CHAPTER 9

Graph Algorithms

In this chapter, we discuss several common problems in graph theory. Not only are these algorithms useful in practice, they are also interesting because in many real-life applications they are too slow unless careful attention is paid to the choice of data structures. We will...

- Show several real-life problems, which can be converted to problems on graphs.
- Give algorithms to solve several common graph problems.
- Show how the proper choice of data structures can drastically reduce the running time of these algorithms.
- See an important technique, known as depth-first search, and show how it can be used to solve several seemingly nontrivial problems in linear time.

9.1 Definitions

A **graph** $G = (V, E)$ consists of a set of **vertices,** V, and a set of **edges,** E. Each edge is a pair (v, w), where $v, w \in V$. Edges are sometimes referred to as **arcs.** If the pair is ordered, then the graph is **directed.** Directed graphs are sometimes referred to as **digraphs.** Vertex w is **adjacent** to v if and only if $(v, w) \in E$. In an undirected graph with edge (v, w), and hence (w, v), w is adjacent to v and v is adjacent to w. Sometimes an edge has a third component, known as either a **weight** or a **cost.**

A **path** in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < N$. The **length** of such a path is the number of edges on the path, which is equal to $N - 1$. We allow a path from a vertex to itself; if this path contains no edges, then the path length is 0. This is a convenient way to define an otherwise special case. If the graph contains an edge (v, v) from a vertex to itself, then the path v, v is sometimes referred to as a **loop.** The graphs we will consider will generally be loopless. A **simple path** is a path such that all vertices are distinct, except that the first and last could be the same.

A **cycle** in a directed graph is a path of length at least 1 such that $w_1 = w_N$; this cycle is simple if the path is simple. For undirected graphs, we require that the edges be distinct. The logic of these requirements is that the path u, v, u in an undirected graph should not be considered a cycle, because (u, v) and (v, u) are the same edge. In a directed graph, these are different edges, so it makes sense to call this a cycle. A directed graph is **acyclic** if it has no cycles. A directed acyclic graph is sometimes referred to by its abbreviation, **DAG.**

An undirected graph is **connected** if there is a path from every vertex to every other vertex. A directed graph with this property is called **strongly connected.** If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be **weakly connected.** A **complete graph** is a graph in which there is an edge between every pair of vertices.

An example of a real-life situation that can be modeled by a graph is the airport system. Each airport is a vertex, and two vertices are connected by an edge if there is a nonstop flight from the airports that are represented by the vertices. The edge could have a weight, representing the time, distance, or cost of the flight. It is reasonable to assume that such a graph is directed, since it might take longer or cost more (depending on local taxes, for example) to fly in different directions. We would probably like to make sure that the airport system is strongly connected, so that it is always possible to fly from any airport to any other airport. We might also like to quickly determine the best flight between any two airports. "Best" could mean the path with the fewest number of edges or could be taken with respect to one, or all, of the weight measures.

Traffic flow can be modeled by a graph. Each street intersection represents a vertex, and each street is an edge. The edge costs could represent, among other things, a speed limit or a capacity (number of lanes). We could then ask for the shortest route or use this information to find the most likely location for bottlenecks.

In the remainder of this chapter, we will see several more applications of graphs. Many of these graphs can be quite large, so it is important that the algorithms we use be efficient.

9.1.1 Representation of Graphs

We will consider directed graphs (undirected graphs are similarly represented).

Suppose, for now, that we can number the vertices, starting at 1. The graph shown in Figure 9.1 represents 7 vertices and 12 edges.

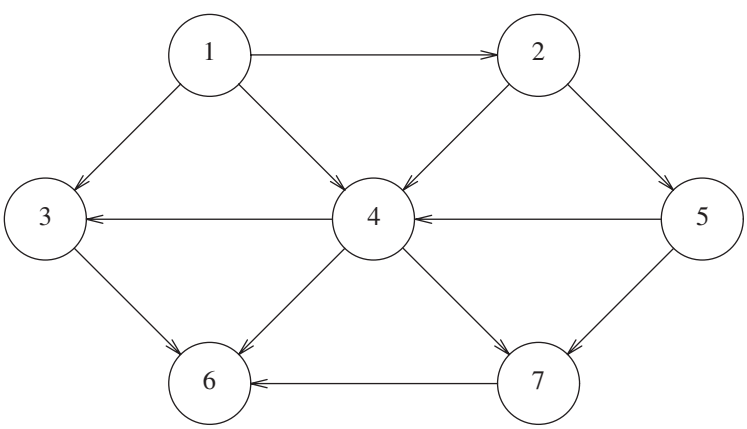

Figure 9.1 A directed graph

One simple way to represent a graph is to use a two-dimensional array. This is known as an **adjacency matrix** representation. For each edge (u, v), we set $A[u][v]$ to `true`; otherwise the entry in the array is `false`. If the edge has a weight associated with it, then we can set $A[u][v]$ equal to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges. For instance, if we were looking for the cheapest airplane route, we could represent nonexistent flights with a cost of ∞. If we were looking, for some strange reason, for the most expensive airplane route, we could use $-\infty$ (or perhaps 0) to represent nonexistent edges.

Although this has the merit of extreme simplicity, the space requirement is $\Theta(|V|^2)$, which can be prohibitive if the graph does not have very many edges. An adjacency matrix is an appropriate representation if the graph is **dense:** $|E| = \Theta(|V|^2)$. In most of the applications that we shall see, this is not true. For instance, suppose the graph represents a street map. Assume a Manhattan-like orientation, where almost all the streets run either north–south or east–west. Therefore, any intersection is attached to roughly four streets, so if the graph is directed and all streets are two-way, then $|E| \approx 4|V|$. If there are 3,000 intersections, then we have a 3,000-vertex graph with 12,000 edge entries, which would require an array of size 9,000,000. Most of these entries would contain zero. This is intuitively bad, because we want our data structures to represent the data that are actually there and not the data that are not present.

If the graph is not dense, in other words, if the graph is **sparse,** a better solution is an **adjacency list** representation. For each vertex, we keep a list of all adjacent vertices. The space requirement is then $O(|E| + |V|)$, which is linear in the size of the graph.[1] The abstract representation should be clear from Figure 9.2. If the edges have weights, then this additional information is also stored in the adjacency lists.

Adjacency lists are the standard way to represent graphs. Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage essentially doubles. A common requirement in graph algorithms is to find all vertices adjacent to some given vertex v, and this can be done, in time proportional to the number of such vertices found, by a simple scan down the appropriate adjacency list.

There are several alternatives for maintaining the adjacency lists. First, observe that the lists themselves can be maintained in either `vector`s or `list`s. However, for sparse graphs, when using `vector`s, the programmer may need to initialize each `vector` with a smaller capacity than the default; otherwise, there could be significant wasted space.

Because it is important to be able to quickly obtain the list of adjacent vertices for any vertex, the two basic options are to use a map in which the keys are vertices and the values are adjacency lists, or to maintain each adjacency list as a data member of a `Vertex` class. The first option is arguably simpler, but the second option can be faster, because it avoids repeated lookups in the map.

In the second scenario, if the vertex is a `string` (for instance, an airport name, or the name of a street intersection), then a map can be used in which the key is the vertex name and the value is a `Vertex` (typically a pointer to a `Vertex`), and each `Vertex` object keeps a list of (pointers to the) adjacent vertices and perhaps also the original `string` name.

[1] When we speak of linear-time graph algorithms, $O(|E| + |V|)$ is the running time we require.

1	2, 4, 3
2	4, 5
3	6
4	6, 7, 3
5	4, 7
6	(empty)
7	6

Figure 9.2 An adjacency list representation of a graph

In most of the chapter, we present the graph algorithms using pseudocode. We will do this to save space and, of course, to make the presentation of the algorithms much clearer. At the end of Section 9.3, we provide a working C++ implementation of a routine that makes underlying use of a shortest-path algorithm to obtain its answers.

9.2 Topological Sort

A **topological sort** is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears *after* v_i in the ordering. The graph in Figure 9.3 represents the course prerequisite structure at a state university in Miami. A directed edge (v, w) indicates that course v must be completed before course w may be attempted. A topological ordering of these courses is any course sequence that does not violate the prerequisite requirement.

It is clear that a topological ordering is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v. Furthermore, the ordering is not necessarily unique; any legal ordering will do. In the graph in Figure 9.4, $v_1, v_2, v_5, v_4, v_3, v_7, v_6$ and $v_1, v_2, v_5, v_4, v_7, v_3, v_6$ are both topological orderings.

A simple algorithm to find a topological ordering is first to find any vertex with no incoming edges. We can then print this vertex, and remove it, along with its edges, from the graph. Then we apply this same strategy to the rest of the graph.

To formalize this, we define the **indegree** of a vertex v as the number of edges (u, v). We compute the indegrees of all vertices in the graph. Assuming that the indegree for each

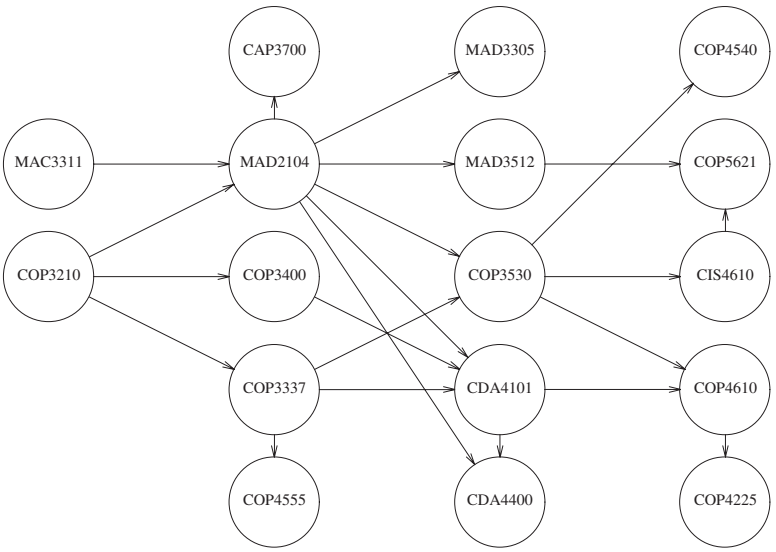

Figure 9.3 An acyclic graph representing course prerequisite structure

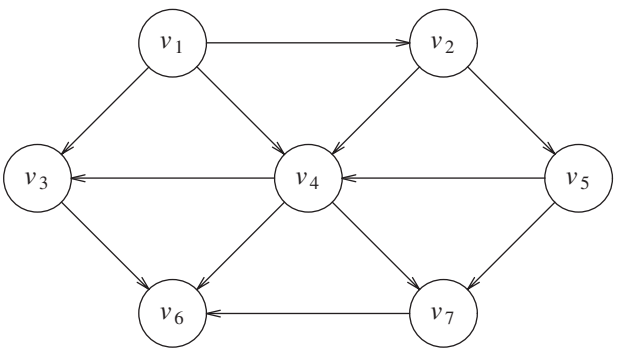

Figure 9.4 An acyclic graph

vertex is stored, and that the graph is read into an adjacency list, we can then apply the algorithm in Figure 9.5 to generate a topological ordering.

The function `findNewVertexOfIndegreeZero` scans the array of vertices looking for a vertex with indegree 0 that has not already been assigned a topological number. It returns `NOT_A_VERTEX` if no such vertex exists; this indicates that the graph has a cycle.

Because `findNewVertexOfIndegreeZero` is a simple sequential scan of the array of vertices, each call to it takes $O(|V|)$ time. Since there are $|V|$ such calls, the running time of the algorithm is $O(|V|^2)$.

By paying more careful attention to the data structures, it is possible to do better. The cause of the poor running time is the sequential scan through the array of vertices. If the

```
void Graph::topsort( )
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == NOT_A_VERTEX )
            throw CycleFoundException{ };
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

Figure 9.5 Simple topological sort pseudocode

graph is sparse, we would expect that only a few vertices have their indegrees updated during each iteration. However, in the search for a vertex of indegree 0, we look at (potentially) all the vertices, even though only a few have changed.

We can remove this inefficiency by keeping all the (unassigned) vertices of indegree 0 in a special *box*. The `findNewVertexOfIndegreeZero` function then returns (and removes) any vertex in the box. When we decrement the indegrees of the adjacent vertices, we check each vertex and place it in the box if its indegree falls to 0.

To implement the box, we can use either a stack or a queue; we will use a queue. First, the indegree is computed for every vertex. Then all vertices of indegree 0 are placed on an initially empty queue. While the queue is not empty, a vertex v is removed, and all vertices adjacent to v have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0. The topological ordering then is the order in which the vertices dequeue. Figure 9.6 shows the status after each phase.

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

Figure 9.6 Result of applying topological sort to the graph in Figure 9.4

```
void Graph::topsort( )
{
    Queue<Vertex> q;
    int counter = 0;

    q.makeEmpty( );
    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter;  // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw CycleFoundException{ };
}
```

Figure 9.7 Pseudocode to perform topological sort

A pseudocode implementation of this algorithm is given in Figure 9.7. As before, we will assume that the graph is already read into an adjacency list and that the indegrees are computed and stored with the vertices. We also assume each vertex has a named data member, `topNum`, in which to place its topological numbering.

The time to perform this algorithm is $O(|E| + |V|)$ if adjacency lists are used. This is apparent when one realizes that the body of the `for` loop is executed at most once per edge. Computing the indegrees can be done with the following code; this same logic shows that the cost of this computation is $O(|E| + |V|)$, even though there are nested loops.

```
for each Vertex v
    v.indegree = 0;

for each Vertex v
    for each Vertex w adjacent to v
        w.indegree++;
```

The queue operations are done at most once per vertex, and the other initialization steps, including the computation of indegrees, also take time proportional to the size of the graph.

9.3 Shortest-Path Algorithms

In this section we examine various shortest-path problems. The input is a weighted graph: Associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the edge. The cost of a path $v_1 v_2 \ldots v_N$ is $\sum_{i=1}^{N-1} c_{i,i+1}$. This is referred to as the **weighted path length.** The **unweighted path length** is merely the number of edges on the path, namely, $N - 1$.

Single-Source Shortest-Path Problem

Given as input a weighted graph, $G = (V, E)$, and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

For example, in the graph in Figure 9.8, the shortest weighted path from v_1 to v_6 has a cost of 6 and goes from v_1 to v_4 to v_7 to v_6. The shortest unweighted path between these vertices is 2. Generally, when it is not specified whether we are referring to a weighted or an unweighted path, the path is weighted if the graph is. Notice also that in this graph there is no path from v_6 to v_1.

The graph in the preceding example has no edges of negative cost. The graph in Figure 9.9 shows the problems that negative edges can cause. The path from v_5 to v_4 has

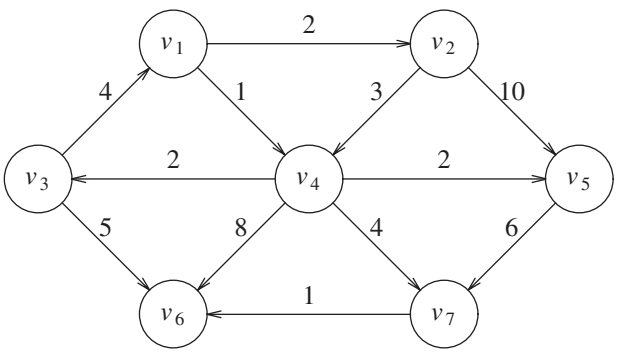

Figure 9.8 A directed graph G

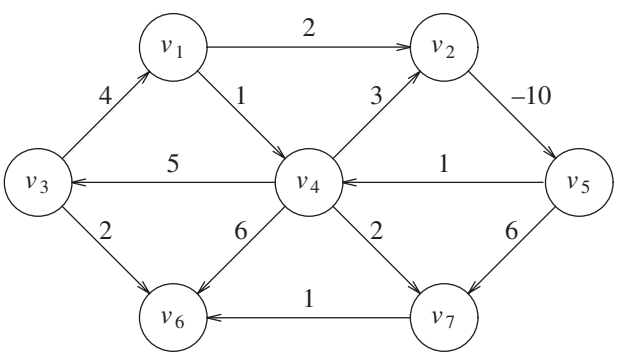

Figure 9.9 A graph with a negative-cost cycle

cost 1, but a shorter path exists by following the loop v_5, v_4, v_2, v_5, v_4, which has cost -5. This path is still not the shortest, because we could stay in the loop arbitrarily long. Thus, the shortest path between these two points is undefined. Similarly, the shortest path from v_1 to v_6 is undefined, because we can get into the same loop. This loop is known as a **negative-cost cycle;** when one is present in the graph, the shortest paths are not defined. Negative-cost edges are not necessarily bad, as the cycles are, but their presence seems to make the problem harder. For convenience, in the absence of a negative-cost cycle, the shortest path from s to s is zero.

There are many examples where we might want to solve the shortest-path problem. If the vertices represent computers; the edges represent a link between computers; and the costs represent communication costs (phone bill per a megabyte of data), delay costs (number of seconds required to transmit a megabyte), or a combination of these and other factors, then we can use the shortest-path algorithm to find the cheapest way to send electronic news from one computer to a set of other computers.

We can model airplane or other mass transit routes by graphs and use a shortest-path algorithm to compute the best route between two points. In this and many practical applications, we might want to find the shortest path from one vertex, s, to only one other vertex, t. Currently there are no algorithms in which finding the path from s to one vertex is any faster (by more than a constant factor) than finding the path from s to all vertices.

We will examine algorithms to solve four versions of this problem. First, we will consider the unweighted shortest-path problem and show how to solve it in $O(|E|+|V|)$. Next, we will show how to solve the weighted shortest-path problem if we assume that there are no negative edges. The running time for this algorithm is $O(|E| \log |V|)$ when implemented with reasonable data structures.

If the graph has negative edges, we will provide a simple solution, which unfortunately has a poor time bound of $O(|E| \cdot |V|)$. Finally, we will solve the weighted problem for the special case of acyclic graphs in linear time.

9.3.1 Unweighted Shortest Paths

Figure 9.10 shows an unweighted graph, G. Using some vertex, s, which is an input parameter, we would like to find the shortest path from s to all other vertices. We are only interested in the number of edges contained on the path, so there are no weights on the edges. This is clearly a special case of the weighted shortest-path problem, since we could assign all edges a weight of 1.

For now, suppose we are interested only in the length of the shortest paths, not in the actual paths themselves. Keeping track of the actual paths will turn out to be a matter of simple bookkeeping.

Suppose we choose s to be v_3. Immediately, we can tell that the shortest path from s to v_3 is then a path of length 0. We can mark this information, obtaining the graph in Figure 9.11.

Now we can start looking for all vertices that are a distance 1 away from s. These can be found by looking at the vertices that are adjacent to s. If we do this, we see that v_1 and v_6 are one edge from s. This is shown in Figure 9.12.

We can now find vertices whose shortest path from s is exactly 2, by finding all the vertices adjacent to v_1 and v_6 (the vertices at distance 1), whose shortest paths are not

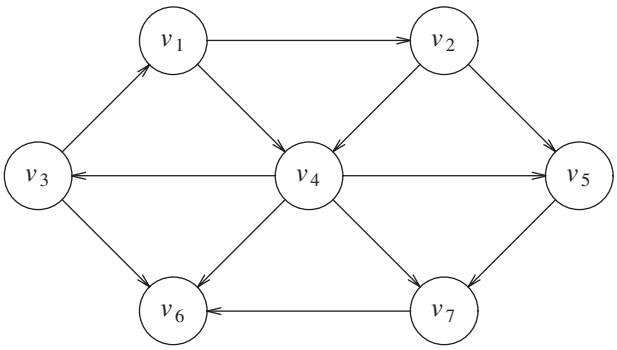

Figure 9.10 An unweighted directed graph G

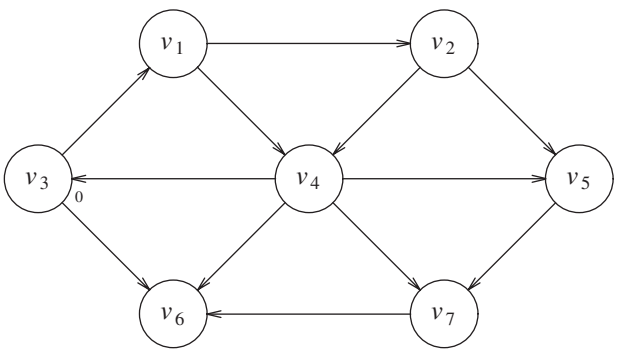

Figure 9.11 Graph after marking the start node as reachable in zero edges

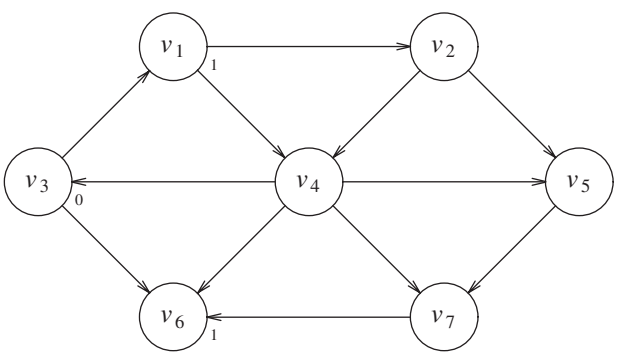

Figure 9.12 Graph after finding all vertices whose path length from s is 1

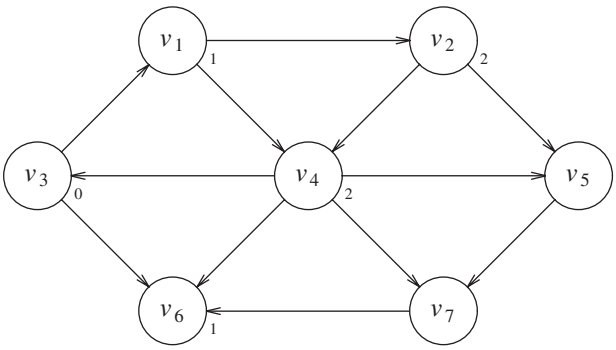

Figure 9.13 Graph after finding all vertices whose shortest path is 2

already known. This search tells us that the shortest path to v_2 and v_4 is 2. Figure 9.13 shows the progress that has been made so far.

Finally we can find, by examining vertices adjacent to the recently evaluated v_2 and v_4, that v_5 and v_7 have a shortest path of three edges. All vertices have now been calculated, and so Figure 9.14 shows the final result of the algorithm.

This strategy for searching a graph is known as **breadth-first search**. It operates by processing vertices in layers: The vertices closest to the start are evaluated first, and the most distant vertices are evaluated last. This is much the same as a level-order traversal for trees.

Given this strategy, we must translate it into code. Figure 9.15 shows the initial configuration of the table that our algorithm will use to keep track of its progress.

For each vertex, we will keep track of three pieces of information. First, we will keep its distance from s in the entry d_v. Initially all vertices are unreachable except for s, whose path length is 0. The entry in p_v is the bookkeeping variable, which will allow us to print the actual paths. The entry *known* is set to `true` after a vertex is processed. Initially, all entries are not *known,* including the start vertex. When a vertex is marked *known,* we have

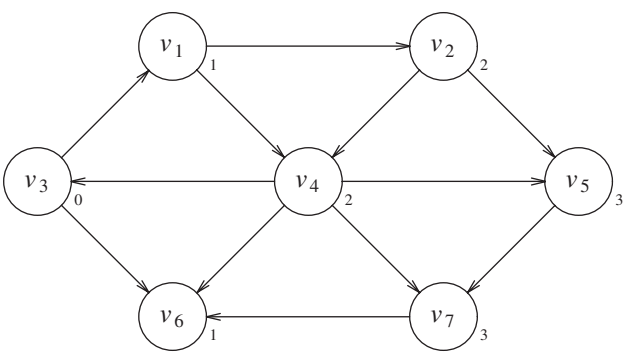

Figure 9.14 Final shortest paths

v	*known*	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.15 Initial configuration of table used in unweighted shortest-path computation

a guarantee that no cheaper path will ever be found, and so processing for that vertex is essentially complete.

The basic algorithm can be described in Figure 9.16. The algorithm in Figure 9.16 mimics the diagrams by declaring as *known* the vertices at distance $d = 0$, then $d = 1$, then $d = 2$, and so on, and setting all the adjacent vertices w that still have $d_w = \infty$ to a distance $d_w = d + 1$.

