classes to be generated may be regarded as sets. These sets are disjoint as no variable can be in more than one equivalence class. To begin with all n variables are in equivalence classes of their own; thus PARENT(i) = -1, $1 \leq i \leq n$. If an equivalence pair, $i \equiv j$, is to be processed, we must first determine the sets containing i and j. If these are different, then the two sets are to be replaced by their union. If the two sets are the same, t
nothing is to be done as the relation $i \equiv j$ is redundar a
in the same equivalence class. To process each equi we
to perform at most two finds and one union. Thus, if we have n varia
and $m \geq n$ equivalence pairs, the total processing time is at most $O(m\alpha(m, n))$. The major advantage of this algorithm is that it works "on-line." This means that at any time it can answer questions about the equivalence class of an element rather than require all pairs to be presented to it first. In the following chapters we will see other fruitful uses of these two set manipulation algorithms.

2.5 GRAPHS

Now we consider the data object graph, an important structure which was first introduced by the mathematician L. Euler in 1736. A *graph* G consists of two sets called the *vertices* V and the *edges* E. V is a finite non-empty set of vertices (sometimes called nodes) usually numbered 1, 2, . . . , n and E is a finite set of pairs of vertices. Each pair in E is an edge of G.

If the pairs are ordered (i.e. the pair $\langle i, j \rangle$ is different than the pair $\langle j, i \rangle$ then we call the graph *directed*. Otherwise we call it *undirected*. We will use angle brackets to denote directed edges and parentheses to denote undirected edges. Thus, $\langle i, j \rangle$ represents a directed edge while (i, j) represents an undirected edge. Note that edges of the type $\langle i,i \rangle$ or (i,i) are not permitted. For many applications there is often a positive real number, called a cost, which is attached to each edge. Such a graph is called a *network*.

In an undirected graph we say that the vertex i is *adjacent* to vertex j if the edge (i, j) exists. The degree of a vertex is the number of its adjacent vertices. For directed graphs we distinguish between the *in-degree* of a vertex i which is the number of edges with i as its second component, and the *out-degree* of i, the number of edges with i as the first component. If the directed edge $\langle i,j \rangle$ is present, then i is *adjacent-to* j and j is *adjacent-from* i.

A *path* from vertex v_p to v_q is a sequence of vertices v_p, v_{i1}, v_{i2}, . . . , v_{in}, v_q such that (v_p, v_{i1}), (v_{i1}, v_{i2}), . . . , (v_{in}, v_q) are edges in E(G). The *length* of a path is the number of edges on it. A *simple path* is a path in

which all vertices except possibly the first and last are 95/643 cle simple path in which the first and last vertices are the s.

In Figure 2.22 we have an example of a directed and an undirected graph both containing 5 vertices and 5 edges. In the directed graph vertex 1 has zero as its in-degree and three as its out-degree. The degree of vertex 1 in the undirected graph is three. In the undirected graph there is a path between every pair of vertices, whereas in the directed graph there is no way to go from vertex 3 (or vertex 5) to any other vertex. In Figure 2.22 (ii) the edges (1,2) (2,3) form a simple path and the path (1,2) (2,3) (3,1) is a cycle.

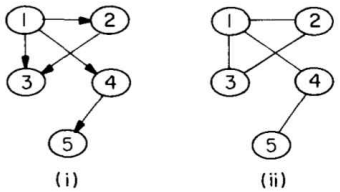


Figure 2.22 Two sample graphs

The last notion we will define before discussing representations of graphs is connectedness. An undirected graph is called *connected* if for every pair of vertices there exists a path between them. If a graph is not connected then we refer to its connected subgraphs separately. A subgraph of a graph is a subset of the vertices in V say VB, and a subset of the edges of E which connect vertices in VB. A subgraph $G' = (V', E')$ is a connected component of the undirected graph $G = (V, E)$ iff G' is connected and there exists no other subgraph $G'' = (V'', E'')$ of G which is also connected and either $V' \subset V''$ or $E' \subset E''$. I.e., a connected component is a maximal connected subgraph. For directed graphs the connectedness idea is strengthened. If for every pair of vertices, i, j there exists a path from i to j and a path from j to i then we say that directed graph is *strongly connected*.

There are two common ways to represent graphs. These may be thought of as the sequential and linked representations. The sequential form uses a square table with n rows and columns where n is the number of vertices. This table is called the *adjacency matrix*. For an undirected graph, the adjacency matrix, GRAPH(1:n, 1:n), is defined such that GRAPH(i, j) = 1 if the edge (i, j) is present and 0 otherwise. If the graph is a network then GRAPH(i, j) = the cost of edge (i, j). If (i, j) is not present the value of GRAPH(i, j) is $+\infty$. For a directed graph, GRAPH(i, j) = 1 iff $\langle i, j \rangle$ is an edge. Graph (i, j) is similarly defined in case of a directed network.

