlined above, we can reduce the time required to compute the link components of G to $O(m)$ time by using an auxiliary graph of smaller size, which is a spanning forest of B. The algorithm is described in Algorithm 13.21.

Algorithm LinkComponents(G):
Input: A connected graph G
Output: The link components of G

```
Let F be an initially empty auxiliary graph.
Perform a DFS traversal of G starting at an arbitrary vertex s.
Add each DFS discovery edge f as a vertex in F and mark f "unlinked."
For each vertex v of G, let p(v) be the parent of v in the DFS spanning tree.
for each vertex v, in increasing rank order as visited in the DFS traversal do
    for each back edge e = (u, v) with destination v do
        Add e as a vertex of the graph F.
        // March up from u to s adding edges to F only as necessary.
        while u ≠ v do
            Let f be the vertex in F corresponding to the discovery edge (u, p(u)).
            Add the edge (e, f) to F.
            if f is marked "unlinked" then
                Mark f as "linked."
                u ← p(u)
            else
                u ← v        // shortcut to the end of the while loop
Compute the connected components of the graph F.
```

Algorithm 13.21: A linear-time algorithm for computing the link components. Note that a connected component in F consisting of an individual "unlinked" vertex corresponds to a separation edge (related only to itself in the link relation).

Let us analyze the running time of LinkComponents, from Algorithm 13.21. The initial DFS traversal of G takes $O(m)$ time. The main computation, however, is the construction of the auxiliary graph F, which takes time proportional to the number of vertices and edges of F. Note that at some point in the execution of the algorithm, each edge of G is added as a vertex of F. We use an accounting charge method to account for the edges of F. Namely, each time we add to F an edge (e, f), from a newly encountered back edge e to a discover edge f, let us charge this operation to f if f is marked "unlinked" and to e otherwise. From the construction of the inner while-loop, we see that we charge each vertex of F at most once during the algorithm using this scheme. We conclude that the construction of F takes $O(m)$ time. Finally, the computation of the connected components of F, which correspond to the link components of G, takes $O(m)$ time.

The correctness of the above algorithm follows from the fact that the graph F in LinkComponents is a spanning forest of the graph B mentioned in Lemma 13.25. For details, see Exercise C-13.8. Therefore, we summarize with the following theorem and give an example of LinkComponents in Figure 13.22.

Theorem 13.26: *Given a connected graph G with m edges, we can compute G's biconnected components, separation vertices, and separation edges in $O(m)$ time.*