```
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
            {
                v.known = true;
                for each Vertex w adjacent to v
                    if( w.dist == INFINITY )
                    {
                        w.dist = currDist + 1;
                        w.path = v;
                    }
            }
}
```

Figure 9.16 Pseudocode for unweighted shortest-path algorithm

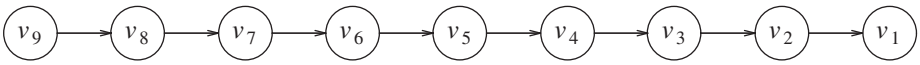

Figure 9.17 A bad case for unweighted shortest-path algorithm using Figure 9.16

By tracing back through the p_v variable, the actual path can be printed. We will see how when we discuss the weighted case.

The running time of the algorithm is $O(|V|^2)$, because of the doubly nested `for` loops. An obvious inefficiency is that the outside loop continues until `NUM_VERTICES-1`, even if all the vertices become *known* much earlier. Although an extra test could be made to avoid this, it does not affect the worst-case running time, as can be seen by generalizing what happens when the input is the graph in Figure 9.17 with start vertex v_9.

We can remove the inefficiency in much the same way as was done for topological sort. At any point in time, there are only two types of *unknown* vertices that have $d_v \neq \infty$. Some have $d_v =$ `currDist`, and the rest have $d_v =$ `currDist + 1`. Because of this extra structure, it is very wasteful to search through the entire table to find a proper vertex.

A very simple but abstract solution is to keep two boxes. Box #1 will have the unknown vertices with $d_v =$ `currDist`, and box #2 will have $d_v =$ `currDist + 1`. The test to find an appropriate vertex v can be replaced by finding any vertex in box #1. After updating w (inside the innermost `if` block), we can add w to box #2. After the outermost `for` loop terminates, box #1 is empty, and box #2 can be transferred to box #1 for the next pass of the `for` loop.

We can refine this idea even further by using just one queue. At the start of the pass, the queue contains only vertices of distance `currDist`. When we add adjacent vertices of distance `currDist + 1`, since they enqueue at the rear, we are guaranteed that they will not be processed until after all the vertices of distance `currDist` have been processed. After the last vertex at distance `currDist` dequeues and is processed, the queue only contains vertices of distance `currDist + 1`, so this process perpetuates. We merely need to begin the process by placing the start node on the queue by itself.

The refined algorithm is shown in Figure 9.18. In the pseudocode, we have assumed that the start vertex, s, is passed as a parameter. Also, it is possible that the queue might empty prematurely, if some vertices are unreachable from the start node. In this case, a distance of `INFINITY` will be reported for these nodes, which is perfectly reasonable. Finally, the `known` data member is not used; once a vertex is processed it can never enter the queue again, so the fact that it need not be reprocessed is implicitly marked. Thus, the `known` data member can be discarded. Figure 9.19 shows how the values on the graph we have been using are changed during the algorithm (it includes the changes that would occur to `known` if we had kept it).

Using the same analysis as was performed for topological sort, we see that the running time is $O(|E| + |V|)$, as long as adjacency lists are used.

9.3.2 Dijkstra's Algorithm

If the graph is weighted, the problem (apparently) becomes harder, but we can still use the ideas from the unweighted case.

```
void Graph::unweighted( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

Figure 9.18 Psuedocode for unweighted shortest-path algorithm

We keep all of the same information as before. Thus, each vertex is marked as either *known* or *unknown*. A tentative distance d_v is kept for each vertex, as before. This distance turns out to be the shortest path length from s to v using only *known* vertices as intermediates. As before, we record p_v, which is the last vertex to cause a change to d_v.

The general method to solve the single-source shortest-path problem is known as **Dijkstra's algorithm.** This thirty-year-old solution is a prime example of a **greedy algorithm.** Greedy algorithms generally solve a problem in stages by doing what appears to be the best thing at each stage. For example, to make change in U.S. currency, most people count out the quarters first, then the dimes, nickels, and pennies. This greedy algorithm gives change using the minimum number of coins. The main problem with greedy algorithms is that they do not always work. The addition of a 12-cent piece breaks the coin-changing algorithm for returning 15 cents, because the answer it gives (one 12-cent piece and three pennies) is not optimal (one dime and one nickel).

Dijkstra's algorithm proceeds in stages, just like the unweighted shortest-path algorithm. At each stage, Dijkstra's algorithm selects a vertex, v, which has the smallest d_v among all the *unknown* vertices and declares that the shortest path from s to v is *known*. The remainder of a stage consists of updating the values of d_w.

In the unweighted case, we set $d_w = d_v + 1$ if $d_w = \infty$. Thus, we essentially lowered the value of d_w if vertex v offered a shorter path. If we apply the same logic to the weighted

	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Figure 9.19 How the data change during the unweighted shortest-path algorithm

case, then we should set $d_w = d_v + c_{v,w}$ if this new value for d_w would be an improvement. Put simply, the algorithm decides whether or not it is a good idea to use v on the path to w. The original cost, d_w, is the cost without using v; the cost calculated above is the cheapest path using v (and only *known* vertices).

The graph in Figure 9.20 is our example. Figure 9.21 represents the initial configuration, assuming that the start node, s, is v_1. The first vertex selected is v_1, with path length 0. This vertex is marked *known*. Now that v_1 is *known,* some entries need to be adjusted. The vertices adjacent to v_1 are v_2 and v_4. Both these vertices get their entries adjusted, as indicated in Figure 9.22.

Next, v_4 is selected and marked *known*. Vertices v_3, v_5, v_6, and v_7 are adjacent, and it turns out that all require adjusting, as shown in Figure 9.23.

Next, v_2 is selected. v_4 is adjacent but already known, so no work is performed on it. v_5 is adjacent but not adjusted, because the cost of going through v_2 is $2 + 10 = 12$ and a path of length 3 is already known. Figure 9.24 shows the table after these vertices are selected.

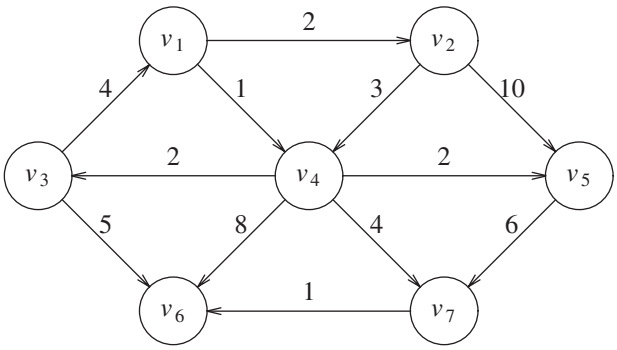

Figure 9.20 The directed graph G (again)

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.21 Initial configuration of table used in Dijkstra's algorithm

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.22 After v_1 is declared *known*

The next vertex selected is v_5 at cost 3. v_7 is the only adjacent vertex, but it is not adjusted, because $3 + 6 > 5$. Then v_3 is selected, and the distance for v_6 is adjusted down to $3 + 5 = 8$. The resulting table is depicted in Figure 9.25.

Next, v_7 is selected; v_6 gets updated down to $5 + 1 = 6$. The resulting table is Figure 9.26.

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.23 After v_4 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

Figure 9.24 After v_2 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

Figure 9.25 After v_5 and then v_3 are declared *known*

Finally, v_6 is selected. The final table is shown in Figure 9.27. Figure 9.28 graphically shows how edges are marked *known* and vertices updated during Dijkstra's algorithm.

To print out the actual path from a start vertex to some vertex v, we can write a recursive routine to follow the trail left in the p variables.

We now give pseudocode to implement Dijkstra's algorithm. Each `Vertex` stores various data members that are used in the algorithm. This is shown in Figure 9.29.

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

Figure 9.26 After v_7 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Figure 9.27 After v_6 is declared *known* and algorithm terminates

The path can be printed out using the recursive routine in Figure 9.30. The routine recursively prints the path all the way up to the vertex before v on the path, and then just prints v. This works because the path is simple.

Figure 9.31 shows the main algorithm, which is just a `for` loop to fill up the table using the greedy selection rule.

A proof by contradiction will show that this algorithm always works as long as no edge has a negative cost. If any edge has negative cost, the algorithm could produce the wrong answer (see Exercise 9.7(a)). The running time depends on how the vertices are manipulated, which we have yet to consider. If we use the obvious algorithm of sequentially scanning the vertices to find the minimum d_v, each phase will take $O(|V|)$ time to find the minimum, and thus $O(|V|^2)$ time will be spent finding the minimum over the course of the algorithm. The time for updating d_w is constant per update, and there is at most one update per edge for a total of $O(|E|)$. Thus, the total running time is $O(|E| + |V|^2) = O(|V|^2)$. If the graph is dense, with $|E| = \Theta(|V|^2)$, this algorithm is not only simple but also essentially optimal, since it runs in time linear in the number of edges.

If the graph is sparse, with $|E| = \Theta(|V|)$, this algorithm is too slow. In this case, the distances would need to be kept in a priority queue. There are actually two ways to do this; both are similar.

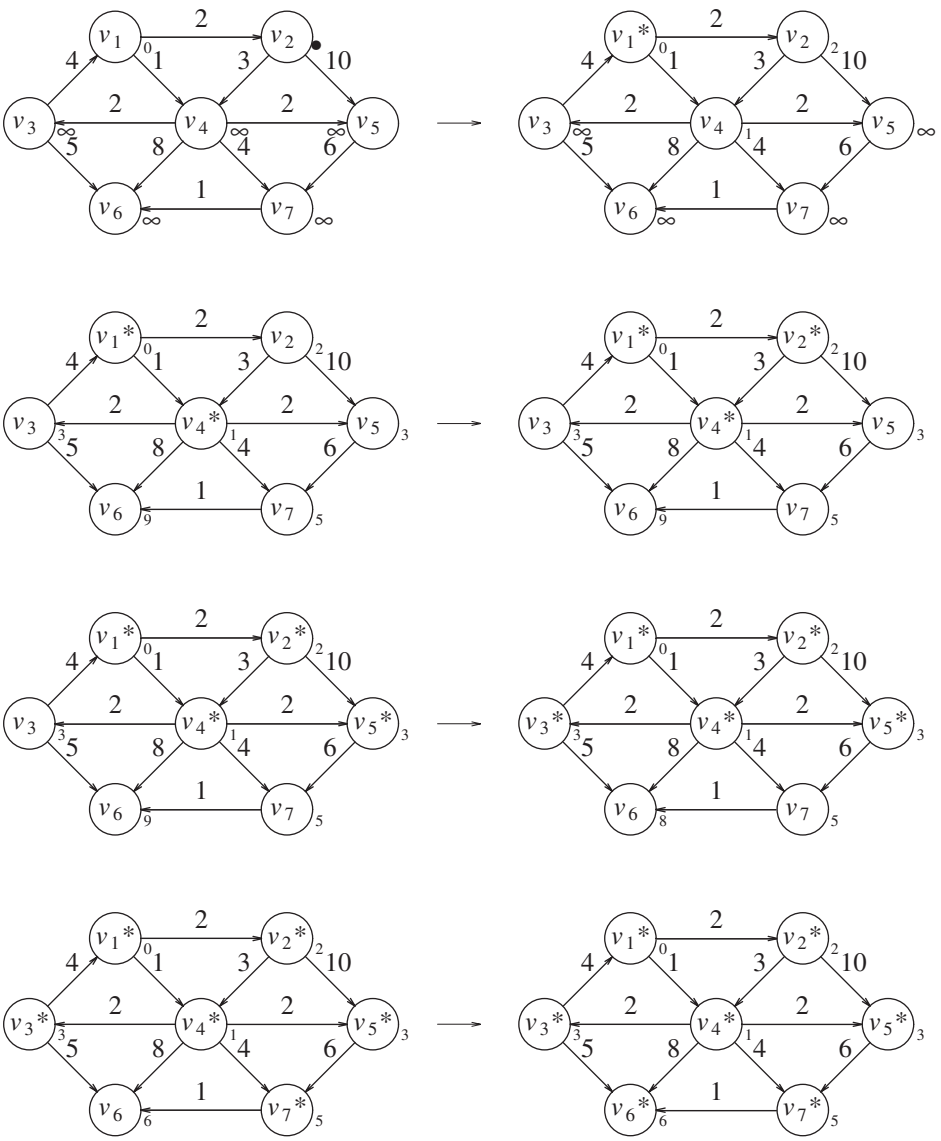

Figure 9.28 Stages of Dijkstra's algorithm

Selection of the vertex v is a `deleteMin` operation, since once the unknown minimum vertex is found, it is no longer unknown and must be removed from future consideration. The update of w's distance can be implemented two ways.

One way treats the update as a `decreaseKey` operation. The time to find the minimum is then $O(\log |V|)$, as is the time to perform updates, which amount to `decreaseKey` operations. This gives a running time of $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$, an improvement

```
/**
 * PSEUDOCODE sketch of the Vertex structure.
 * In real C++, path would be of type Vertex *,
 * and many of the code fragments that we describe
 * require either a dereferencing * or use the
 * -> operator instead of the . operator.
 * Needless to say, this obscures the basic algorithmic ideas.
 */
struct Vertex
{
    List      adj;     // Adjacency list
    bool      known;
    DistType  dist;    // DistType is probably int
    Vertex    path;    // Probably Vertex *, as mentioned above
        // Other data and member functions as needed
};
```

Figure 9.29 `Vertex` class for Dijkstra's algorithm (pseudocode)

```
/**
 * Print shortest path to v after dijkstra has run.
 * Assume that the path exists.
 */
void Graph::printPath( Vertex v )
{
    if( v.path != NOT_A_VERTEX )
    {
        printPath( v.path );
        cout << " to ";
    }
    cout << v;
}
```

Figure 9.30 Routine to print the actual shortest path

over the previous bound for sparse graphs. Since priority queues do not efficiently support the `find` operation, the location in the priority queue of each value of d_i will need to be maintained and updated whenever d_i changes in the priority queue. If the priority queue is implemented by a binary heap, this will be messy. If a pairing heap (Chapter 12) is used, the code is not too bad.

An alternate method is to insert w and the new value d_w into the priority queue every time w's distance changes. Thus, there may be more than one representative for each vertex in the priority queue. When the `deleteMin` operation removes the smallest vertex from the priority queue, it must be checked to make sure that it is not already *known* and, if

```
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;

        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```

Figure 9.31 Pseudocode for Dijkstra's algorithm

it is, it is simply ignored and another `deleteMin` is performed. Although this method is superior from a software point of view, and is certainly much easier to code, the size of the priority queue could get to be as large as $|E|$. This does not affect the asymptotic time bounds, since $|E| \leq |V|^2$ implies that $\log |E| \leq 2 \log |V|$. Thus, we still get an $O(|E| \log |V|)$ algorithm. However, the space requirement does increase, and this could be important in some applications. Moreover, because this method requires $|E|$ `deleteMin`s instead of only $|V|$, it is likely to be slower in practice.

Notice that for the typical problems, such as computer mail and mass transit commutes, the graphs are typically very sparse because most vertices have only a couple of edges, so it is important in many applications to use a priority queue to solve this problem.

There are better time bounds possible using Dijkstra's algorithm if different data structures are used. In Chapter 11, we will see another priority queue data structure called the

Fibonacci heap. When this is used, the running time is $O(|E|+|V| \log |V|)$. Fibonacci heaps have good theoretical time bounds but a fair amount of overhead, so it is not clear whether using Fibonacci heaps is actually better in practice than Dijkstra's algorithm with binary heaps. To date, there are no meaningful average-case results for this problem.

9.3.3 Graphs with Negative Edge Costs

If the graph has negative edge costs, then Dijkstra's algorithm does not work. The problem is that once a vertex, u, is declared *known,* it is possible that from some other *unknown* vertex, v, there is a path back to u that is very negative. In such a case, taking a path from s to v back to u is better than going from s to u without using v. Exercise 9.7(a) asks you to construct an explicit example.

A tempting solution is to add a constant Δ to each edge cost, thus removing negative edges, calculate a shortest path on the new graph, and then use that result on the original. The naive implementation of this strategy does not work because paths with many edges become more weighty than paths with few edges.

A combination of the weighted and unweighted algorithms will solve the problem, but at the cost of a drastic increase in running time. We forget about the concept of *known* vertices, since our algorithm needs to be able to change its mind. We begin by placing s on a queue. Then, at each stage, we dequeue a vertex v. We find all vertices w adjacent to v such that $d_w > d_v + c_{v,w}$. We update d_w and p_w, and place w on a queue if it is not already there. A bit can be set for each vertex to indicate presence in the queue. We repeat the process until the queue is empty. Figure 9.32 (almost) implements this algorithm.

Although the algorithm works if there are no negative-cost cycles, it is no longer true that the code in the inner `for` loop is executed once per edge. Each vertex can dequeue at most $|V|$ times, so the running time is $O(|E| \cdot |V|)$ if adjacency lists are used (Exercise 9.7(b)). This is quite an increase from Dijkstra's algorithm, so it is fortunate that, in practice, edge costs are nonnegative. If negative-cost cycles are present, then the algorithm as written will loop indefinitely. By stopping the algorithm after any vertex has dequeued $|V| + 1$ times, we can guarantee termination.

9.3.4 Acyclic Graphs

If the graph is known to be acyclic, we can improve Dijkstra's algorithm by changing the order in which vertices are declared *known,* otherwise known as the vertex selection rule. The new rule is to select vertices in topological order. The algorithm can be done in one pass, since the selections and updates can take place as the topological sort is being performed.

This selection rule works because when a vertex v is selected, its distance, d_v, can no longer be lowered, since by the topological ordering rule it has no incoming edges emanating from *unknown* nodes.

There is no need for a priority queue with this selection rule; the running time is $O(|E| + |V|)$, since the selection takes constant time.

An acyclic graph could model some downhill skiing problem—we want to get from point a to b, but can only go downhill, so clearly there are no cycles. Another possible