Table 2.3 shows the adjacency matrices for the directed and undirected graphs of Figure 2.22. Both matrices are 5 × 5 and have entries which are zero or one. Note how in both cases the diagonal elements are zero indicating no "self-edges." The second matrix has a special structure which all undirected graphs will have, and that is that GRAPH(i, j) = GRAPH(j, i). Such a matrix is said to be *symmetric*. Though the adjacency matrix normally requires n^2 locations, for undirected graphs it would suffice to keep only an upper triangular matrix, or $n(n-1)/2$ elements. Note that the main diagonal need not be stored as GRAPH(i, i) = 0.

	1	2	3	4	5
1)	0	1	1	1	0
2)	0	0	1	0	0
3)	0	0	0	0	0
4)	0	0	0	0	1
5)	0	0	0	0	0

	1	2	3	4	5
	0	1	1	1	0
	1	0	1	0	0
	1	1	0	0	0
	1	0	0	0	1
	0	0	0	1	0

Table 2.3 Adjacency matrices for Figure 2.22

Before beginning any computation on a graph we will normally have to initialize an adjacency matrix so that it contains the graph we are going to operate on. This step will typically require at least $O(n^2)$ operations. Thus, the computing time of most any algorithm using this form of representation will be *at least* $O(n^2)$. This will be true even if the graph has only $O(n)$ edges! This fact leads us to consider an alternative representation.

Given a graph, its *adjacency list* representation consists of n lists, one for each vertex i. The list for vertex i contains just those vertices adjacent from i. Because we often need to access the adjacent vertices of a random vertex we insist that the heads of the lists are stored sequentially. But the list of a vertex's neighbors may be linked together. Figure 2.23 shows the adjacency lists for the two graphs of Figure 2.22.

For both graphs there are five sequential locations (head nodes) whose values are either zero (if no neighbors exist) or a pointer to a list of vertices. Each node on the list has two fields, a vertex and a pointer to the next element on the list. The directed graph has 5 nodes and the undirected graph has 10. In general, a directed graph with n vertices and e edges will require n locations plus e nodes while an undirected graph will require n locations plus $2e$ nodes. This can be quite a bit better than the requirements of the adjacency matrix representation.

In case no insertion or deletion of edges or vertices are to be performed on the graph, the adjacency lists may themselves be represented sequentially

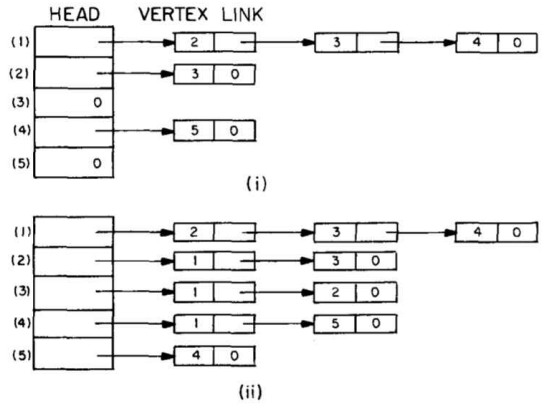


Figure 2.23 Adjacency lists for Figure 2.22

in a one dimensional array VERTEX(1:p) where $p = e$ if the graph is directed and $p = 2e$ if the graph is undirected. HEAD(i), $1 \leq i \leq n$ gives the starting point for the adjacency list for vertex i. If we define HEAD($n + 1$) = $p + 1$ then the vertices on the adjacency list for vertex i are stored in VERTEX(j), where HEAD(i) $\leq j <$ HEAD($i + 1$). If the list for vertex i is empty, then HEAD(i) = HEAD($i + 1$). Figure 2.24 gives the sequential adjacency list representations corresponding to the linked representations of Figure 2.23.

This concludes section 2.5. In the following chapters we will encounter many algorithms on graphs, so make sure that you are familiar with these representation schemes.

2.6 HASHING

A *symbol table* is a data structure which allows one to easily determine the presence or absence of an arbitrary element. It also permits easy insertion and deletion of elements. In this section we present what is undoubtedly the most practical technique for maintaining a symbol table, hashing. Though many of the tree organizations of symbol tables (e.g. binary search trees) are useful when special information about the identifiers is known, in the absence of a priori statistical information, hashing is both conceptually simple and, as we shall see, very efficient.

$$\begin{array}{r} \text{S E N D} \\ +\ \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$

Two conditions are assumed: first, the correspondence between letters and decimal digits is one-to-one, i.e., each letter represents one digit only and different letters represent different digits. Second, the digit zero does not appear as the left-most digit in any of the numbers. To solve an alphametic means to find which digit each letter represents. Note that a solution's uniqueness cannot be assumed and has to be verified by the solver.

a. Write a program for solving cryptarithms by exhaustive search. Assume that a given cryptarithm is a sum of two words.

b. Solve Dudeney's puzzle the way it was expected to be solved when it was first published in 1924.