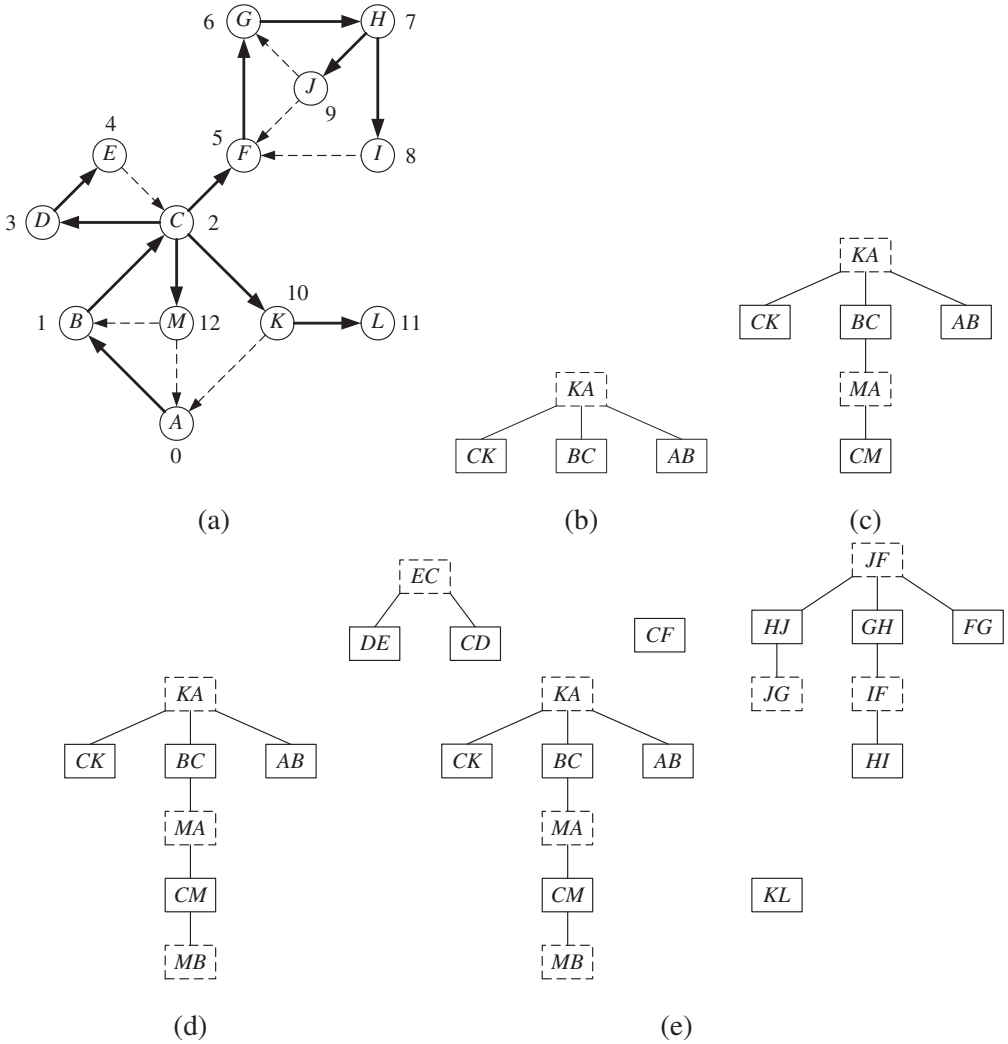


Figure 13.22: Sample execution of algorithm LinkComponents (Algorithm 13.21): (a) input graph G after a DFS traversal (the vertices are labeled by their rank in the visit order, and the back edges are drawn with dashed lines); auxiliary graph F after processing (b) back edge (K, A), (c) back edge (M, A), and (d) back edge $(M.B)$; (e) graph F at the end of the algorithm.

13.6 Exercises

Reinforcement

R-13.1 Draw a simple undirected graph G that has 12 vertices, 18 edges, and 3 connected components. Why would it be impossible to draw G with 3 connected components if G had 66 edges?

R-13.2 Let G be a simple connected graph with n vertices and m edges. Explain why $O(\log m)$ is $O(\log n)$.

R-13.3 Draw a simple connected directed graph with 8 vertices and 16 edges, such that the in-degree and out-degree of each vertex is 2. Show that there is a single cycle (which may not necessarily be simple) that includes all the edges of your graph, that is, you can trace all the edges in their respective directions without ever lifting your pencil. (Such a cycle is called an ***Euler tour***.)

R-13.4 Bob loves foreign languages and wants to plan his course schedule to take the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are:

- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32.

Find a sequence of courses that allows Bob to satisfy all the prerequisites.

R-13.5 Suppose we represent a graph G having n vertices with an adjacency matrix. Why, in this case, would inserting an undirected edge in G run in $O(1)$ time while inserting a new vertex would take $O(n^2)$ time?

R-13.6 Let G be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table below:

vertex	adjacent vertices
1	(2, 3, 4)
2	(1, 3, 4)
3	(1, 2, 4)
4	(1, 2, 3, 6)
5	(6, 7, 8)
6	(4, 5, 7)
7	(5, 6, 8)
8	(5, 7)

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the above table.

a. Draw G.
b. Order the vertices as they are visited in a DFS traversal starting at vertex 1.
c. Order the vertices as they are visited in a BFS traversal starting at vertex 1.

R-13.7 Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

a. The graph has 10,000 vertices and 20,000 edges, and it is important to use as little space as possible.
b. The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.
c. You need to answer the query areAdjacent as fast as possible, no matter how much space you use.

R-13.8 Explain why the DFS traversal runs in $\Theta(n^2)$ time on an n-vertex simple graph that is represented with the adjacency matrix structure.

R-13.9 Draw the transitive closure of the directed graph shown in Figure 13.2.

R-13.10 Compute a topological ordering for the vertices in the directed graph drawn with solid edges in Figure 13.10d.

R-13.11 Can we use a queue instead of a stack as an auxiliary data structure in the topological sorting algorithm shown in Algorithm 13.16?

R-13.12 Give the order in which the edges are labeled by the DFS traversal shown in Figure 13.5.

R-13.13 Give the order in which the edges are labeled by the BFS traversal shown in Figure 13.9.

R-13.14 Give the order in which the edges are labeled by the DFS traversal shown in Figure 13.12.

R-13.15 How many biconnected components would be in the graph shown in Figure 13.5a if we were to remove the edge (B,C) and the edge (N,K)?

Creativity

C-13.1 Justify Theorem 13.11.

C-13.2 Describe the details of an $O(n + m)$-time algorithm for computing all the connected components of an undirected graph G with n vertices and m edges.