```
void Graph::weightedNegative( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( v.dist + cvw < w.dist )
            {
                // Update w
                w.dist = v.dist + cvw;
                w.path = v;
                if( w is not already in q )
                    q.enqueue( w );
            }
    }
}
```

Figure 9.32 Pseudocode for weighted shortest-path algorithm with negative edge costs

application might be the modeling of (nonreversible) chemical reactions. We could have each vertex represent a particular state of an experiment. Edges would represent a transition from one state to another, and the edge weights might represent the energy released. If only transitions from a higher energy state to a lower are allowed, the graph is acyclic.

A more important use of acyclic graphs is **critical path analysis.** The graph in Figure 9.33 will serve as our example. Each node represents an activity that must be performed, along with the time it takes to complete the activity. This graph is thus known as an *activity-node* graph. The edges represent precedence relationships: An edge (v, w) means that activity v must be completed before activity w may begin. Of course, this implies that the graph must be acyclic. We assume that any activities that do not depend (either directly or indirectly) on each other can be performed in parallel by different servers.

This type of a graph could be (and frequently is) used to model construction projects. In this case, there are several important questions which would be of interest to answer. First, what is the earliest completion time for the project? We can see from the graph that 10 time units are required along the path A, C, F, H. Another important question is to determine which activities can be delayed, and by how long, without affecting the minimum completion time. For instance, delaying any of A, C, F, or H would push the completion

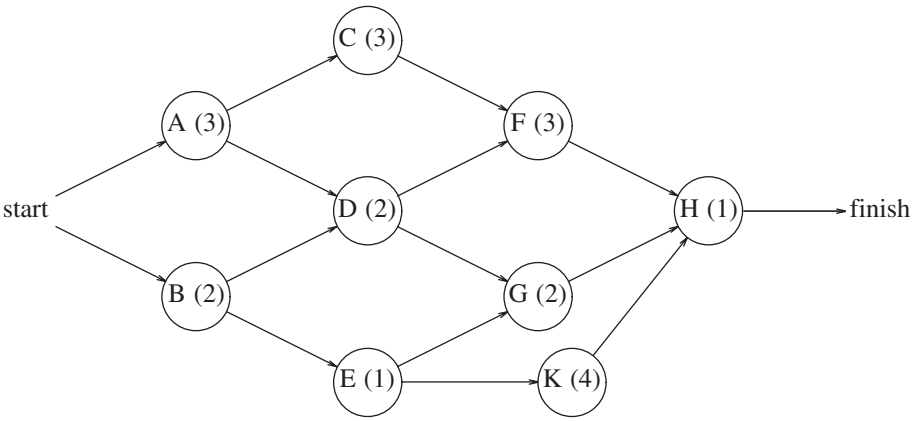

Figure 9.33 Activity-node graph

time past 10 units. On the other hand, activity *B* is less critical and can be delayed up to two time units without affecting the final completion time.

To perform these calculations, we convert the activity-node graph to an **event-node graph**. Each event corresponds to the completion of an activity and all its dependent activities. Events reachable from a node *v* in the event-node graph may not commence until after the event *v* is completed. This graph can be constructed automatically or by hand. Dummy edges and nodes may need to be inserted in the case where an activity depends on several others. This is necessary in order to avoid introducing false dependencies (or false lack of dependencies). The event-node graph corresponding to the graph in Figure 9.33 is shown in Figure 9.34.

To find the earliest completion time of the project, we merely need to find the length of the *longest* path from the first event to the last event. For general graphs, the longest-path problem generally does not make sense, because of the possibility of **positive-cost cycles.** These are the equivalent of negative-cost cycles in shortest-path problems. If positive-cost cycles are present, we could ask for the longest *simple* path, but no satisfactory solution is known for this problem. Since the event-node graph is acyclic, we need not worry about cycles. In this case, it is easy to adapt the shortest-path algorithm to compute the earliest

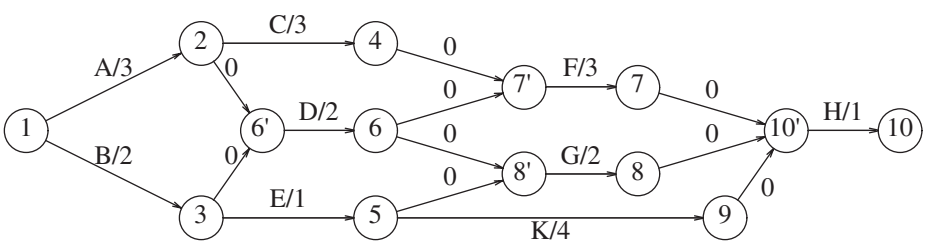

Figure 9.34 Event-node graph

completion time for all nodes in the graph. If EC_i is the earliest completion time for node i, then the applicable rules are

$$EC_1 = 0$$
$$EC_w = \max_{(v,w)\in E}(EC_v + c_{v,w})$$

Figure 9.35 shows the earliest completion time for each event in our example event-node graph.

We can also compute the latest time, LC_i, that each event can finish without affecting the final completion time. The formulas to do this are

$$LC_n = EC_n$$
$$LC_v = \min_{(v,w)\in E}(LC_w - c_{v,w})$$

These values can be computed in linear time by maintaining, for each vertex, a list of all adjacent and preceding vertices. The earliest completion times are computed for vertices by their topological order, and the latest completion times are computed by reverse topological order. The latest completion times are shown in Figure 9.36.

The **slack time** for each edge in the event-node graph represents the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion. It is easy to see that

$$Slack_{(v,w)} = LC_w - EC_v - c_{v,w}$$

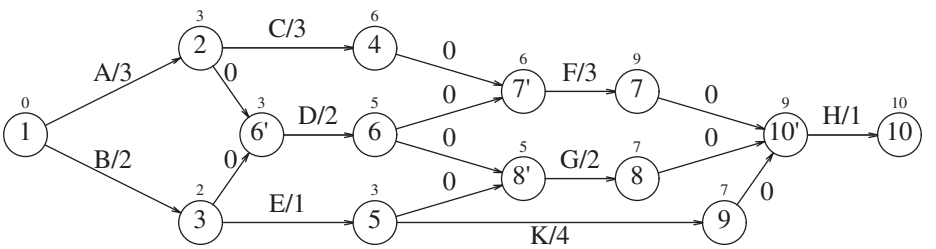

Figure 9.35 Earliest completion times

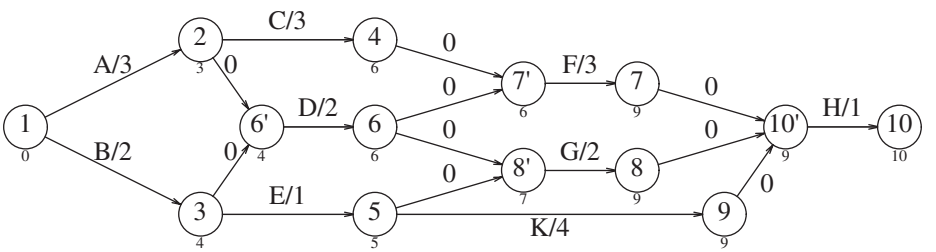

Figure 9.36 Latest completion times

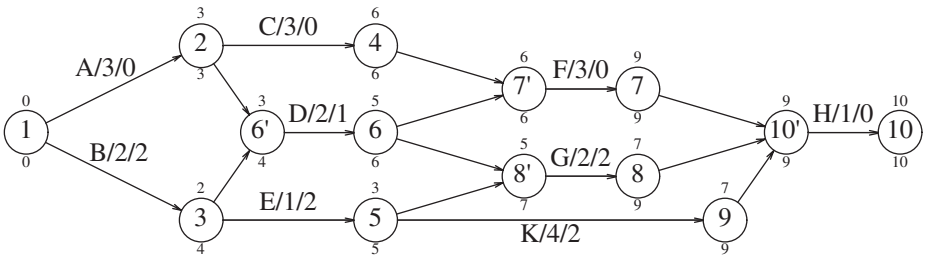

Figure 9.37 Earliest completion time, latest completion time, and slack

Figure 9.37 shows the slack (as the third entry) for each activity in the event-node graph. For each node, the top number is the earliest completion time and the bottom entry is the latest completion time.

Some activities have zero slack. These are critical activities, which must finish on schedule. There is at least one path consisting entirely of zero-slack edges; such a path is a **critical path.**

9.3.5 All-Pairs Shortest Path

Sometimes it is important to find the shortest paths between all pairs of vertices in the graph. Although we could just run the appropriate single-source algorithm $|V|$ times, we might expect a somewhat faster solution, especially on a dense graph, if we compute all the information at once.

In Chapter 10, we will see an $O(|V|^3)$ algorithm to solve this problem for weighted graphs. Although, for dense graphs, this is the same bound as running a simple (non-priority queue) Dijkstra's algorithm $|V|$ times, the loops are so tight that the specialized all-pairs algorithm is likely to be faster in practice. On sparse graphs, of course, it is faster to run $|V|$ Dijkstra's algorithms coded with priority queues.

9.3.6 Shortest Path Example

In this section we write some C++ routines to compute word ladders. In a word ladder each word is formed by changing one character in the ladder's previous word. For instance, we can convert `zero` to `five` by a sequence of one-character substitutions as follows: `zero hero here hire fire five`.

This is an unweighted shortest problem in which each word is a vertex, and two vertices have edges (in both directions) between them if they can be converted to each other with a one-character substitution.

In Section 4.8, we described and wrote a C++ routine that would create a `map` in which the keys are words, and the values are `vector`s containing the words that can result from a one-character transformation. As such, this `map` represents the graph, in adjacency list format, and we only need to write one routine to run the single-source unweighted shortest-path algorithm and a second routine to output the sequence of words, after the

```
// Runs the shortest path calculation from the adjacency map, returning a vector
// that contains the sequence of word changes to get from first to second.
unordered_map<string,string>
findChain( const unordered_map<string,vector<string>> & adjacentWords,
           const string & first, const string & second )
{
    unordered_map<string,string> previousWord;
    queue<string> q;

    q.push( first );

    while( !q.empty( ) )
    {
        string current = q.front( ); q.pop( );
        auto itr = adjacentWords.find( current );

        const vector<string> & adj = itr->second;
        for( string & str : adj )
            if( previousWord[ str ] == "" )
            {
                previousWord[ str ] = current;
                q.push( str );
            }
    }
    previousWord[ first ] = "";

    return previousWord;
}

// After the shortest path calculation has run, computes the vector that
// contains the sequence of words changes to get from first to second.
vector<string> getChainFromPreviousMap(
    const unordered_map<string,string> & previous, const string & second )
{
    vector<string> result;
    auto & prev = const_cast<unordered_map<string,string> &>( previous );

    for( string current = second; current != ""; current = prev[ current ] )
        result.push_back( current );

    reverse( begin( result ), end( result ) );
    return result;
}
```

Figure 9.38 C++ code to find word ladders

single-source shortest-path algorithm has completed. These two routines are both shown in Figure 9.38.

The first routine is `findChain`, which takes the `map` representing the adjacency lists and the two words to be connected and returns a `map` in which the keys are words, and the corresponding value is the word prior to the key on the shortest ladder starting at `first`. In other words, in the example above, if the starting word is `zero`, the value for key `five` is `fire`, the value for key `fire` is `hire`, the value for key `hire` is `here`, and so on. Clearly this provides enough information for the second routine, `getChainFromPreviousMap`, which can work its way backward.

`findChain` is a direct implementation of the pseudocode in Figure 9.18, and for simplicity, it assumes that `first` is a key in `adjacentWords` (this is easily tested prior to the call, or we can add extra code at line 16 that throws an exception if this condition is not satisfied). The basic loop incorrectly assigns a previous entry for `first` (when the initial word adjacent to `first` is processed) so at line 25 that entry is repaired.

`getChainFromPrevMap` uses the `prev` map and `second`, which presumably is a key in the `map` and returns the words used to form the word ladder by working its way backward through `prev`. This generates the words backward, so the STL `reverse` algorithm is used to fix the problem. The cast at line 36 is needed because `operator[]` cannot be applied on an immutable `map`.

It is possible to generalize this problem to allow single-character substitutions that include the deletion of a character or the addition of a character. To compute the adjacency list requires only a little more effort: In the last algorithm in Section 4.8, every time a representative for word w in group g is computed, we check if the representative is a word in group $g - 1$. If it is, then the representative is adjacent to w (it is a single-character deletion), and w is adjacent to the representative (it is a single-character addition). It is also possible to assign a cost to a character deletion or insertion (that is higher than a simple substitution), and this yields a weighted shortest-path problem that can be solved with Dijkstra's algorithm.

9.4 Network Flow Problems

Suppose we are given a directed graph $G = (V, E)$ with edge capacities $c_{v,w}$. These capacities could represent the amount of water that could flow through a pipe or the amount of traffic that could flow on a street between two intersections. We have two vertices: s, which we call the **source**, and t, which is the **sink**. Through any edge, (v, w), at most $c_{v,w}$ units of "flow" may pass. At any vertex, v, that is not either s or t, the total flow coming in must equal the total flow going out. The maximum-flow problem is to determine the maximum amount of flow that can pass from s to t. As an example, for the graph in Figure 9.39 on the left the maximum flow is 5, as indicated by the graph on the right. Although this example graph is acyclic, this is not a requirement; our (eventual) algorithm will work even if the graph has a cycle.

As required by the problem statement, no edge carries more flow than its capacity. Vertex a has three units of flow coming in, which it distributes to c and d. Vertex d takes three units of flow from a and b and combines this, sending the result to t. A vertex can

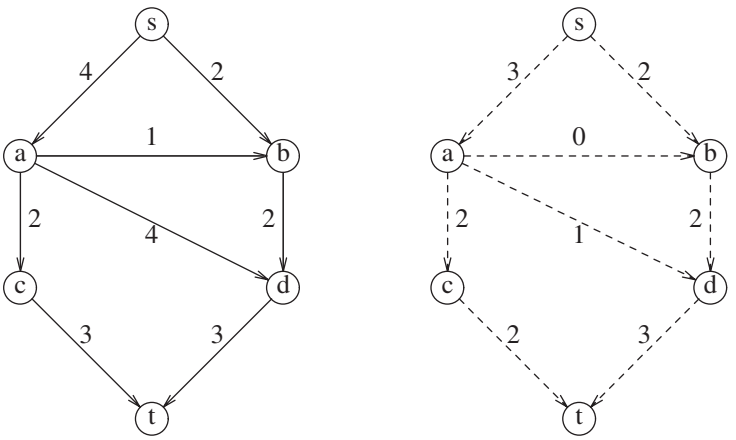

Figure 9.39 A graph (left) and its maximum flow

combine and distribute flow in any manner that it likes, as long as edge capacities are not violated and as long as flow conservation is maintained (what goes in must come out).

Looking at the graph, we see that s has edges of capacities 4 and 2 leaving it, and t has edges of capacities 3 and 3 entering it. So perhaps the maximum flow could be 6 instead of 5. However, Figure 9.40 shows how we can prove that the maximum flow is 5. We cut the graph into two parts; one part contains s and some other vertices; the other part contains t. Since flow must cross through the cut, the total capacity of all edges (u, v) where u is in s's partition and v is in t's partition is a bound on the maximum flow. These edges are (a, c) and (d, t), with total capacity 5, so the maximum flow cannot exceed 5. Any graph has a large number of cuts; the cut with minimum total capacity provides a bound on the maximum flow, and as it turns out (but it is not immediately obvious), the minimum cut capacity is exactly equal to the maximum flow.

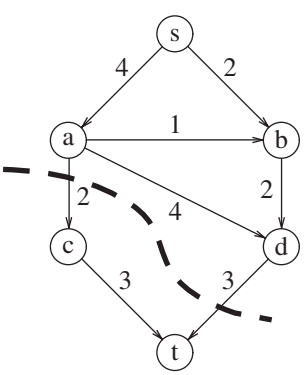

Figure 9.40 A cut in graph G partitions the vertices with s and t in different groups. The total edge cost across the cut is 5, proving that a flow of 5 is maximum.

9.4.1 A Simple Maximum-Flow Algorithm

A first attempt to solve the problem proceeds in stages. We start with our graph, G, and construct a flow graph G_f. G_f tells the flow that has been attained at any stage in the algorithm. Initially all edges in G_f have no flow, and we hope that when the algorithm terminates, G_f contains a maximum flow. We also construct a graph, G_r, called the **residual graph**. G_r tells, for each edge, how much more flow can be added. We can calculate this by subtracting the current flow from the capacity for each edge. An edge in G_r is known as a **residual edge**.

At each stage, we find a path in G_r from s to t. This path is known as an **augmenting path**. The minimum edge on this path is the amount of flow that can be added to every edge on the path. We do this by adjusting G_f and recomputing G_r. When we find no path from s to t in G_r, we terminate. This algorithm is nondeterministic, in that we are free to choose *any* path from s to t; obviously some choices are better than others, and we will address this issue later. We will run this algorithm on our example. The graphs below are G, G_f, G_r, respectively. Keep in mind that there is a slight flaw in this algorithm. The initial configuration is in Figure 9.41.

There are many paths from s to t in the residual graph. Suppose we select s, b, d, t. Then we can send two units of flow through every edge on this path. We will adopt the convention that once we have filled (**saturated**) an edge, it is removed from the residual graph. We then obtain Figure 9.42.

Next, we might select the path s, a, c, t, which also allows two units of flow. Making the required adjustments gives the graphs in Figure 9.43.

The only path left to select is s, a, d, t, which allows one unit of flow. The resulting graphs are shown in Figure 9.44.

The algorithm terminates at this point, because t is unreachable from s. The resulting flow of 5 happens to be the maximum. To see what the problem is, suppose that with our initial graph, we chose the path s, a, d, t. This path allows three units of flow and thus seems to be a good choice. The result of this choice, however, leaves only one path from s to t in the residual graph; it allows one more unit of flow, and thus, our algorithm has

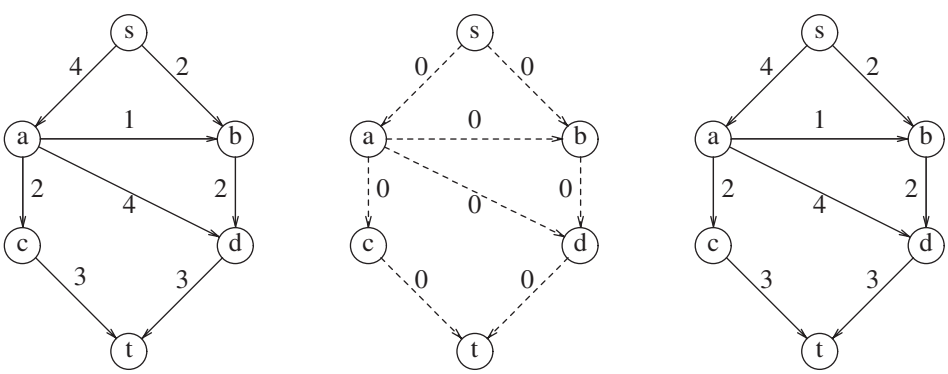

Figure 9.41 Initial stages of the graph, flow graph, and residual graph

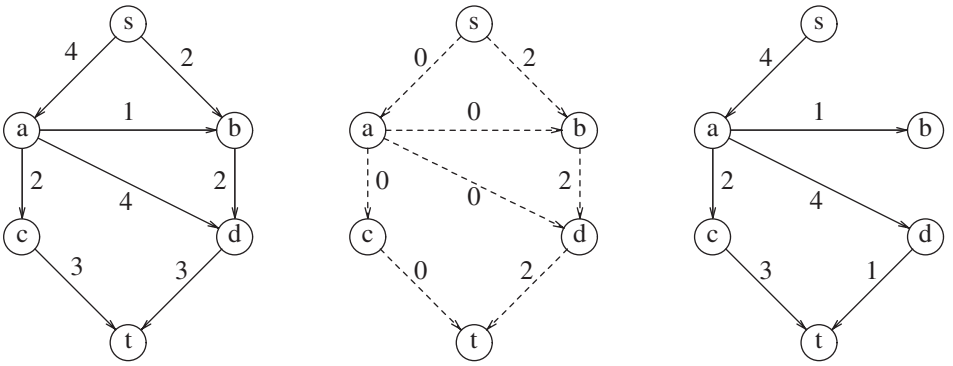

Figure 9.42 G, G_f, G_r after two units of flow added along s, b, d, t

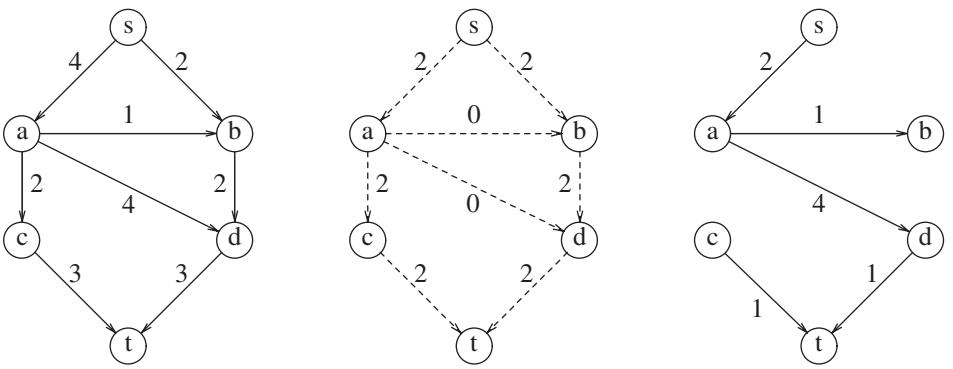

Figure 9.43 G, G_f, G_r after two units of flow added along s, a, c, t

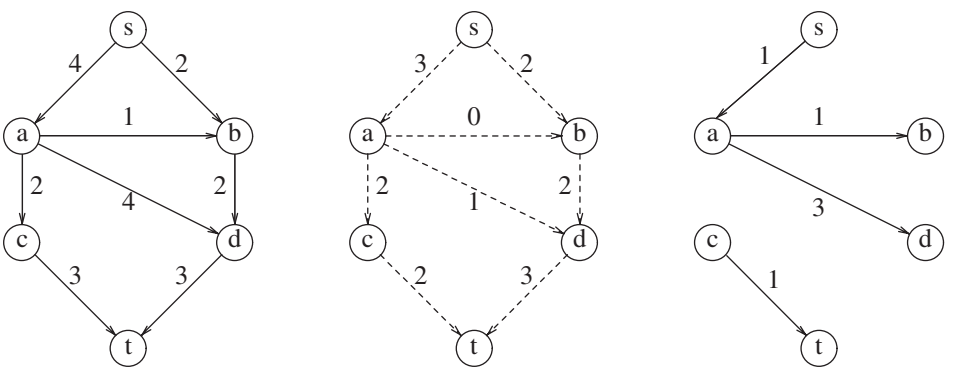

Figure 9.44 G, G_f, G_r after one unit of flow added along s, a, d, t—algorithm terminates

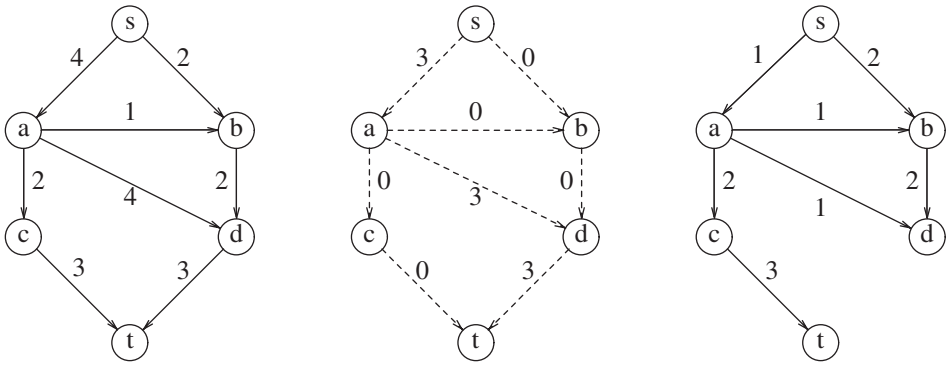

Figure 9.45 G, G_f, G_r if initial action is to add three units of flow along s, a, d, t—algorithm terminates after one more step with suboptimal solution

failed to find an optimal solution. This is an example of a greedy algorithm that does not work. Figure 9.45 shows why the algorithm fails.

In order to make this algorithm work, we need to allow the algorithm to change its mind. To do this, for every edge (v, w) with flow $f_{v,w}$ in the flow graph, we will add an edge in the residual graph (w, v) of capacity $f_{v,w}$. In effect, we are allowing the algorithm to undo its decisions by sending flow back in the opposite direction. This is best seen by example. Starting from our original graph and selecting the augmenting path s, a, d, t, we obtain the graphs in Figure 9.46.

Notice that in the residual graph, there are edges in both directions between a and d. Either one more unit of flow can be pushed from a to d, or up to three units can be pushed back—we can undo flow. Now the algorithm finds the augmenting path s, b, d, a, c, t, of flow 2. By pushing two units of flow from d to a, the algorithm takes two units of flow away from the edge (a, d) and is essentially changing its mind. Figure 9.47 shows the new graphs.

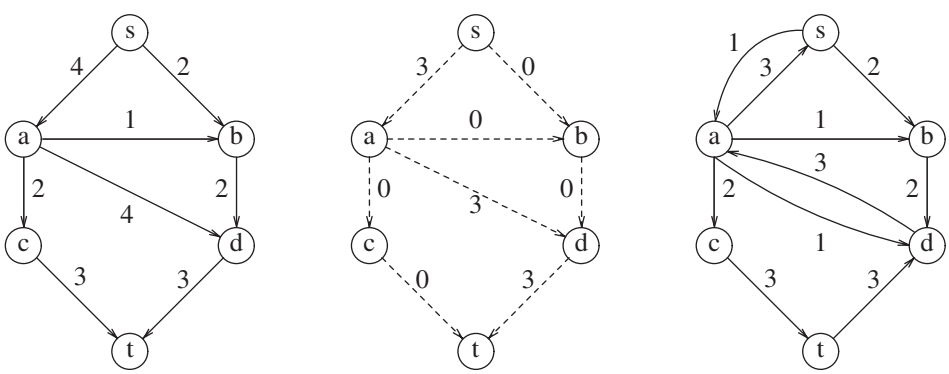

Figure 9.46 Graphs after three units of flow added along s, a, d, t using correct algorithm

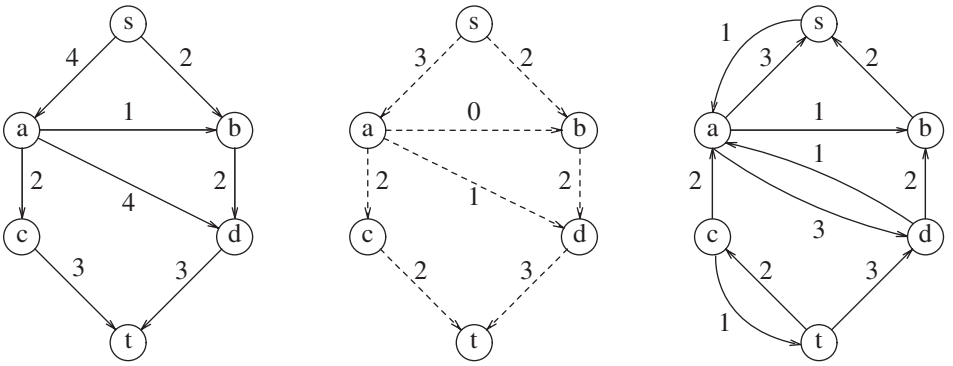

Figure 9.47 Graphs after two units of flow added along s, b, d, a, c, t using correct algorithm

There is no augmenting path in this graph, so the algorithm terminates. Note that the same result would occur if at Figure 9.46, the augmenting path *s*, *a*, *c*, *t* was chosen which allows one unit of flow, because then a subsequent augmenting path could be found.

It is easy to see that *if* the algorithm terminates, then it must terminate with a maximum flow. Termination implies that there is no path from *s* to *t* in the residual graph. So cut the residual graph, putting the vertices reachable from *s* on one side and the unreachables (which include *t*) on the other side. Figure 9.48 shows the cut. Clearly any edges in the original graph *G* that cross the cut must be saturated; otherwise, there would be residual flow remaining on one of the edges, which would then imply an edge that crosses the cut (in the wrong disallowed direction) in G_r. But that means that the flow in *G* is exactly equal to the capacity of a cut in *G*; hence, we have a maximum flow.

If the edge costs in the graph are integers, then the algorithm *must* terminate; each augmentation adds a unit of flow, so we eventually reach the maximum flow, though there

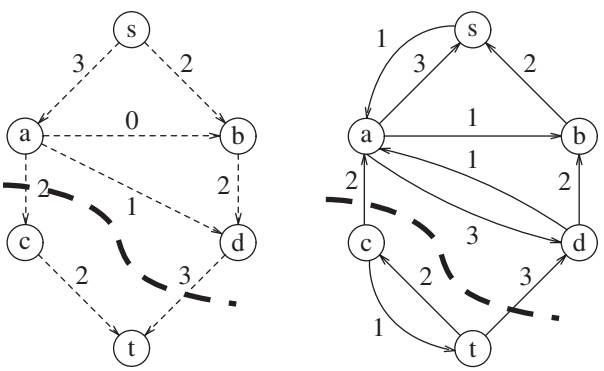

Figure 9.48 The vertices reachable from *s* in the residual graph form one side of a cut; the unreachables form the other side of the cut

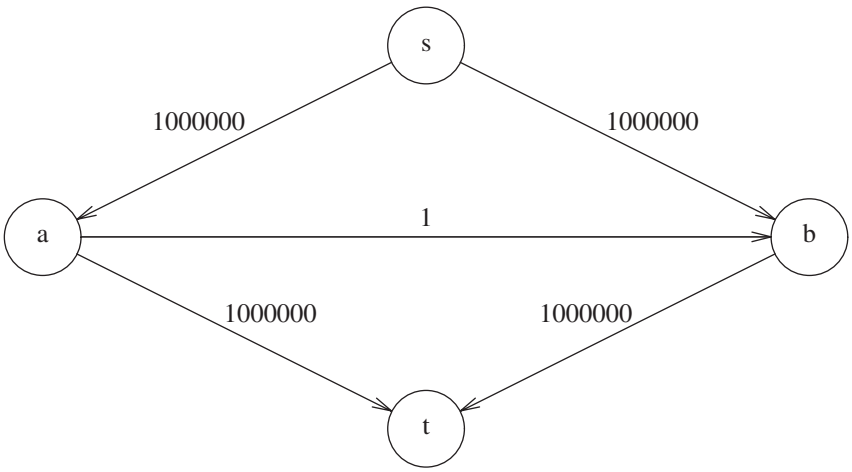

Figure 9.49 The classic bad case for augmenting

is no guarantee that this will be efficient. In particular, if the capacities are all integers and the maximum flow is f, then, since each augmenting path increases the flow value by at least 1, f stages suffice, and the total running time is $O(f \cdot |E|)$, since an augmenting path can be found in $O(|E|)$ time by an unweighted shortest-path algorithm. The classic example of why this is a bad running time is shown by the graph in Figure 9.49.

The maximum flow is seen by inspection to be 2,000,000 by sending 1,000,000 down each side. Random augmentations could continually augment along a path that includes the edge connected by a and b. If this were to occur repeatedly, 2,000,000 augmentations would be required, when we could get by with only 2.

A simple method to get around this problem is always to choose the augmenting path that allows the largest increase in flow. Finding such a path is similar to solving a weighted shortest-path problem, and a single-line modification to Dijkstra's algorithm will do the trick. If $cap_{\max}$ is the maximum edge capacity, then one can show that $O(|E| \log cap_{\max})$ augmentations will suffice to find the maximum flow. In this case, since $O(|E| \log |V|)$ time is used for each calculation of an augmenting path, a total bound of $O(|E|^2 \log |V| \log cap_{\max})$ is obtained. If the capacities are all small integers, this reduces to $O(|E|^2 \log |V|)$.

Another way to choose augmenting paths is always to take the path with the least number of edges, with the plausible expectation that by choosing a path in this manner, it is less likely that a small, flow-restricting edge will turn up on the path. With this rule, each augmenting step computes the shortest unweighted path from s to t in the residual graph, so assume that each vertex in the graph maintains d_v, representing the shortest-path distance from s to v in the residual graph. Each augmenting step can add new edges into the residual graph, but it is clear that no d_v can decrease, because an edge is added in the opposite direction of an existing shortest path.

Each augmenting step saturates at least one edge. Suppose edge (u, v) is saturated; at that point, u had distance d_u and v had distance $d_v = d_u + 1$; then (u, v) was removed from

the residual graph, and edge (v, u) was added. (u, v) cannot reappear in the residual graph again, unless and until (v, u) appears in a future augmenting path. But if it does, then the distance to u at that point must be $d_v + 1$, which would be 2 higher than at the time (u, v) was previously removed.

This means that each time (u, v) reappears, u's distance goes up by 2. This means that any edge can reappear at most $|V|/2$ times. Each augmentation causes some edge to reappear so the number of augmentations is $O(|E||V|)$. Each step takes $O(|E|)$, due to the unweighted shortest-path calculation, yielding an $O(|E|^2|V|)$ bound on the running time.

Further data structure improvements are possible to this algorithm, and there are several, more complicated, algorithms. A long history of improved bounds has lowered the current best-known bound for this problem to $O(|E||V|)$. There are also a host of very good bounds for special cases. For instance, $O(|E||V|^{1/2})$ time finds a maximum flow in a graph, having the property that all vertices except the source and sink have either a single incoming edge of capacity 1 or a single outgoing edge of capacity 1. These graphs occur in many applications.

The analyses required to produce these bounds are rather intricate, and it is not clear how the worst-case results relate to the running times encountered in practice. A related, even more difficult problem is the **min-cost flow** problem. Each edge has not only a capacity but also a cost per unit of flow. The problem is to find, among all maximum flows, the one flow of minimum cost. Both of these problems are being actively researched.

9.5 Minimum Spanning Tree

The next problem we will consider is that of finding a **minimum spanning tree** in an undirected graph. The problem makes sense for directed graphs but appears to be more difficult. Informally, a minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest total cost. A minimum spanning tree exists if and only if G is connected. Although a robust algorithm should report the case that G is unconnected, we will assume that G is connected and leave the issue of robustness as an exercise to the reader.

In Figure 9.50 the second graph is a minimum spanning tree of the first (it happens to be unique, but this is unusual). Notice that the number of edges in the minimum spanning tree is $|V| - 1$. The minimum spanning tree is a *tree* because it is acyclic, it is *spanning* because it covers every vertex, and it is *minimum* for the obvious reason. If we need to wire a house with a minimum of cable (assuming no other electrical constraints), then a minimum spanning tree problem needs to be solved.

For any spanning tree, T, if an edge, e, that is not in T is added, a cycle is created. The removal of any edge on the cycle reinstates the spanning tree property. The cost of the spanning tree is lowered if e has lower cost than the edge that was removed. If, as a spanning tree is created, the edge that is added is the one of minimum cost that avoids creation of a cycle, then the cost of the resulting spanning tree cannot be improved, because any replacement edge would have cost at least as much as an edge already in the spanning tree. This shows that greed works for the minimum spanning tree problem. The two algorithms we present differ in how a minimum edge is selected.

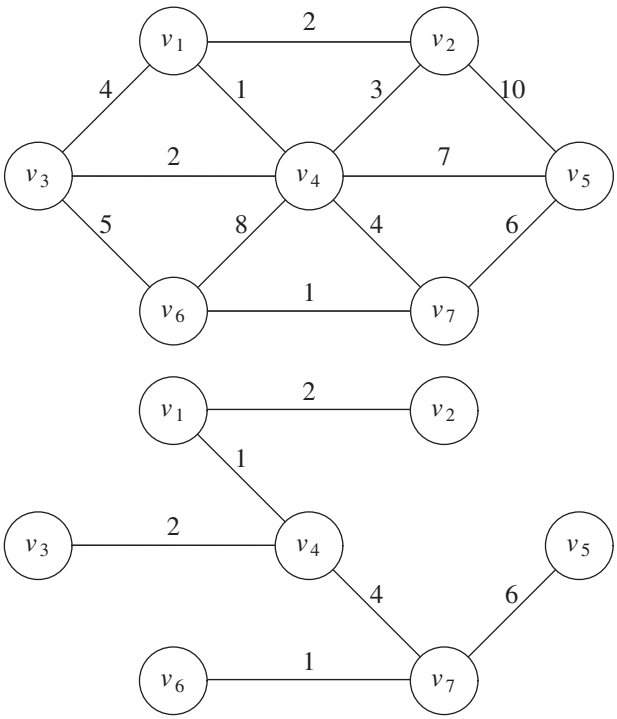

Figure 9.50 A graph G and its minimum spanning tree

9.5.1 Prim's Algorithm

One way to compute a minimum spanning tree is to grow the tree in successive stages. In each stage, one node is picked as the root, and we add an edge, and thus an associated vertex, to the tree.

At any point in the algorithm, we can see that we have a set of vertices that have already been included in the tree; the rest of the vertices have not. The algorithm then finds, at each stage, a new vertex to add to the tree by choosing the edge (u, v) such that the cost of (u, v) is the smallest among all edges where u is in the tree and v is not. Figure 9.51 shows how this algorithm would build the minimum spanning tree, starting from v_1. Initially, v_1 is in the tree as a root with no edges. Each step adds one edge and one vertex to the tree.

We can see that Prim's algorithm is essentially identical to Dijkstra's algorithm for shortest paths. As before, for each vertex we keep values d_v and p_v and an indication of whether it is *known* or *unknown*. d_v is the weight of the shortest edge connecting v to a *known* vertex, and p_v, as before, is the last vertex to cause a change in d_v. The rest of the algorithm is exactly the same, with the exception that since the definition of d_v is different, so is the update rule. For this problem, the update rule is even simpler than before: After a vertex, v, is selected, for each *unknown* w adjacent to v, $d_w = \min(d_w, c_{w,v})$.

The initial configuration of the table is shown in Figure 9.52. v_1 is selected, and v_2, v_3, and v_4 are updated. The table resulting from this is shown in Figure 9.53. The next vertex

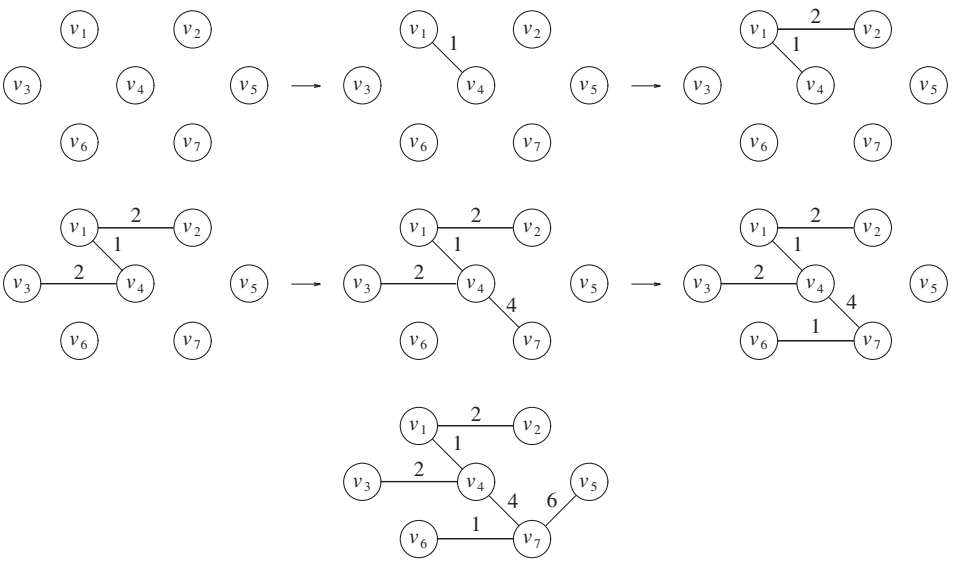

Figure 9.51 Prim's algorithm after each stage

v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.52 Initial configuration of table used in Prim's algorithm

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	4	v_1
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Figure 9.53 The table after v_1 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	8	v_4
v_7	F	4	v_4

Figure 9.54 The table after v_4 is declared *known*

selected is v_4. Every vertex is adjacent to v_4. v_1 is not examined, because it is *known*. v_2 is unchanged, because it has $d_v = 2$ and the edge cost from v_4 to v_2 is 3; all the rest are updated. Figure 9.54 shows the resulting table. The next vertex chosen is v_2 (arbitrarily breaking a tie). This does not affect any distances. Then v_3 is chosen, which affects the distance in v_6, producing Figure 9.55. Figure 9.56 results from the selection of v_7, which forces v_6 and v_5 to be adjusted. v_6 and then v_5 are selected, completing the algorithm.

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	7	v_4
v_6	F	5	v_3
v_7	F	4	v_4

Figure 9.55 The table after v_2 and then v_3 are declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	F	6	v_7
v_6	F	1	v_7
v_7	T	4	v_4

Figure 9.56 The table after v_7 is declared *known*

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	2	v_4
v_4	T	1	v_1
v_5	T	6	v_7
v_6	T	1	v_7
v_7	T	4	v_4

Figure 9.57 The table after v_6 and v_5 are selected (Prim's algorithm terminates)

The final table is shown in Figure 9.57. The edges in the spanning tree can be read from the table: (v_2, v_1), (v_3, v_4), (v_4, v_1), (v_5, v_7), (v_6, v_7), (v_7, v_4). The total cost is 16.

The entire implementation of this algorithm is virtually identical to that of Dijkstra's algorithm, and everything that was said about the analysis of Dijkstra's algorithm applies here. Be aware that Prim's algorithm runs on undirected graphs, so when coding it, remember to put every edge in two adjacency lists. The running time is $O(|V|^2)$ without heaps, which is optimal for dense graphs, and $O(|E| \log |V|)$ using binary heaps, which is good for sparse graphs.

9.5.2 Kruskal's Algorithm

A second greedy strategy is to continually select the edges in order of smallest weight and accept an edge if it does not cause a cycle. The action of the algorithm on the graph in the preceding example is shown in Figure 9.58.

Edge	Weight	Action
(v_1, v_4)	1	Accepted
(v_6, v_7)	1	Accepted
(v_1, v_2)	2	Accepted
(v_3, v_4)	2	Accepted
(v_2, v_4)	3	Rejected
(v_1, v_3)	4	Rejected
(v_4, v_7)	4	Accepted
(v_3, v_6)	5	Rejected
(v_5, v_7)	6	Accepted

Figure 9.58 Action of Kruskal's algorithm on G

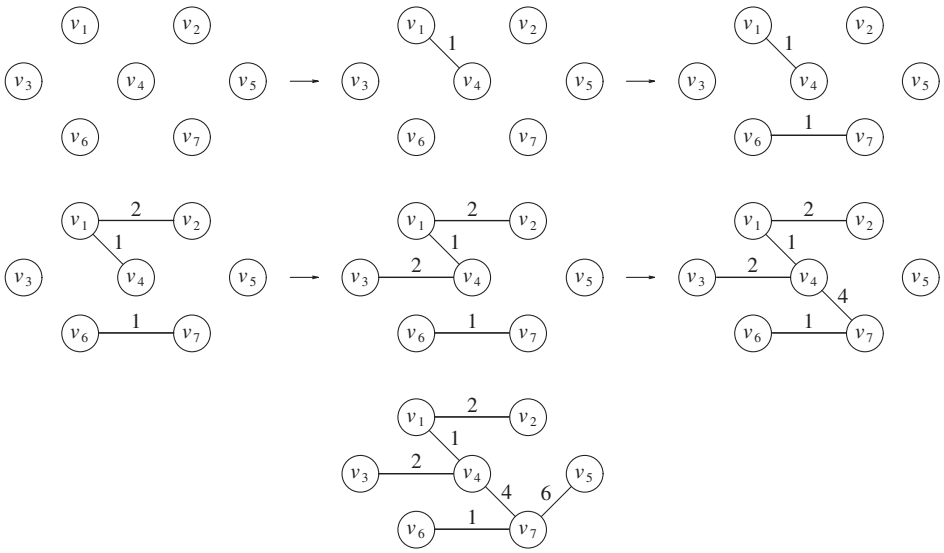

Figure 9.59 Kruskal's algorithm after each stage

Formally, Kruskal's algorithm maintains a forest—a collection of trees. Initially, there are $|V|$ single-node trees. Adding an edge merges two trees into one. When the algorithm terminates, there is only one tree, and this is the minimum spanning tree. Figure 9.59 shows the order in which edges are added to the forest.

The algorithm terminates when enough edges are accepted. It turns out to be simple to decide whether edge (u, v) should be accepted or rejected. The appropriate data structure is the union/find algorithm from Chapter 8.

The invariant we will use is that at any point in the process, two vertices belong to the same set if and only if they are connected in the current spanning forest. Thus, each vertex is initially in its own set. If u and v are in the same set, the edge is rejected, because since they are already connected, adding (u, v) would form a cycle. Otherwise, the edge is accepted, and a `union` is performed on the two sets containing u and v. It is easy to see that this maintains the set invariant, because once the edge (u, v) is added to the spanning forest, if w was connected to u and x was connected to v, then x and w must now be connected, and thus belong in the same set.

The edges could be sorted to facilitate the selection, but building a heap in linear time is a much better idea. Then `deleteMin`s give the edges to be tested in order. Typically, only a small fraction of the edges need to be tested before the algorithm can terminate, although it is always possible that all the edges must be tried. For instance, if there was an extra vertex v_8 and edge (v_5, v_8) of cost 100, all the edges would have to be examined. Function `kruskal` in Figure 9.60 finds a minimum spanning tree.

The worst-case running time of this algorithm is $O(|E| \log |E|)$, which is dominated by the heap operations. Notice that since $|E| = O(|V|^2)$, this running time is