C-13.3 Let T be the spanning tree rooted at the start vertex produced by the depth-first search of a connected, undirected graph, G. Argue why every edge of G, not in T, goes from a vertex in T to one of its ancestors, that is, it is a ***back edge***.

Hint: Suppose that such a nontree edge is a cross edge, and argue based upon the order the DFS visits the end vertices of this edge how this leads to a contradiction.

C-13.4 Suppose G is a graph with n vertices and m edges. Describe a way to represent G using $O(n + m)$ space so as to support in $O(\log n)$ time an operation that can test, for any two vertices v and w, whether v and w are adjacent.

C-13.5 Give a proof of Lemma 13.23.

C-13.6 Show that if a graph G has at least three vertices, then it has a separation edge only if it has a separation vertex.

C-13.7 Give a proof of Lemma 13.25.

C-13.8 Supply the details of the proof of correctness of the LinkComponents algorithm (Algorithm 13.21).

C-13.9 Show how to perform a BFS traversal using, as an auxiliary data structure, a single queue instead of the level containers $L_0, L_1, \ldots$.

C-13.10 Show that, if T is a BFS tree produced for a connected graph G, then, for each vertex v at level i, the path of T between s and v has i edges, and any other path of G between s and v has at least i edges.

C-13.11 The directed version of the BFS algorithm classifies nontree edges as being either back edges or cross edges, but it does not distinguish between these two types. Given a BFS spanning tree, T, for a directed graph, $\vec{G}$, and a set of nontree edges, E', describe an algorithm that can correctly label each edge in E' as being either a back edge or cross edge. Your algorithm should run in $O(n+m)$ time, where n is the number of vertices and m is the number of edges.

Hint: Consider first constructing an Euler tour traversal of the tree T.

C-13.12 Explain why there are no forward nontree edges with respect to a BFS tree constructed for a directed graph.

C-13.13 In the pseudocode description of the directed DFS traversal algorithm we did not distinguish the labeling of cross edges and forward edges. Describe how to modify the directed DFS algorithm so that it correctly labels each nontree edge as a back edge, forward edge, or cross edge.

C-13.14 Explain why the strong connectivity testing algorithm given in Section 13.4.1 is correct.

C-13.15 Let G be an undirected graph with n vertices and m edges. Describe an $O(n+m)$-time algorithm to determine whether G contains at least two cycles.

C-13.16 Let G be an undirected graph with n vertices and m edges. Describe an algorithm running in $O(n+m)$ time that can determine whether G contains exactly two cycles that have no edges in common.

C-13.17 Justify Theorem 13.14.

C-13.18 Show that it is possible to count the total number of paths in a directed acyclic graph, $\vec{G}$, with n vertices and m edges using $O(n+m)$ additions. Also, show that there is a graph, $\vec{G}$, where this number is at least $2^{n/2}$.

Applications

A-13.1 A ***road network*** is a mixed graph defined by the roads in a geographic region. Vertices in this graph are defined by road intersections and dead ends, and edges are defined by the portions of roads that connect such vertices. An edge is directed if its associated road is a one-way street; otherwise, an edge is undirected. Imagine that you are the manager of a post office delivery system, and that you need to map out the route that a postal worker should drive to deliver mail to all the houses on the roads in a given geographic region. Furthermore, suppose that

all the streets in this region are one-way, so the road network for this region is a directed graph, $\vec{G}$. An ***Euler tour*** of such a directed graph, $\vec{G}$, is a cycle that traverses each edge of $\vec{G}$ exactly once according to its direction. Note that an Euler tour is necessarily the shortest route that starts at the post office (assuming it is one of the vertices in $\vec{G}$) and visits every street in the road network represented by $\vec{G}$ in the appropriate direction (where u-turns are allowed at a vertex with an outgoing edge going back to the origin of an incoming edge). Such a tour always exists if $\vec{G}$ is connected and the in-degree equals the out-degree of each vertex in $\vec{G}$. Describe an $O(n + m)$-time algorithm for finding an Euler tour of such a digraph, $\vec{G}$, if such a tour exists, starting from some vertex, v, where n is the number of vertices in $\vec{G}$ and m is the number of edges in $\vec{G}$.

A-13.2 Suppose you are given a connected road network, G, as described in the previous exercise, except that none of the edges in G are directed. Describe an efficient method for designing a tour of G that starts at some vertex, v, and traverses each edge of G exactly once in each direction (with u-turns allowed). What is the running time of your algorithm?