```
vector<Edge> kruskal( vector<Edge> edges, int numVertices )
{
    DisjSets ds{ numVertices };
    priority_queue pq{ edges };
    vector<Edge> mst;

    while( mst.size( ) != numVertices - 1 )
    {
        Edge e = pq.pop( );       // Edge e = (u, v)
        SetType uset = ds.find( e.getu( ) );
        SetType vset = ds.find( e.getv( ) );

        if( uset != vset )
        {
            // Accept the edge
            mst.push_back( e );
            ds.union( uset, vset );
        }
    }

    return mst;
}
```

Figure 9.60 Pseudocode for Kruskal's algorithm

actually $O(|E| \log |V|)$. In practice, the algorithm is much faster than this time bound would indicate.

9.6 Applications of Depth-First Search

Depth-first search is a generalization of preorder traversal. Starting at some vertex, v, we process v and then recursively traverse all vertices adjacent to v. If this process is performed on a tree, then all tree vertices are systematically visited in a total of $O(|E|)$ time, since $|E| = \Theta(|V|)$. If we perform this process on an arbitrary graph, we need to be careful to avoid cycles. To do this, when we visit a vertex, v, we *mark* it visited, since now we have been there, and recursively call depth-first search on all adjacent vertices that are not already marked. We implicitly assume that for undirected graphs every edge (v, w) appears twice in the adjacency lists: once as (v, w) and once as (w, v). The procedure in Figure 9.61 performs a depth-first search (and does absolutely nothing else) and is a template for the general style.

For each vertex, the data member `visited` is initialized to `false`. By recursively calling the procedures only on nodes that have not been visited, we guarantee that we do not loop indefinitely. If the graph is undirected and not connected, or directed and not strongly connected, this strategy might fail to visit some nodes. We then search for an unmarked node,

```
void Graph::dfs( Vertex v )
{
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}
```

Figure 9.61 Template for depth-first search (pseudocode)

apply a depth-first traversal there, and continue this process until there are no unmarked nodes.[2] Because this strategy guarantees that each edge is encountered only once, the total time to perform the traversal is $O(|E| + |V|)$, as long as adjacency lists are used.

9.6.1 Undirected Graphs

An undirected graph is connected if and only if a depth-first search starting from any node visits every node. Because this test is so easy to apply, we will assume that the graphs we deal with are connected. If they are not, then we can find all the connected components and apply our algorithm on each of these in turn.

As an example of depth-first search, suppose in the graph of Figure 9.62 we start at vertex A. Then we mark A as visited and call `dfs(B)` recursively. `dfs(B)` marks B as visited and calls `dfs(C)` recursively. `dfs(C)` marks C as visited and calls `dfs(D)` recursively. `dfs(D)` sees both A and B, but both of these are marked, so no recursive calls are made. `dfs(D)` also sees that C is adjacent but marked, so no recursive call is made there, and `dfs(D)` returns back to `dfs(C)`. `dfs(C)` sees B adjacent, ignores it, finds a previously unseen vertex E adjacent, and thus calls `dfs(E)`. `dfs(E)` marks E, ignores A and C, and returns to `dfs(C)`. `dfs(C)` returns to `dfs(B)`. `dfs(B)` ignores both A and D and returns. `dfs(A)` ignores both D and E and returns. (We have actually touched every edge twice, once as (v, w) and again as (w, v), but this is really once per adjacency list entry.)

We graphically illustrate these steps with a **depth-first spanning tree.** The root of the tree is A, the first vertex visited. Each edge (v, w) in the graph is present in the tree. If, when we process (v, w), we find that w is unmarked, or if, when we process (w, v), we find that v is unmarked, we indicate this with a tree edge. If, when we process (v, w), we find that w is already marked, and when processing (w, v), we find that v is already marked, we draw a dashed line, which we will call a **back edge,** to indicate that this "edge" is not really part of the tree. The depth-first search of the graph in Figure 9.62 is shown in Figure 9.63.

The tree will simulate the traversal we performed. A preorder numbering of the tree, using only tree edges, tells us the order in which the vertices were marked. If the graph is not connected, then processing all nodes (and edges) requires several calls to `dfs`, and each generates a tree. This entire collection is a **depth-first spanning forest.**

[2] An efficient way of implementing this is to begin the depth-first search at v_1. If we need to restart the depth-first search, we examine the sequence $v_k, v_{k+1}, \ldots$ for an unmarked vertex, where v_{k-1} is the vertex where the last depth-first search was started. This guarantees that throughout the algorithm, only $O(|V|)$ is spent looking for vertices where new depth-first search trees can be started.

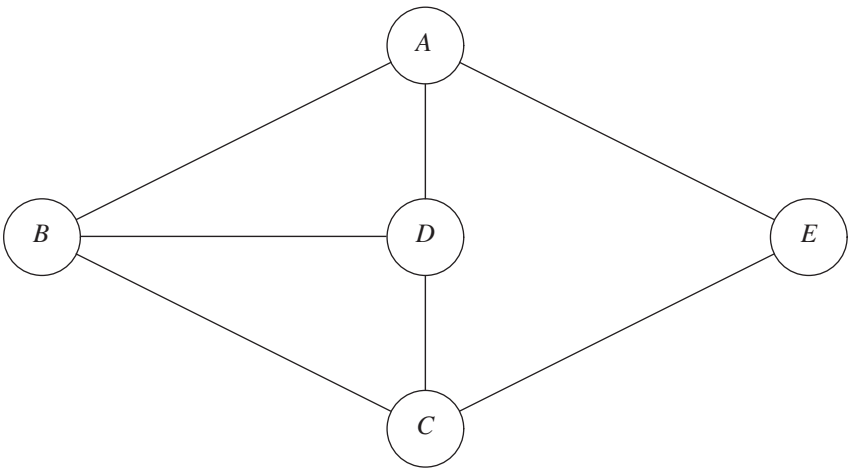

Figure 9.62 An undirected graph

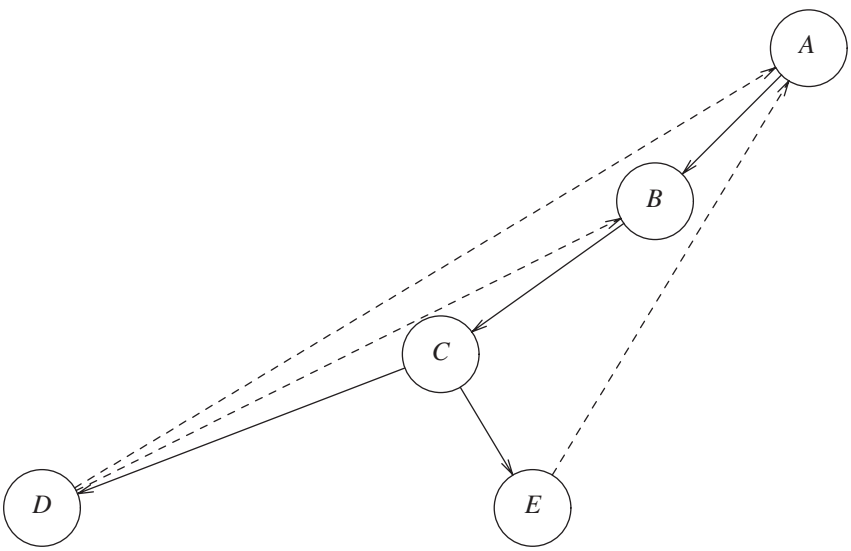

Figure 9.63 Depth-first search of previous graph

9.6.2 Biconnectivity

A connected undirected graph is **biconnected** if there are no vertices whose removal disconnects the rest of the graph. The graph in Figure 9.62 is biconnected. If the nodes are computers and the edges are links, then if any computer goes down, network mail is

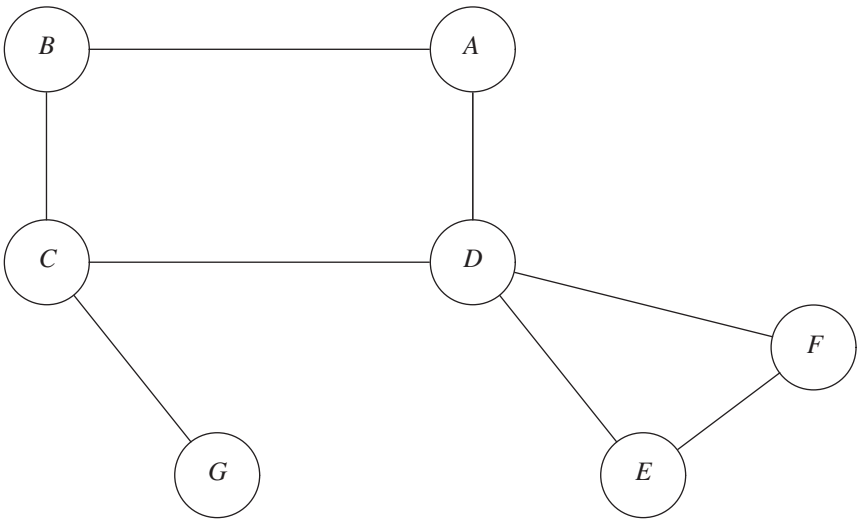

Figure 9.64 A graph with articulation points *C* and *D*

unaffected, except, of course, at the down computer. Similarly, if a mass transit system is biconnected, users always have an alternate route should some terminal be disrupted.

If a graph is not biconnected, the vertices whose removal would disconnect the graph are known as **articulation points.** These nodes are critical in many applications. The graph in Figure 9.64 is not biconnected: *C* and *D* are articulation points. The removal of *C* would disconnect *G*, and the removal of *D* would disconnect *E* and *F*, from the rest of the graph.

Depth-first search provides a linear-time algorithm to find all articulation points in a connected graph. First, starting at any vertex, we perform a depth-first search and number the nodes as they are visited. For each vertex, v, we call this preorder number $Num(v)$. Then, for every vertex, v, in the depth-first search spanning tree, we compute the lowest-numbered vertex, which we call $Low(v)$, that is reachable from v by taking zero or more tree edges and then possibly one back edge (in that order). The depth-first search tree in Figure 9.65 shows the preorder number first, and then the lowest-numbered vertex reachable under the rule described above.

The lowest-numbered vertex reachable by *A*, *B*, and *C* is vertex 1 (*A*), because they can all take tree edges to *D* and then one back edge back to *A*. We can efficiently compute *Low* by performing a postorder traversal of the depth-first spanning tree. By the definition of *Low*, $Low(v)$ is the minimum of

1. $Num(v)$
2. the lowest $Num(w)$ among all back edges (v, w)
3. the lowest $Low(w)$ among all tree edges (v, w)

The first condition is the option of taking no edges, the second way is to choose no tree edges and a back edge, and the third way is to choose some tree edges and possibly a

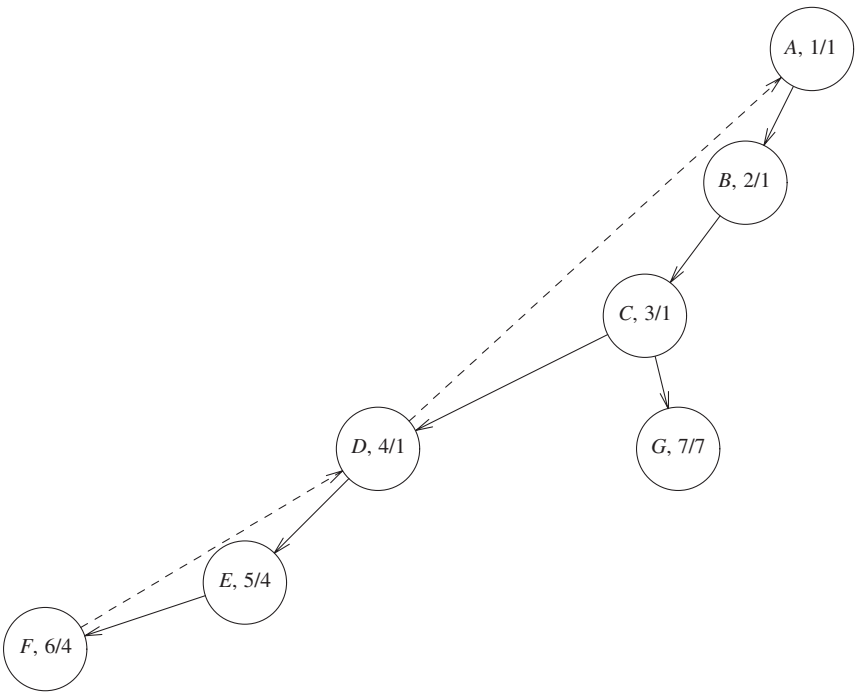

Figure 9.65 Depth-first tree for previous graph, with *Num* and *Low*

back edge. This third method is succinctly described with a recursive call. Since we need to evaluate *Low* for all the children of v before we can evaluate $Low(v)$, this is a postorder traversal. For any edge (v, w), we can tell whether it is a tree edge or a back edge merely by checking $Num(v)$ and $Num(w)$. Thus, it is easy to compute $Low(v)$: We merely scan down v's adjacency list, apply the proper rule, and keep track of the minimum. Doing all the computation takes $O(|E| + |V|)$ time.

All that is left to do is to use this information to find articulation points. The root is an articulation point if and only if it has more than one child, because if it has two children, removing the root disconnects nodes in different subtrees, and if it has only one child, removing the root merely disconnects the root. Any other vertex v is an articulation point if and only if v has some child w such that $Low(w) \geq Num(v)$. Notice that this condition is always satisfied at the root, hence the need for a special test.

The *if* part of the proof is clear when we examine the articulation points that the algorithm determines, namely, C and D. D has a child E, and $Low(E) \geq Num(D)$, since both are 4. Thus, there is only one way for E to get to any node above D, and that is by going through D. Similarly, C is an articulation point, because $Low(G) \geq Num(C)$. To prove that this algorithm is correct, one must show that the *only if* part of the assertion is true (that is, this finds *all* articulation points). We leave this as an exercise. As a second example, we show (Fig. 9.66) the result of applying this algorithm on the same graph, starting the depth-first search at C.

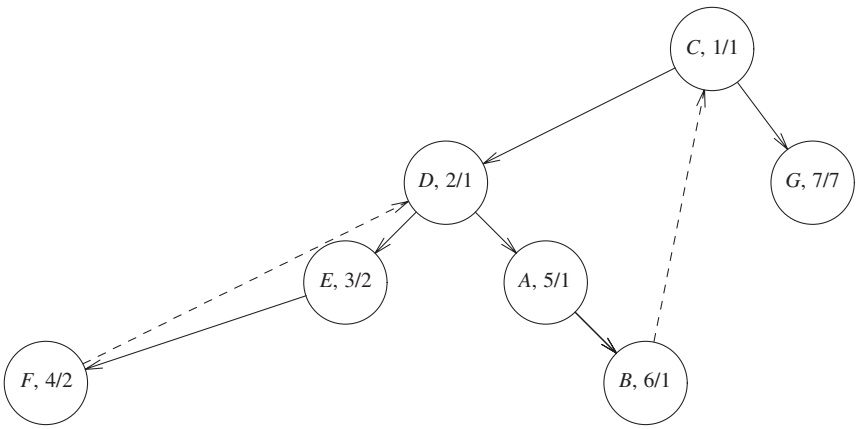

Figure 9.66 Depth-first tree that results if depth-first search starts at *C*

We close by giving pseudocode to implement this algorithm. We will assume that `Vertex` contains the data members `visited` (initialized to `false`), `num`, `low`, and `parent`. We will also keep a (`Graph`) class variable called `counter`, which is initialized to 1, to assign the preorder traversal numbers, `num`. We also leave out the easily implemented test for the root.

As we have already stated, this algorithm can be implemented by performing a preorder traversal to compute *Num* and then a postorder traversal to compute *Low*. A third traversal can be used to check which vertices satisfy the articulation point criteria. Performing three traversals, however, would be a waste. The first pass is shown in Figure 9.67.

The second and third passes, which are postorder traversals, can be implemented by the code in Figure 9.68. The last `if` statement handles a special case. If *w* is adjacent to

```
/**
 * Assign num and compute parents.
 */
void Graph::assignNum( Vertex v )
{
    v.num = counter++;
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
        {
            w.parent = v;
            assignNum( w );
        }
}
```

Figure 9.67 Routine to assign *Num* to vertices (pseudocode)

```
/**
 * Assign low; also check for articulation points.
 */
void Graph::assignLow( Vertex v )
{
    v.low = v.num;  // Rule 1
    for each Vertex w adjacent to v
    {
        if( w.num > v.num )  // Forward edge
        {
            assignLow( w );
            if( w.low >= v.num )
                cout << v << " is an articulation point" << endl;
            v.low = min( v.low, w.low );  // Rule 3
        }
        else
        if( v.parent != w )  // Back edge
            v.low = min( v.low, w.num );  // Rule 2
    }
}
```

Figure 9.68 Pseudocode to compute *Low* and to test for articulation points (test for the root is omitted)

v, then the recursive call to *w* will find *v* adjacent to *w*. This is not a back edge, only an edge that has already been considered and needs to be ignored. Otherwise, the procedure computes the minimum of the various `low` and `num` entries, as specified by the algorithm.

There is no rule that a traversal must be either preorder or postorder. It is possible to do processing both before and after the recursive calls. The procedure in Figure 9.69 combines the two routines `assignNum` and `assignLow` in a straightforward manner to produce the procedure `findArt`.

9.6.3 Euler Circuits

Consider the three figures in Figure 9.70. A popular puzzle is to reconstruct these figures using a pen, drawing each line exactly once. The pen may not be lifted from the paper while the drawing is being performed. As an extra challenge, make the pen finish at the same point at which it started. This puzzle has a surprisingly simple solution. Stop reading if you would like to try to solve it.

The first figure can be drawn only if the starting point is the lower left- or right-hand corner, and it is not possible to finish at the starting point. The second figure is easily drawn with the finishing point the same as the starting point, but the third figure cannot be drawn at all within the parameters of the puzzle.

We can convert this problem to a graph theory problem by assigning a vertex to each intersection. Then the edges can be assigned in the natural manner, as in Figure 9.71.

```
void Graph::findArt( Vertex v )
{
    v.visited = true;
    v.low = v.num = counter++;  // Rule 1
    for each Vertex w adjacent to v
    {
        if( !w.visited )  // Forward edge
        {
            w.parent = v;
            findArt( w );
            if( w.low >= v.num )
                cout << v << " is an articulation point" << endl;
            v.low = min( v.low, w.low );  // Rule 3
        }
        else
        if( v.parent != w )  // Back edge
            v.low = min( v.low, w.num );  // Rule 2
    }
}
```

Figure 9.69 Testing for articulation points in one depth-first search (test for the root is omitted) (pseudocode)

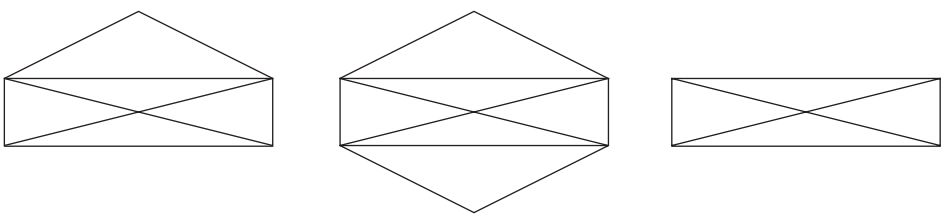

Figure 9.70 Three drawings

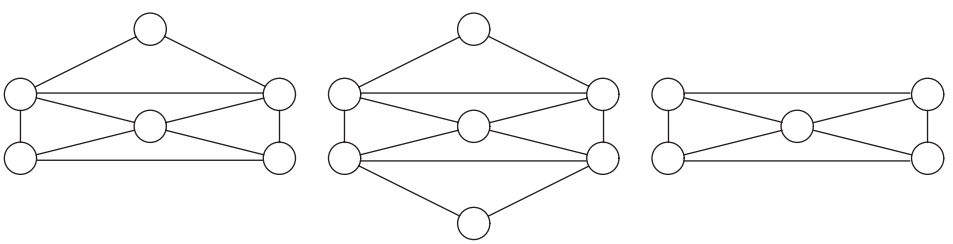

Figure 9.71 Conversion of puzzle to graph

After this conversion is performed, we must find a path in the graph that visits every edge exactly once. If we are to solve the "extra challenge," then we must find a cycle that visits every edge exactly once. This graph problem was solved in 1736 by Euler and marked the beginning of graph theory. The problem is thus commonly referred to as an **Euler path** (sometimes **Euler tour**) or **Euler circuit problem,** depending on the specific problem statement. The Euler tour and Euler circuit problems, though slightly different, have the same basic solution. Thus, we will consider the Euler circuit problem in this section.

The first observation that can be made is that an Euler circuit, which must end on its starting vertex, is possible only if the graph is connected and each vertex has an even degree (number of edges). This is because, on the Euler circuit, a vertex is entered and then left. If any vertex v has odd degree, then eventually we will reach the point where only one edge into v is unvisited, and taking it will strand us at v. If exactly two vertices have odd degree, an Euler tour, which must visit every edge but need not return to its starting vertex, is still possible if we start at one of the odd-degree vertices and finish at the other. If more than two vertices have odd degree, then an Euler tour is not possible.

The observations of the preceding paragraph provide us with a necessary condition for the existence of an Euler circuit. It does not, however, tell us that all connected graphs that satisfy this property must have an Euler circuit, nor does it give us guidance on how to find one. It turns out that the necessary condition is also sufficient. That is, any connected graph, all of whose vertices have even degree, must have an Euler circuit. Furthermore, a circuit can be found in linear time.

We can assume that we know that an Euler circuit exists, since we can test the necessary and sufficient condition in linear time. Then the basic algorithm is to perform a depth-first search. There are a surprisingly large number of "obvious" solutions that do not work. Some of these are presented in the exercises.

The main problem is that we might visit a portion of the graph and return to the starting point prematurely. If all the edges coming out of the start vertex have been used up, then part of the graph is untraversed. The easiest way to fix this is to find the first vertex on this path that has an untraversed edge and perform another depth-first search. This will give another circuit, which can be spliced into the original. This is continued until all edges have been traversed.

As an example, consider the graph in Figure 9.72. It is easily seen that this graph has an Euler circuit. Suppose we start at vertex 5, and traverse the circuit 5, 4, 10, 5. Then we are stuck, and most of the graph is still untraversed. The situation is shown in Figure 9.73.

We then continue from vertex 4, which still has unexplored edges. A depth-first search might come up with the path 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4. If we splice this path into the previous path of 5, 4, 10, 5, then we get a new path of 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

The graph that remains after this is shown in Figure 9.74. Notice that in this graph, all the vertices must have even degree, so we are guaranteed to find a cycle to add. The remaining graph might not be connected, but this is not important. The next vertex on the path that has untraversed edges is vertex 3. A possible circuit would then be 3, 2, 8, 9, 6, 3. When spliced in, this gives the path 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

The graph that remains is in Figure 9.75. On this path, the next vertex with an untraversed edge is 9, and the algorithm finds the circuit 9, 12, 10, 9. When this is added to the

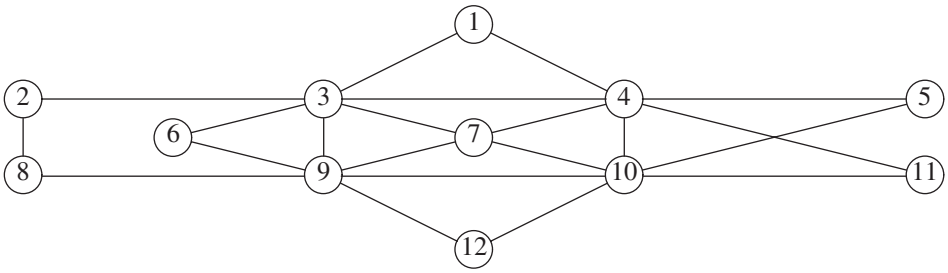

Figure 9.72 Graph for Euler circuit problem

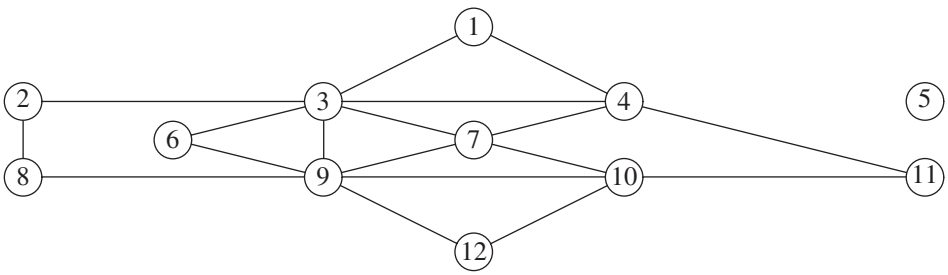

Figure 9.73 Graph remaining after 5, 4, 10, 5

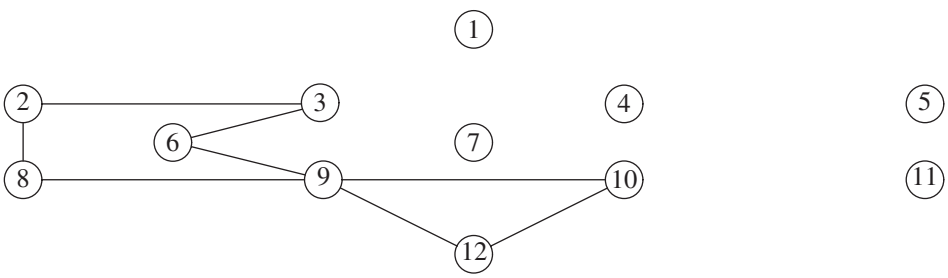

Figure 9.74 Graph after the path 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

current path, a circuit of 5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5 is obtained. As all the edges are traversed, the algorithm terminates with an Euler circuit.

To make this algorithm efficient, we must use appropriate data structures. We will sketch some of the ideas, leaving the implementation as an exercise. To make splicing simple, the path should be maintained as a linked list. To avoid repetitious scanning of adjacency lists, we must maintain, for each adjacency list, a pointer to the last edge scanned. When a path is spliced in, the search for a new vertex from which to perform the next depth-first search must begin at the start of the splice point. This guarantees that

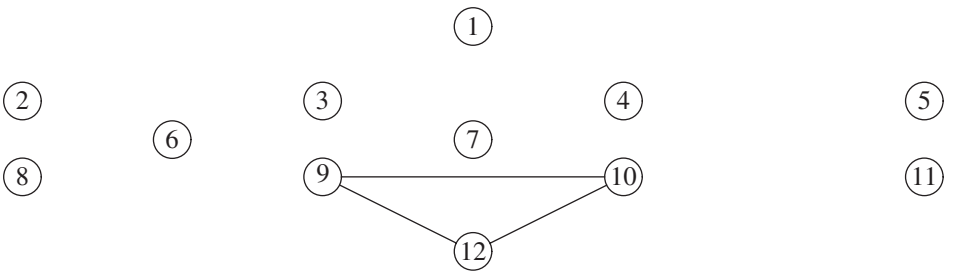

Figure 9.75 Graph remaining after the path 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

the total work performed on the vertex search phase is $O(|E|)$ during the entire life of the algorithm. With the appropriate data structures, the running time of the algorithm is $O(|E| + |V|)$.

A very similar problem is to find a simple cycle, in an undirected graph, that visits every vertex. This is known as the **Hamiltonian cycle problem**. Although it seems almost identical to the Euler circuit problem, no efficient algorithm for it is known. We shall see this problem again in Section 9.7.

9.6.4 Directed Graphs

Using the same strategy as with undirected graphs, directed graphs can be traversed in linear time, using depth-first search. If the graph is not strongly connected, a depth-first search starting at some node might not visit all nodes. In this case, we repeatedly perform depth-first searches, starting at some unmarked node, until all vertices have been visited. As an example, consider the directed graph in Figure 9.76.

We arbitrarily start the depth-first search at vertex B. This visits vertices B, C, A, D, E, and F. We then restart at some unvisited vertex. Arbitrarily, we start at H, which visits J and I. Finally, we start at G, which is the last vertex that needs to be visited. The corresponding depth-first search tree is shown in Figure 9.77.

The dashed arrows in the depth-first spanning forest are edges (v, w) for which w was already marked at the time of consideration. In undirected graphs, these are always back edges, but, as we can see, there are three types of edges that do not lead to new vertices. First, there are **back edges,** such as (A, B) and (I, H). There are also **forward edges,** such as (C, D) and (C, E), that lead from a tree node to a descendant. Finally, there are **cross edges,** such as (F, C) and (G, F), which connect two tree nodes that are not directly related. Depth-first search forests are generally drawn with children and new trees added to the forest from left to right. In a depth-first search of a directed graph drawn in this manner, cross edges always go from right to left.

Some algorithms that use depth-first search need to distinguish between the three types of nontree edges. This is easy to check as the depth-first search is being performed, and it is left as an exercise.

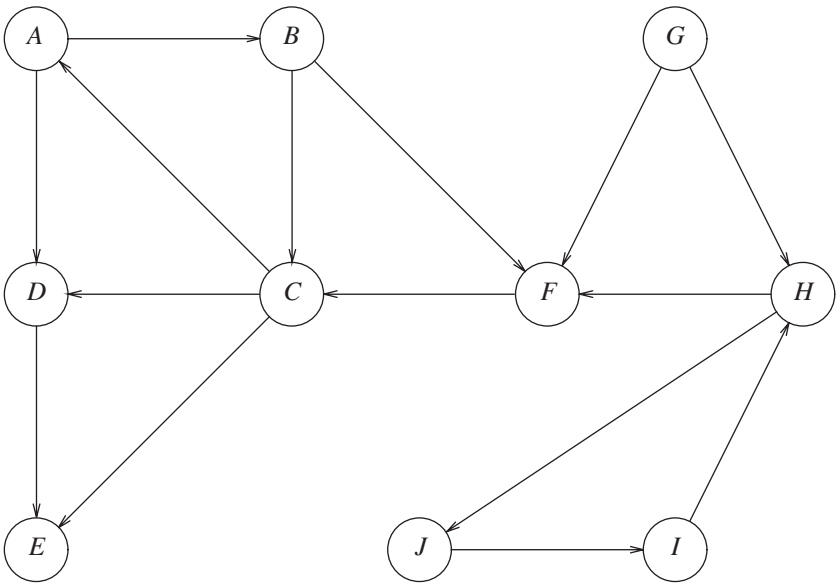

Figure 9.76 A directed graph

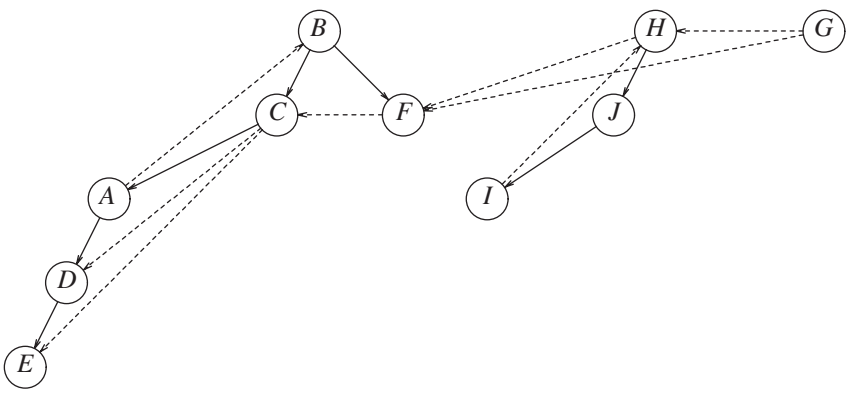

Figure 9.77 Depth-first search of previous graph

One use of depth-first search is to test whether or not a directed graph is acyclic. The rule is that a directed graph is acyclic if and only if it has no back edges. (The graph above has back edges, and thus is not acyclic.) The reader may remember that a topological sort can also be used to determine whether a graph is acyclic. Another way to perform topological sorting is to assign the vertices topological numbers $N, N-1, \ldots, 1$ by postorder traversal of the depth-first spanning forest. As long as the graph is acyclic, this ordering will be consistent.

9.6.5 Finding Strong Components

By performing two depth-first searches, we can test whether a directed graph is strongly connected, and if it is not, we can actually produce the subsets of vertices that are strongly connected to themselves. This can also be done in only one depth-first search, but the method used here is much simpler to understand.

First, a depth-first search is performed on the input graph G. The vertices of G are numbered by a postorder traversal of the depth-first spanning forest, and then all edges in G are reversed, forming G_r. The graph in Figure 9.78 represents G_r for the graph G shown in Figure 9.76; the vertices are shown with their numbers.

The algorithm is completed by performing a depth-first search on G_r, always starting a new depth-first search at the highest-numbered vertex. Thus, we begin the depth-first search of G_r at vertex G, which is numbered 10. This leads nowhere, so the next search is started at H. This call visits I and J. The next call starts at B and visits A, C, and F. The next calls after this are `dfs(D)` and finally `dfs(E)`. The resulting depth-first spanning forest is shown in Figure 9.79.

Each of the trees (this is easier to see if you completely ignore all nontree edges) in this depth-first spanning forest forms a strongly connected component. Thus, for our example, the strongly connected components are $\{G\}$, $\{H, I, J\}$, $\{B, A, C, F\}$, $\{D\}$, and $\{E\}$.

To see why this algorithm works, first note that if two vertices v and w are in the same strongly connected component, then there are paths from v to w and from w to v in the original graph G, and hence also in G_r. Now, if two vertices v and w are not in the same depth-first spanning tree of G_r, clearly they cannot be in the same strongly connected component.

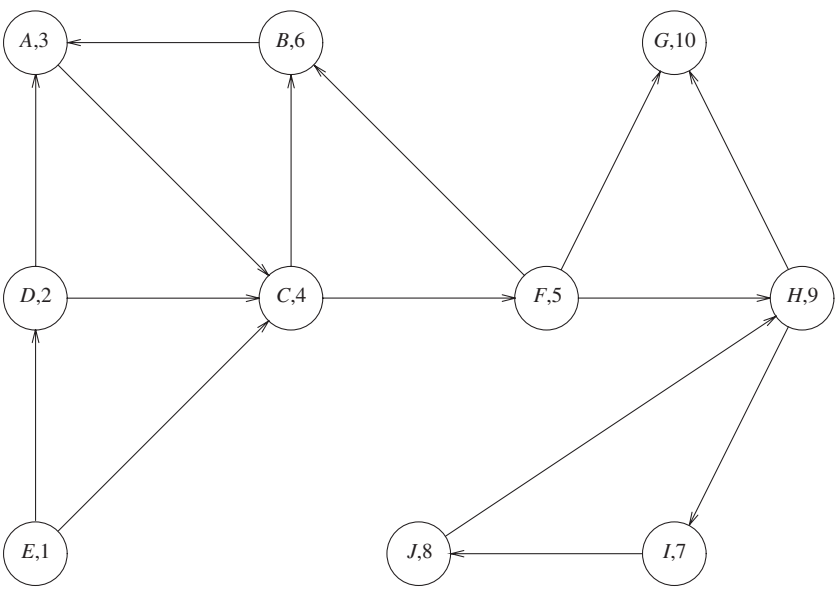

Figure 9.78 G_r numbered by postorder traversal of G (from Fig. 9.76)

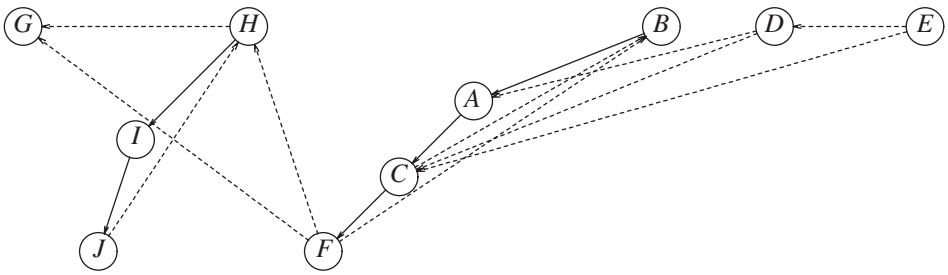

Figure 9.79 Depth-first search of G_r—strong components are {G}, {H, I, J}, {B, A, C, F}, {D}, {E}

To prove that this algorithm works, we must show that if two vertices v and w are in the same depth-first spanning tree of G_r, there must be paths from v to w and from w to v. Equivalently, we can show that if x is the root of the depth-first spanning tree of G_r containing v, then there is a path from x to v and from v to x. Applying the same logic to w would then give a path from x to w and from w to x. These paths would imply paths from v to w and w to v (going through x).

Since v is a descendant of x in G_r's depth-first spanning tree, there is a path from x to v in G_r and thus a path from v to x in G. Furthermore, since x is the root, x has the higher postorder number from the first depth-first search. Therefore, during the first depth-first search, all the work processing v was completed before the work at x was completed. Since there is a path from v to x, it follows that v must be a descendant of x in the spanning tree for G—otherwise v would finish *after* x. This implies a path from x to v in G and completes the proof.

9.7 Introduction to NP-Completeness

In this chapter, we have seen solutions to a wide variety of graph theory problems. All these problems have polynomial running times, and with the exception of the network flow problem, the running time is either linear or only slightly more than linear ($O(|E| \log |E|)$). We have also mentioned, in passing, that for some problems certain variations seem harder than the original.

Recall that the Euler circuit problem, which finds a path that touches every edge exactly once, is solvable in linear time. The Hamiltonian cycle problem asks for a simple cycle that contains every vertex. No linear algorithm is known for this problem.

The single-source unweighted shortest-path problem for directed graphs is also solvable in linear time. No linear-time algorithm is known for the corresponding longest-simple-path problem.

The situation for these problem variations is actually much worse than we have described. Not only are no linear algorithms known for these variations, but there are no known algorithms that are guaranteed to run in polynomial time. The best known algorithms for these problems could take exponential time on some inputs.

In this section we will take a brief look at this problem. This topic is rather complex, so we will only take a quick and informal look at it. Because of this, the discussion may be (necessarily) somewhat imprecise in places.

We will see that there are a host of important problems that are roughly equivalent in complexity. These problems form a class called the *NP-complete* problems. The exact complexity of these *NP*-complete problems has yet to be determined and remains the foremost open problem in theoretical computer science. Either all these problems have polynomial-time solutions or none of them do.

9.7.1 Easy vs. Hard

When classifying problems, the first step is to examine the boundaries. We have already seen that many problems can be solved in linear time. We have also seen some $O(\log N)$ running times, but these either assume some preprocessing (such as input already being read or a data structure already being built) or occur on arithmetic examples. For instance, the *gcd* algorithm, when applied on two numbers M and N, takes $O(\log N)$ time. Since the numbers consist of $\log M$ and $\log N$ bits, respectively, the *gcd* algorithm is really taking time that is linear in the *amount* or *size* of input. Thus, when we measure running time, we will be concerned with the running time as a function of the amount of input. Generally, we cannot expect better than linear running time.

At the other end of the spectrum lie some truly hard problems. These problems are so hard that they are *impossible*. This does not mean the typical exasperated moan, which means that it would take a genius to solve the problem. Just as real numbers are not sufficient to express a solution to $x^2 < 0$, one can prove that computers cannot solve every problem that happens to come along. These "impossible" problems are called **undecidable problems**.

One particular undecidable problem is the **halting problem.** Is it possible to have your C++ compiler have an extra feature that not only detects syntax errors but also all infinite loops? This seems like a hard problem, but one might expect that if some very clever programmers spent enough time on it, they could produce this enhancement.

The intuitive reason that this problem is undecidable is that such a program might have a hard time checking itself. For this reason, these problems are sometimes called *recursively undecidable*.

If an infinite loop–checking program could be written, surely it could be used to check itself. We could then produce a program called *LOOP*. *LOOP* takes as input a program, P, and runs P on itself. It prints out the phrase *YES* if P loops when run on itself. If P terminates when run on itself, a natural thing to do would be to print out *NO*. Instead of doing that, we will have *LOOP* go into an infinite loop.

What happens when *LOOP* is given itself as input? Either *LOOP* halts, or it does not halt. The problem is that both these possibilities lead to contradictions, in much the same way as does the phrase "This sentence is a lie."

By our definition, *LOOP(P)* goes into an infinite loop if *P(P)* terminates. Suppose that when $P = LOOP$, *P(P)* terminates. Then, according to the *LOOP* program, *LOOP(P)* is obligated to go into an infinite loop. Thus, we must have *LOOP(LOOP)* terminating *and* entering an infinite loop, which is clearly not possible. On the other hand, suppose that when $P = LOOP$, *P(P)* enters an infinite loop. Then *LOOP(P)* must terminate,

and we arrive at the same set of contradictions. Thus, we see that the program *LOOP* cannot possibly exist.

9.7.2 The Class NP