A-13.3 Suppose you work for a company that is giving a smartphone to each of its employees. Unfortunately, the companies that make apps for these smartphones are constantly suing each other over their respective intellectual property. Say that two apps, A and B, are ***litigation-conflicting*** if A contains some disputed technology that is also contained in B. Your job is to pre-install a set of apps on the company smartphones, but you have been asked by the company lawyers to avoid installing any litigation-conflicting apps. To make your job a little easier, these lawyers have given you a graph, G, whose vertices consist of all the possible apps you might want to install and whose edges consist of all the pairs of litigation-conflicting apps. An independent set of such an undirected graph, $G = (V, E)$, is a subset, I, of V, such that no two vertices in I are adjacent. That is, if $u, v \in I$, then $(u, v) \notin E$. A ***maximal independent set*** M is an independent set such that, if we were to add any additional vertex to M, then it would not be independent any longer. In the case of the graph, G, of litigation-conflicting apps, a maximal independent set in G corresponds to a set of nonconflicting apps such that if we were to add any other app to this set, it would conflict with at least one of the apps in the set. Give an efficient algorithm that computes a maximal independent set for a such a graph, G. What is the running time of your algorithm?

A-13.4 Tamarindo University and many other schools worldwide are doing a joint project on multimedia. A computer network is built to connect these schools using communication links that form a free tree. The schools decide to install a file server at one of the schools to share data among all the schools. Since the transmission time on a link is dominated by the link setup and synchronization, the cost of a data transfer is proportional to the number of links used. Hence, it is desirable to choose a "central" location for the file server. Given a free tree T and a node v of T, the *eccentricity* of v is the length of a longest path from v to any other node of T. A node of T with minimum eccentricity is called a *center* of T.

a. Design an efficient algorithm that, given an n-node free tree T, computes a center of T.
b. Is the center unique? If not, how many distinct centers can a free tree have?

A-13.5 The time delay of a long-distance call can be determined by multiplying a small fixed constant by the number of communication links on the telephone network between the caller and callee. Suppose the telephone network of a company named RT&T is a free tree. The engineers of RT&T want to compute the maximum possible time delay that may be experienced in a long-distance call. Given a free tree T, the *diameter* of T is the length of a longest path between two nodes of T. Give an efficient algorithm for computing the diameter of T.

A-13.6 A company named RT&T has a network of n stations connected by m high-speed communication links. Each customer's phone is connected to one station in his or her area. The engineers of RT&T have developed a prototype video-phone system that allows two customers to see each other during a phone call. In order to have acceptable image quality, however, the number of links used to transmit video signals between the two parties cannot exceed 4. Suppose that RT&T's network is represented by a graph. Design an efficient algorithm that computes, for each station, the set of stations it can reach using no more than 4 links.

A-13.7 Imagine that you are a medical practitioner for a developing country, Strategia, and it is your job to inoculate people in each village in Strategia so as to limit the ability of the Kissoba virus to spread in Strategia. The Kissoba virus can only be spread between two people if they kiss. For each village, you are given a ***kissing*** graph, G, whose vertices are the people in that village and whose edges are pairs of people who regularly kiss. Unfortunately, you don't have an unlimited supply of the Kissoba vaccine, and each shot is expensive. So the president of Strategia has asked that you limit the people you vaccinate to those who are ***central*** kissers, where a central kisser is a person, p, such that there are no two people, r and q, who are kissed by p such that there is a sequence of kissing pairs of people that starts with r and leads to q while avoiding p. Given a graph, G, representing the kissing graph for a village in Strategia, describe an efficient algorithm for identifying all the central kissers in G, and analyze its running time.

Chapter Notes

DFS is a part of the "folklore" of computer science, but Hopcroft and Tarjan [102, 205] showed how useful this algorithm is for solving several different graph problems. Knuth [129] discusses the topological sorting problem. The simple linear-time algorithm in Section 13.4.1 for determining if a directed graph is strongly connected is due to Kosaraju. The Floyd-Warshall algorithm appears in a paper by Floyd [72] and is based upon a theorem of Warshall [214]. The mark-sweep garbage collection method we describe is one of many different algorithms for performing garbage collection. We encourage the reader interested in further study of garbage collection to examine the book by Jones [114]. To learn about algorithms for drawing graphs, see the book by Di Battista *et al.* [55] and the handbook edited by Tamassia [203]. The reader interested in further study of graph algorithms is referred to the books by Ahuja, Magnanti, and Orlin [10], Cormen, Leiserson, and Rivest [50], Even [68], Gibbons [81], Mehlhorn [158], and Tarjan [207], and the book chapter by van Leeuwen [210]. For more applications of graph algorithms to social networks, see the book by Easley and Kleinberg [60].