A few steps down from the horrors of undecidable problems lies the class *NP*. *NP* stands for **nondeterministic polynomial-time.** A deterministic machine, at each point in time, is executing an instruction. Depending on the instruction, it then goes to some next instruction, which is unique. A nondeterministic machine has a choice of next steps. It is free to choose any that it wishes, and if one of these steps leads to a solution, it will always choose the correct one. A nondeterministic machine thus has the power of extremely good (optimal) guessing. This probably seems like a ridiculous model, since nobody could possibly build a nondeterministic computer, and because it would seem to be an incredible upgrade to your standard computer (every problem might now seem trivial). We will see that nondeterminism is a very useful theoretical construct. Furthermore, nondeterminism is not as powerful as one might think. For instance, undecidable problems are still undecidable, even if nondeterminism is allowed.

A simple way to check if a problem is in *NP* is to phrase the problem as a yes/no question. The problem is in *NP* if, in polynomial time, we can prove that any "yes" instance is correct. We do not have to worry about "no" instances, since the program always makes the right choice. Thus, for the Hamiltonian cycle problem, a "yes" instance would be any simple circuit in the graph that includes all the vertices. This is in *NP*, since, given the path, it is a simple matter to check that it is really a Hamiltonian cycle. Appropriately phrased questions, such as "Is there a simple path of length $> K$?" can also easily be checked and are in *NP*. Any path that satisfies this property can be checked trivially.

The class *NP* includes all problems that have polynomial-time solutions, since obviously the solution provides a check. One would expect that since it is so much easier to check an answer than to come up with one from scratch, there would be problems in *NP* that do not have polynomial-time solutions. To date no such problem has been found, so it is entirely possible, though not considered likely by experts, that nondeterminism is not such an important improvement. The problem is that proving exponential lower bounds is an extremely difficult task. The information theory bound technique, which we used to show that sorting requires $\Omega(N \log N)$ comparisons, does not seem to be adequate for the task, because the decision trees are not nearly large enough.

Notice also that not all decidable problems are in *NP*. Consider the problem of determining whether a graph *does not* have a Hamiltonian cycle. To prove that a graph has a Hamiltonian cycle is a relatively simple matter—we just need to exhibit one. Nobody knows how to show, in polynomial time, that a graph does not have a Hamiltonian cycle. It seems that one must enumerate all the cycles and check them one by one. Thus the non–Hamiltonian cycle problem is not known to be in *NP*.

9.7.3 NP-Complete Problems

Among all the problems known to be in *NP*, there is a subset, known as the **NP-complete problems**, which contains the hardest. An *NP*-complete problem has the property that any problem in *NP* can be **polynomially reduced** to it.

A problem, P_1, can be reduced to P_2 as follows: Provide a mapping so that any instance of P_1 can be transformed to an instance of P_2. Solve P_2, and then map the answer back to the original. As an example, numbers are entered into a pocket calculator in decimal. The decimal numbers are converted to binary, and all calculations are performed in binary. Then the final answer is converted back to decimal for display. For P_1 to be polynomially reducible to P_2, all the work associated with the transformations must be performed in polynomial time.

The reason that *NP*-complete problems are the hardest *NP* problems is that a problem that is *NP*-complete can essentially be used as a subroutine for *any* problem in *NP*, with only a polynomial amount of overhead. Thus, if any *NP*-complete problem has a polynomial-time solution, then *every* problem in *NP* must have a polynomial-time solution. This makes the *NP*-complete problems the hardest of all *NP* problems.

Suppose we have an *NP*-complete problem, P_1. Suppose P_2 is known to be in *NP*. Suppose further that P_1 polynomially reduces to P_2, so that we can solve P_1 by using P_2 with only a polynomial time penalty. Since P_1 is *NP*-complete, every problem in *NP* polynomially reduces to P_1. By applying the closure property of polynomials, we see that every problem in *NP* is polynomially reducible to P_2: We reduce the problem to P_1 and then reduce P_1 to P_2. Thus, P_2 is *NP*-complete.

As an example, suppose that we already know that the Hamiltonian cycle problem is *NP*-complete. The **traveling salesman problem** is as follows.

Traveling Salesman Problem

Given a complete graph, $G = (V, E)$, with edge costs, and an integer K, is there a simple cycle that visits all vertices and has total cost $\leq K$?

The problem is different from the Hamiltonian cycle problem, because all $|V|(|V|-1)/2$ edges are present and the graph is weighted. This problem has many important applications. For instance, printed circuit boards need to have holes punched so that chips, resistors, and other electronic components can be placed. This is done mechanically. Punching the hole is a quick operation; the time-consuming step is positioning the hole puncher. The time required for positioning depends on the distance traveled from hole to hole. Since we would like to punch every hole (and then return to the start for the next board), and minimize the total amount of time spent traveling, what we have is a traveling salesman problem.

The traveling salesman problem is *NP*-complete. It is easy to see that a solution can be checked in polynomial time, so it is certainly in *NP*. To show that it is *NP*-complete, we polynomially reduce the Hamiltonian cycle problem to it. To do this we construct a new graph, G'. G' has the same vertices as G. For G', each edge (v, w) has a weight of 1 if $(v, w) \in G$, and 2 otherwise. We choose $K = |V|$. See Figure 9.80.

It is easy to verify that G has a Hamiltonian cycle if and only if G' has a traveling salesman tour of total weight $|V|$.

There is now a long list of problems known to be *NP*-complete. To prove that some new problem is *NP*-complete, it must be shown to be in *NP*, and then an appropriate *NP*-complete problem must be transformed into it. Although the transformation to a traveling salesman problem was rather straightforward, most transformations are actually quite involved and require some tricky constructions. Generally, several different *NP*-complete

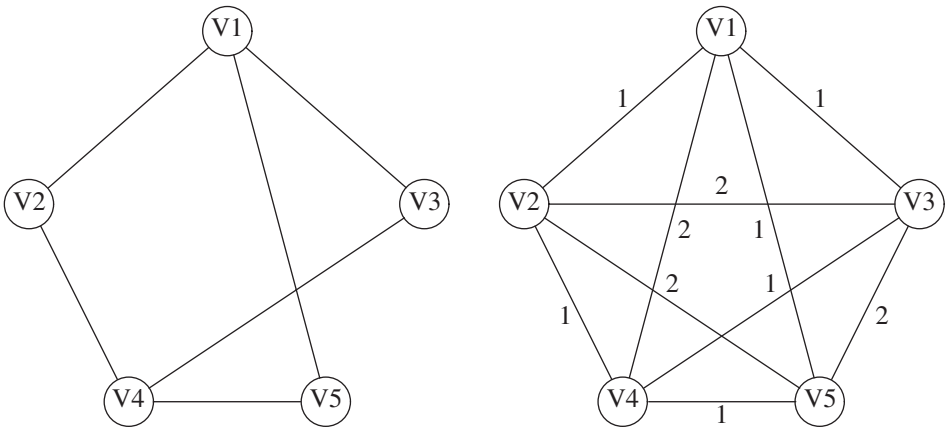

Figure 9.80 Hamiltonian cycle problem transformed to traveling salesman problem

problems are considered before the problem that actually provides the reduction. As we are only interested in the general ideas, we will not show any more transformations; the interested reader can consult the references.

The alert reader may be wondering how the first *NP*-complete problem was actually proven to be *NP*-complete. Since proving that a problem is *NP*-complete requires transforming it from another *NP*-complete problem, there must be some *NP*-complete problem for which this strategy will not work. The first problem that was proven to be *NP*-complete was the satisfiability problem. The **satisfiability problem** takes as input a Boolean expression and asks whether the expression has an assignment to the variables that gives a value of `true`.

Satisfiability is certainly in *NP*, since it is easy to evaluate a Boolean expression and check whether the result is true. In 1971, Cook showed that satisfiability was *NP*-complete by directly proving that all problems that are in *NP* could be transformed to satisfiability. To do this, he used the one known fact about every problem in *NP*: Every problem in *NP* can be solved in polynomial time by a nondeterministic computer. The formal model for a computer is known as a **Turing machine.** Cook showed how the actions of this machine could be simulated by an extremely complicated and long, but still polynomial, Boolean formula. This Boolean formula would be true if and only if the program which was being run by the Turing machine produced a "yes" answer for its input.

Once satisfiability was shown to be *NP*-complete, a host of new *NP*-complete problems, including some of the most classic problems, were also shown to be *NP*-complete.

In addition to the satisfiability, Hamiltonian circuit, traveling salesman, and longest-path problems, which we have already examined, some of the more well-known *NP*-complete problems which we have not discussed are *bin packing, knapsack, graph coloring,* and *clique*. The list is quite extensive and includes problems from operating systems (scheduling and security), database systems, operations research, logic, and especially graph theory.

Summary

In this chapter we have seen how graphs can be used to model many real-life problems. Many of the graphs that occur are typically very sparse, so it is important to pay attention to the data structures that are used to implement them.

We have also seen a class of problems that do not seem to have efficient solutions. In Chapter 10, some techniques for dealing with these problems will be discussed.

Exercises

9.1 Find a topological ordering for the graph in Figure 9.81.

9.2 If a stack is used instead of a queue for the topological sort algorithm in Section 9.2, does a different ordering result? Why might one data structure give a "better" answer?

9.3 Write a program to perform a topological sort on a graph.

9.4 An adjacency matrix requires $O(|V|^2)$ merely to initialize using a standard double loop. Propose a method that stores a graph in an adjacency matrix (so that testing for the existence of an edge is $O(1)$) but avoids the quadratic running time.

9.5 a. Find the shortest path from *A* to all other vertices for the graph in Figure 9.82.
b. Find the shortest unweighted path from *B* to all other vertices for the graph in Figure 9.82.

9.6 What is the worst-case running time of Dijkstra's algorithm when implemented with *d*-heaps (Section 6.5)?

9.7 a. Give an example where Dijkstra's algorithm gives the wrong answer in the presence of a negative edge but no negative-cost cycle.
** b. Show that the weighted shortest-path algorithm suggested in Section 9.3.3 works if there are negative-weight edges, but no negative-cost cycles, and that the running time of this algorithm is $O(|E| \cdot |V|)$.

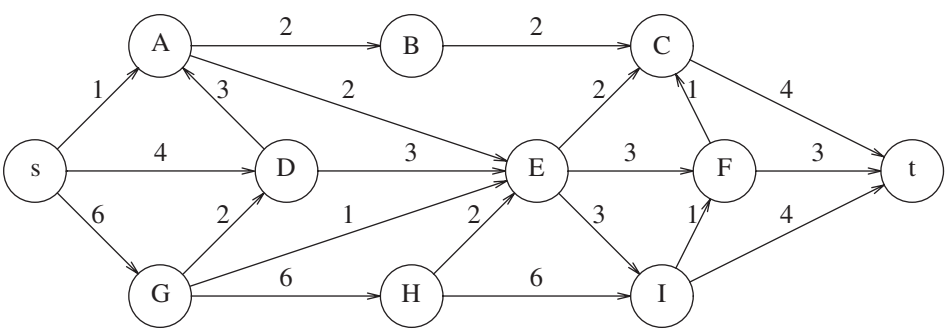

Figure 9.81 Graph used in Exercises 9.1 and 9.11

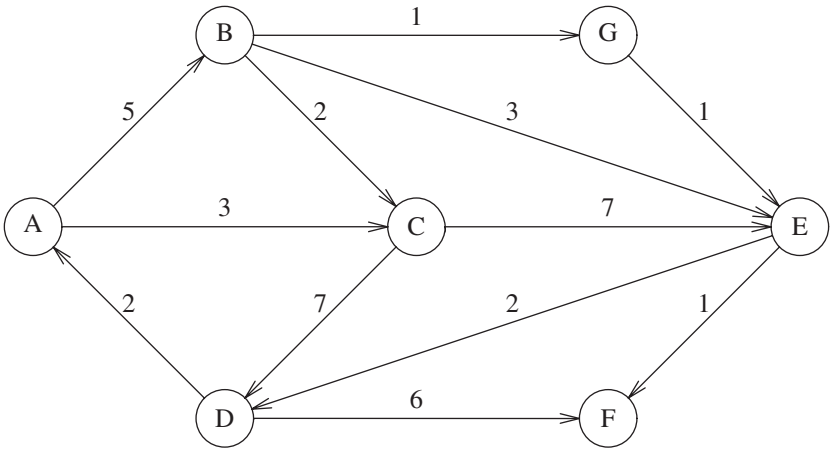

Figure 9.82 Graph used in Exercise 9.5

* **9.8** Suppose all the edge weights in a graph are integers between 1 and $|E|$. How fast can Dijkstra's algorithm be implemented?

9.9 Write a program to solve the single-source shortest-path problem.

9.10 a. Explain how to modify Dijkstra's algorithm to produce a count of the number of different minimum paths from v to w.

b. Explain how to modify Dijkstra's algorithm so that if there is more than one minimum path from v to w, a path with the fewest number of edges is chosen.

9.11 Find the maximum flow in the network of Figure 9.81.

9.12 Suppose that $G = (V, E)$ is a tree, s is the root, and we add a vertex t and edges of infinite capacity from all leaves in G to t. Give a linear-time algorithm to find a maximum flow from s to t.

9.13 A bipartite graph, $G = (V, E)$, is a graph such that V can be partitioned into two subsets, V_1 and V_2, and no edge has both its vertices in the same subset.

a. Give a linear algorithm to determine whether a graph is bipartite.

b. The bipartite matching problem is to find the largest subset E' of E such that no vertex is included in more than one edge. A matching of four edges (indicated by dashed edges) is shown in Figure 9.83. There is a matching of five edges, which is maximum.

Show how the bipartite matching problem can be used to solve the following problem: We have a set of instructors, a set of courses, and a list of courses that each instructor is qualified to teach. If no instructor is required to teach more than one course, and only one instructor may teach a given course, what is the maximum number of courses that can be offered?

c. Show that the network flow problem can be used to solve the bipartite matching problem.

d. What is the time complexity of your solution to part (b)?

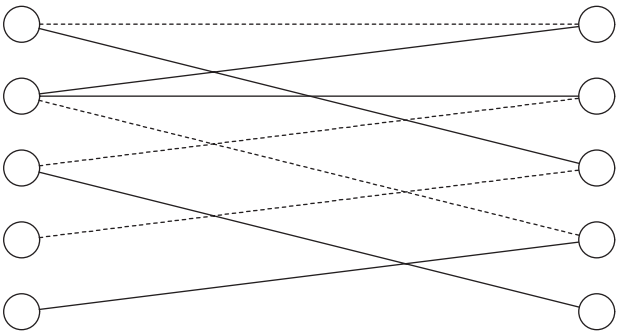

Figure 9.83 A bipartite graph

* **9.14** a. Give an algorithm to find an augmenting path that permits the maximum flow.
b. Let f be the amount of flow remaining in the residual graph. Show that the augmenting path produced by the algorithm in part (a) admits a path of capacity $f/|E|$.
c. Show that after $|E|$ consecutive iterations, the total flow remaining in the residual graph is reduced from f to at most f/e, where $e \approx 2.71828$.
d. Show that $|E| \ln f$ iterations suffice to produce the maximum flow.

9.15 a. Find a minimum spanning tree for the graph in Figure 9.84 using both Prim's and Kruskal's algorithms.
b. Is this minimum spanning tree unique? Why?

9.16 Does either Prim's or Kruskal's algorithm work if there are negative edge weights?

9.17 Show that a graph of V vertices can have V^{V-2} minimum spanning trees.

9.18 Write a program to implement Kruskal's algorithm.

9.19 If all of the edges in a graph have weights between 1 and $|E|$, how fast can the minimum spanning tree be computed?

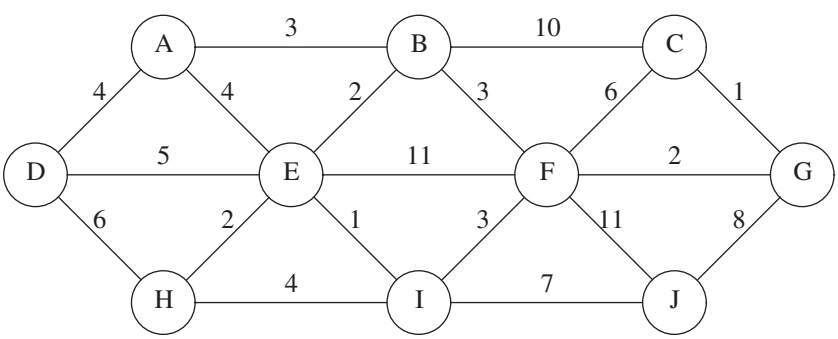

Figure 9.84 Graph used in Exercise 9.15

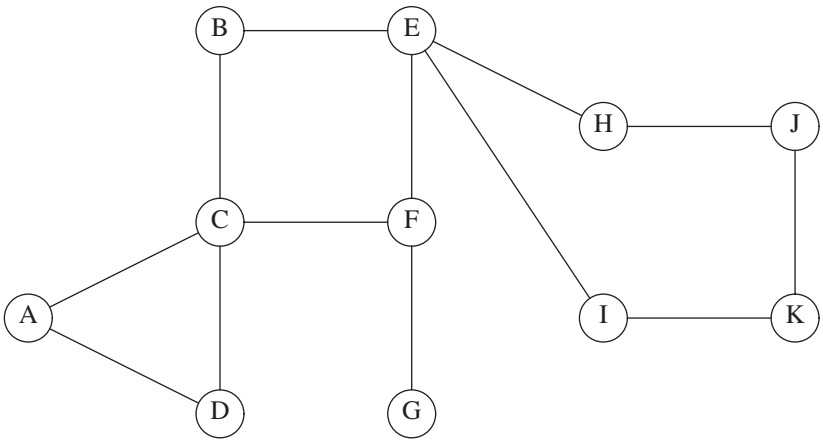

Figure 9.85 Graph used in Exercise 9.21

9.20 Give an algorithm to find a *maximum* spanning tree. Is this harder than finding a minimum spanning tree?

9.21 Find all the articulation points in the graph in Figure 9.85. Show the depth-first spanning tree and the values of *Num* and *Low* for each vertex.

9.22 Prove that the algorithm to find articulation points works.

9.23 a. Give an algorithm to find the minimum number of edges that need to be removed from an undirected graph so that the resulting graph is acyclic.

* b. Show that this problem is *NP*-complete for directed graphs.

9.24 Prove that in a depth-first spanning forest of a directed graph, all cross edges go from right to left.

9.25 Give an algorithm to decide whether an edge (v, w) in a depth-first spanning forest of a directed graph is a tree, back, cross, or forward edge.

9.26 Find the strongly connected components in the graph of Figure 9.86.

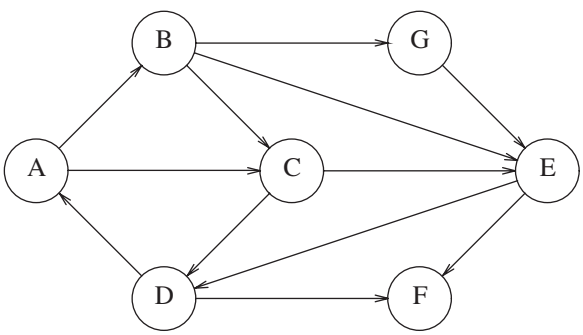

Figure 9.86 Graph used in Exercise 9.26

9.27 Write a program to find the strongly connected components in a digraph.

* **9.28** Give an algorithm that finds the strongly connected components in only one depth-first search. Use an algorithm similar to the biconnectivity algorithm.

9.29 The *biconnected components* of a graph, G, is a partition of the edges into sets such that the graph formed by each set of edges is biconnected. Modify the algorithm in Figure 9.69 to find the biconnected components instead of the articulation points.

9.30 Suppose we perform a breadth-first search of an undirected graph and build a breadth-first spanning tree. Show that all edges in the tree are either tree edges or cross edges.

9.31 Give an algorithm to find in an undirected (connected) graph a path that goes through every edge exactly once in each direction.

9.32 a. Write a program to find an Euler circuit in a graph if one exists.
b. Write a program to find an Euler tour in a graph if one exists.

9.33 An Euler circuit in a directed graph is a cycle in which every edge is visited exactly once.

* a. Prove that a directed graph has an Euler circuit if and only if it is strongly connected and every vertex has equal indegree and outdegree.

* b. Give a linear-time algorithm to find an Euler circuit in a directed graph where one exists.

9.34 a. Consider the following solution to the Euler circuit problem: Assume that the graph is biconnected. Perform a depth-first search, taking back edges only as a last resort. If the graph is not biconnected, apply the algorithm recursively on the biconnected components. Does this algorithm work?
b. Suppose that when taking back edges, we take the back edge to the nearest ancestor. Does the algorithm work?

9.35 A planar graph is a graph that can be drawn in a plane without any two edges intersecting.

* a. Show that neither of the graphs in Figure 9.87 is planar.
b. Show that in a planar graph, there must exist some vertex which is connected to no more than five nodes.

** c. Show that in a planar graph, $|E| \leq 3|V| - 6$.

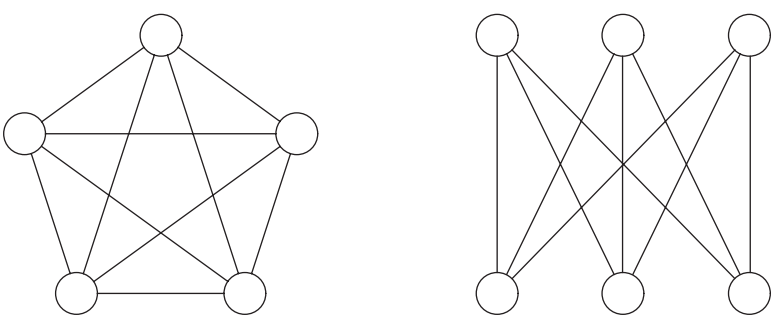

Figure 9.87 Graph used in Exercise 9.35

9.36 A *multigraph* is a graph in which multiple edges are allowed between pairs of vertices. Which of the algorithms in this chapter work without modification for multigraphs? What modifications need to be done for the others?

⋆ **9.37** Let $G = (V, E)$ be an undirected graph. Use depth-first search to design a linear algorithm to convert each edge in G to a directed edge such that the resulting graph is strongly connected, or determine that this is not possible.

9.38 You are given a set of N sticks, which are lying on top of each other in some configuration. Each stick is specified by its two endpoints; each endpoint is an ordered triple giving its x, y, and z coordinates; no stick is vertical. A stick may be picked up only if there is no stick on top of it.

a. Explain how to write a routine that takes two sticks a and b and reports whether a is above, below, or unrelated to b. (This has nothing to do with graph theory.)

b. Give an algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a sequence of stick pickups that accomplishes this.

9.39 A graph is k-colorable if each vertex can be given one of k colors, and no edge connects identically colored vertices. Give a linear-time algorithm to test a graph for two-colorability. Assume graphs are stored in adjacency-list format; you must specify any additional data structures that are needed.

9.40 Give a polynomial-time algorithm that finds $\lceil V/2 \rceil$ vertices that collectively cover at least three-fourths (3/4) of the edges in an arbitrary undirected graph.

9.41 Show how to modify the topological sort algorithm so that if the graph is not acyclic, the algorithm will print out some cycle. You may not use depth-first search.

9.42 Let G be a directed graph with N vertices. A vertex s is called a **sink** if, for every v in V such that $s \neq v$, there is an edge (v, s), and there are no edges of the form (s, v). Give an $O(N)$ algorithm to determine whether or not G has a sink, assuming that G is given by its $n \times n$ adjacency matrix.

9.43 When a vertex and its incident edges are removed from a tree, a collection of subtrees remains. Give a linear-time algorithm that finds a vertex whose removal from an N vertex tree leaves no subtree with more than $N/2$ vertices.

9.44 Give a linear-time algorithm to determine the longest unweighted path in an acyclic undirected graph (that is, a tree).

9.45 Consider an N-by-N grid in which some squares are occupied by black circles. Two squares belong to the same group if they share a common edge. In Figure 9.88, there is one group of four occupied squares, three groups of two occupied squares, and two individual occupied squares. Assume that the grid is represented by a two-dimensional array. Write a program that does the following:

a. Computes the size of a group when a square in the group is given.

b. Computes the number of different groups.

c. Lists all groups.

9.46 Section 8.7 described the generating of mazes. Suppose we want to output the path in the maze. Assume that the maze is represented as a matrix; each cell in the matrix stores information about what walls are present (or absent).

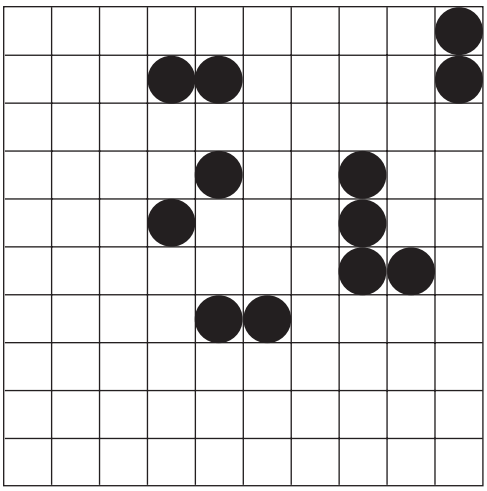

Figure 9.88 Grid for Exercise 9.45

a. Write a program that computes enough information to output a path in the maze. Give output in the form `SEN...` (representing go south, then east, then north, etc.).
b. If you are using a C++ compiler with a windowing package, write a program that draws the maze and, at the press of a button, draws the path.

9.47 Suppose that walls in the maze can be knocked down, with a penalty of P squares. P is specified as a parameter to the algorithm. (If the penalty is 0, then the problem is trivial.) Describe an algorithm to solve this version of the problem. What is the running time for your algorithm?

9.48 Suppose that the maze may or may not have a solution.
a. Describe a linear-time algorithm that determines the minimum number of walls that need to be knocked down to create a solution. *(Hint:* Use a double-ended queue.)
b. Describe an algorithm (not necessarily linear-time) that finds a shortest path after knocking down the minimum number of walls. Note that the solution to part (a) would give no information about which walls would be the best to knock down. (*Hint:* Use Exercise 9.47.)

9.49 Write a program to compute word ladders where single-character substitutions have a cost of 1, and single-character additions or deletions have a cost of $p > 0$, specified by the user. As mentioned at the end of Section 9.3.6, this is essentially a weighted shortest-path problem.

Explain how each of the following problems (Exercises 9.50–9.53) can be solved by applying a shortest-path algorithm. Then design a mechanism for representing an input, and write a program that solves the problem.

9.50 The input is a list of league game scores (and there are no ties). If all teams have at least one win and a loss, we can generally "prove," by a silly transitivity argument,

that any team is better than any other. For instance, in the six-team league where everyone plays three games, suppose we have the following results: A beat B and C; B beat C and F; C beat D; D beat E; E beat A; F beat D and E. Then we can prove that A is better than F, because A beat B, who in turn, beat F. Similarly, we can prove that F is better than A because F beat E and E beat A. Given a list of game scores and two teams X and Y, either find a proof (if one exists) that X is better than Y, or indicate that no proof of this form can be found.

9.51 The input is a collection of currencies and their exchange rates. Is there a sequence of exchanges that makes money instantly? For instance, if the currencies are X, Y, and Z and the exchange rate is 1 X equals 2 Ys, 1 Y equals 2 Zs, and 1 X equals 3 Zs, then 300 Zs will buy 100 Xs, which in turn will buy 200 Ys, which in turn will buy 400 Zs. We have thus made a profit of 33 percent.

9.52 A student needs to take a certain number of courses to graduate, and these courses have prerequisites that must be followed. Assume that all courses are offered every semester and that the student can take an unlimited number of courses. Given a list of courses and their prerequisites, compute a schedule that requires the minimum number of semesters.

9.53 The object of the *Kevin Bacon Game* is to link a movie actor to Kevin Bacon via shared movie roles. The minimum number of links is an actor's *Bacon number.* For instance, Tom Hanks has a Bacon number of 1; he was in *Apollo 13* with Kevin Bacon. Sally Fields has a Bacon number of 2, because she was in *Forrest Gump* with Tom Hanks, who was in *Apollo 13* with Kevin Bacon. Almost all well-known actors have a Bacon number of 1 or 2. Assume that you have a comprehensive list of actors, with roles,[3] and do the following:

a. Explain how to find an actor's Bacon number.
b. Explain how to find the actor with the highest Bacon number.
c. Explain how to find the minimum number of links between two arbitrary actors.

9.54 The *clique* problem can be stated as follows: Given an undirected graph, $G = (V, E)$, and an integer, K, does G contain a complete subgraph of at least K vertices?

The *vertex cover* problem can be stated as follows: Given an undirected graph, $G = (V, E)$, and an integer, K, does G contain a subset $V' \subset V$ such that $|V'| \leq K$ and every edge in G has a vertex in V'? Show that the clique problem is polynomially reducible to vertex cover.

9.55 Assume that the Hamiltonian cycle problem is *NP*-complete for undirected graphs.

a. Prove that the Hamiltonian cycle problem is *NP*-complete for directed graphs.
b. Prove that the unweighted simple longest-path problem is *NP*-complete for directed graphs.

9.56 The *baseball card collector* problem is as follows: Given packets $P_1, P_2, \ldots, P_M$, each of which contains a subset of the year's baseball cards, and an integer, K, is it possible to collect all the baseball cards by choosing $\leq K$ packets? Show that the baseball card collector problem is *NP*-complete.

[3] For instance, see the *Internet Movie Database* files: `actor.list.gz` and `actresses.list.gz` at `ftp://ftp.fu-berlin.de/pub/misc/movies/database`.

References

Good graph theory textbooks include [9], [14], [24], and [39]. More advanced topics, including the more careful attention to running times, are covered in [41], [44], and [51].

Use of adjacency lists was advocated in [26]. The topological sort algorithm is from [31], as described in [36]. Dijkstra's algorithm appeared in [10]. The improvements using *d*-heaps and Fibonacci heaps are described in [30] and [16], respectively. The shortest-path algorithm with negative edge weights is due to Bellman [3]; Tarjan [51] describes a more efficient way to guarantee termination.

Ford and Fulkerson's seminal work on network flow is [15]. The idea of augmenting along shortest paths or on paths admitting the largest flow increase is from [13]. Other approaches to the problem can be found in [11], [34], [23], [7], [35], [22], and [43]. An algorithm for the min-cost flow problem can be found in [20].

An early minimum spanning tree algorithm can be found in [4]. Prim's algorithm is from [45]; Kruskal's algorithm appears in [37]. Two $O(|E| \log \log |V|)$ algorithms are [6] and [52]. The theoretically best-known algorithms appear in [16], [18], [32] and [5]. An empirical study of these algorithms suggests that Prim's algorithm, implemented with `decreaseKey`, is best in practice on most graphs [42].

The algorithm for biconnectivity is from [47]. The first linear-time strong components algorithm (Exercise 9.28) appears in the same paper. The algorithm presented in the text is due to Kosaraju (unpublished) and Sharir [46]. Other applications of depth-first search appear in [27], [28], [48], and [49] (as mentioned in Chapter 8, the results in [48] and [49] have been improved, but the basic algorithm is unchanged).

The classic reference work for the theory of *NP*-complete problems is [21]. Additional material can be found in [1]. The *NP*-completeness of satisfiability is shown in [8] and independently by Levin. The other seminal paper is [33], which showed the *NP*-completeness of 21 problems. An excellent survey of complexity theory is [50]. An approximation algorithm for the traveling salesman problem, which generally gives nearly optimal results, can be found in [40].

A solution to Exercise 9.8 can be found in [2]. Solutions to the bipartite matching problem in Exercise 9.13 can be found in [25] and [38]. The problem can be generalized by adding weights to the edges and removing the restriction that the graph is bipartite. Efficient solutions for the unweighted matching problem for general graphs are quite complex. Details can be found in [12], [17], and [19].

Exercise 9.35 deals with planar graphs, which commonly arise in practice. Planar graphs are very sparse, and many difficult problems are easier on planar graphs. An example is the graph isomorphism problem, which is solvable in linear time for planar graphs [29]. No polynomial time algorithm is known for general graphs.

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.
2. R. K. Ahuja, K. Melhorn, J. B. Orlin, and R. E. Tarjan, "Faster Algorithms for the Shortest Path Problem," *Journal of the ACM,* 37 (1990), 213–223.
3. R. E. Bellman, "On a Routing Problem," *Quarterly of Applied Mathematics,* 16 (1958), 87–90.
4. O. Borůvka, "Ojistém problému minimálním (On a Minimal Problem)," *Práca Moravské Přirodo-vědecké Společnosti,* 3 (1926), 37–58.

5. B. Chazelle, "A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity," *Journal of the ACM*, 47 (2000), 1028–1047.
6. D. Cheriton and R. E. Tarjan, "Finding Minimum Spanning Trees," *SIAM Journal on Computing,* 5 (1976), 724–742.
7. J. Cheriyan and T. Hagerup, "A Randomized Maximum-Flow Algorithm," *SIAM Journal on Computing,* 24 (1995), 203–226.
8. S. Cook, "The Complexity of Theorem Proving Procedures," *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1971), 151–158.
9. N. Deo, *Graph Theory with Applications to Engineering and Computer Science,* Prentice Hall, Englewood Cliffs, N.J., 1974.
10. E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik,* 1 (1959), 269–271.
11. E. A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation," *Soviet Mathematics Doklady,* 11 (1970), 1277–1280.
12. J. Edmonds, "Paths, Trees, and Flowers," *Canadian Journal of Mathematics,* 17 (1965), 449–467.
13. J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *Journal of the ACM,* 19 (1972), 248–264.
14. S. Even, *Graph Algorithms,* Computer Science Press, Potomac, Md., 1979.
15. L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks,* Princeton University Press, Princeton, N.J., 1962.
16. M. L. Fredman and R. E. Tarjan, "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms," *Journal of the ACM,* 34 (1987), 596–615.
17. H. N. Gabow, "Data Structures for Weighted Matching and Nearest Common Ancestors with Linking," *Proceedings of First Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), 434–443.
18. H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, "Efficient Algorithms for Finding Minimum Spanning Trees on Directed and Undirected Graphs," *Combinatorica,* 6 (1986), 109–122.
19. Z. Galil, "Efficient Algorithms for Finding Maximum Matchings in Graphs," *ACM Computing Surveys,* 18 (1986), 23–38.
20. Z. Galil and E. Tardos, "An $O(n^2(m + n \log n) \log n)$ Min-Cost Flow Algorithm," *Journal of the ACM,* 35 (1988), 374–386.
21. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, San Francisco, 1979.
22. A. V. Goldberg and S. Rao, "Beyond the Flow Decomposition Barrier," *Journal of the ACM*, 45 (1998), 783–797.
23. A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum-Flow Problem," *Journal of the ACM,* 35 (1988), 921–940.
24. F. Harary, *Graph Theory,* Addison-Wesley, Reading, Mass., 1969.
25. J. E. Hopcroft and R. M. Karp, "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs," *SIAM Journal on Computing,* 2 (1973), 225–231.
26. J. E. Hopcroft and R. E. Tarjan, "Algorithm 447: Efficient Algorithms for Graph Manipulation," *Communications of the ACM,* 16 (1973), 372–378.

27. J. E. Hopcroft and R. E. Tarjan, "Dividing a Graph into Triconnected Components," *SIAM Journal on Computing,* 2 (1973), 135–158.
28. J. E. Hopcroft and R. E. Tarjan, "Efficient Planarity Testing," *Journal of the ACM,* 21 (1974), 549–568.
29. J. E. Hopcroft and J. K. Wong, "Linear-Time Algorithm for Isomorphism of Planar Graphs," *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing* (1974), 172–184.
30. D. B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM,* 24 (1977), 1–13.
31. A. B. Kahn, "Topological Sorting of Large Networks," *Communications of the ACM,* 5 (1962), 558–562.
32. D. R. Karger, P. N. Klein, and R. E. Tarjan, "A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees," *Journal of the ACM,* 42 (1995), 321–328.
33. R. M. Karp, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations* (eds. R. E. Miller and J. W. Thatcher), Plenum Press, New York, 1972, 85–103.
34. A. V. Karzanov, "Determining the Maximal Flow in a Network by the Method of Preflows," *Soviet Mathematics Doklady,* 15 (1974), 434–437.
35. V. King, S. Rao, and R. E. Tarjan, "A Faster Deterministic Maximum Flow Algorithm," *Journal of Algorithms,* 17 (1994), 447–474.
36. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* 3d ed., Addison-Wesley, Reading, Mass., 1997.
37. J. B. Kruskal, Jr., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society,* 7 (1956), 48–50.
38. H. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly,* 2 (1955), 83–97.
39. E. L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Holt, Reinhart and Winston, New York, 1976.
40. S. Lin and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Operations Research,* 21 (1973), 498–516.
41. K. Melhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-completeness,* Springer-Verlag, Berlin, 1984.
42. B. M. E. Moret and H. D. Shapiro, "An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree," *Proceedings of the Second Workshop on Algorithms and Data Structures* (1991), 400–411.
43. J. B. Orlin, "Max Flows in O(nm) Time, or Better," *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing* (2013).
44. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity,* Prentice Hall, Englewood Cliffs, N.J., 1982.
45. R. C. Prim, "Shortest Connection Networks and Some Generalizations," *Bell System Technical Journal,* 36 (1957), 1389–1401.
46. M. Sharir, "A Strong-Connectivity Algorithm and Its Application in Data Flow Analysis," *Computers and Mathematics with Applications,* 7 (1981), 67–72.
47. R. E. Tarjan, "Depth First Search and Linear Graph Algorithms," *SIAM Journal on Computing,* 1 (1972), 146–160.

48. R. E. Tarjan, "Testing Flow Graph Reducibility," *Journal of Computer and System Sciences,* 9 (1974), 355–365.
49. R. E. Tarjan, "Finding Dominators in Directed Graphs," *SIAM Journal on Computing,* 3 (1974), 62–89.
50. R. E. Tarjan, "Complexity of Combinatorial Algorithms," *SIAM Review,* 20 (1978), 457–491.
51. R. E. Tarjan, *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, 1983.
52. A. C. Yao, "An $O(|E| \log \log |V|)$ Algorithm for Finding Minimum Spanning Trees," *Information Processing Letters,* 4 (1975), 21–23.