5 Depth-First Search and Breadth-First Search

The term "exhaustive search" can also be applied to two very important algorithms that systematically process all vertices and edges of a graph. These two traversal algorithms are ***depth-first search (DFS)*** and ***breadth-first search (BFS)***. These algorithms have proved to be very useful for many applications involving graphs in artificial intelligence and operations research. In addition, they are indispensable for efficient investigation of fundamental properties of graphs such as connectivity and cycle presence.

Depth-First Search

Depth-first search starts a graph's traversal at an arbitrary vertex by marking it as visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. (If there are several such vertices, a tie can be resolved arbitrarily. As a practical matter, which of the adjacent unvisited candidates is chosen is dictated by the data structure representing the graph. In our examples, we always break ties by the alphabetical order of the vertices.) This process continues until a dead end—a vertex with no adjacent unvisited vertices—is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth-first search must be restarted at any one of them.

It is convenient to use a stack to trace the operation of depth-first search. We push a vertex onto the stack when the vertex is reached for the first time (i.e., the

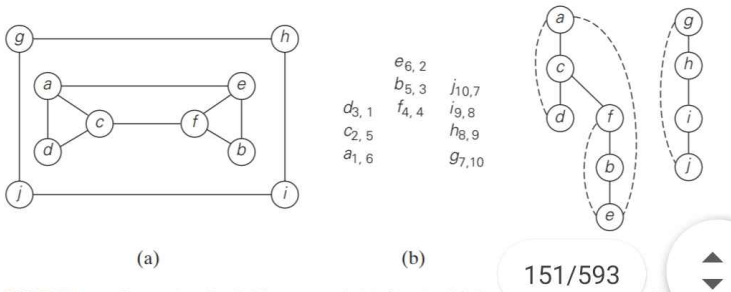


FIGURE 3.10 Example of a DFS traversal. (a) Graph. (b) Trave…e firs… subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

visit of the vertex starts), and we pop a vertex off the stack when it becomes a dead end (i.e., the visit of the vertex ends).

It is also very useful to accompany a depth-first search traversal by constructing the so-called ***depth-first search forest***. The starting vertex of the traversal serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a ***tree edge*** because the set of all such edges forms a forest. The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree). Such an edge is called a ***back edge*** because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest. Figure 3.10 provides an example of a depth-first search traversal, with the traversal stack and corresponding depth-first search forest shown as well.

Here is pseudocode of the depth-first search.

ALGORITHM *DFS*(G)

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)
```

```
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable count
count ← count + 1;   mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

The brevity of the DFS pseudocode and the ease with which it can be performed by hand may create a wrong impression about the level of sophistication of this algorithm. To appreciate its true power and depth, you should trace the algorithm's action by looking not at a graph's diagram but at its adjacency matrix or adjacency lists. (Try it for the graph in Figure 3.10 or a smaller example.)

How efficient is depth-first search? It is not difficult to see that this algorithm is, in fact, quite efficient since it takes just the time proportional to the size of the data structure used for representing the graph in question. Thus, for the adjacency matrix representation, the traversal time is in $\Theta(|V|^2)$, and for the adjacency list representation, it is in $\Theta(|V| + |E|)$ where $|V|$ and $|E|$ are the number of the graph's vertices and edges, respectively.

A DFS forest, which is obtained as a by-product of a DFS traversal, deserves a few comments, too. To begin with, it is not actually a forest. Rather, we can look at it as the given graph with its edges classified by the DFS traversal into two disjoint classes: tree edges and back edges. (No other types are possible for a DFS forest of an undirected graph.) Again, tree edges are edges used by the DFS traversal to reach previously unvisited vertices. If we consider only the edges in this class, we will indeed get a forest. Back edges connect vertices to previously visited vertices other than their immediate predecessors in the traversal. They connect vertices to their ancestors in the forest other than their parents.

A DFS traversal itself and the forest-like representation of the graph it provides have proved to be extremely helpful for the development of efficient algorithms for checking many important properties of graphs.[3] Note that the DFS yields two orderings of vertices: the order in which the vertices are reached for the first time (pushed onto the stack) and the order in which the vertices become dead ends (popped off the stack). These orders are qualitatively different, and various applications can take advantage of either of them.

Important elementary applications of DFS include checking connectivity and checking acyclicity of a graph. Since *dfs* halts after visiting all the vertices con-

3. The discovery of several such applications was an important breakthrough achieved by the two American computer scientists John Hopcroft and Robert Tarjan in the 1970s. For this and other contributions, they were given the Turing Award—the most prestigious prize in the computing field [Hop87, Tar87].

nected by a path to the starting vertex, checking a graph's connectivity can be done as follows. Start a DFS traversal at an arbitrary vertex and check, after the algorithm halts, whether all the vertices of the graph will have been visited. If they have, the graph is connected; otherwise, it is not connected. More generally, we can use DFS for identifying connected components of a graph (how?).

As for checking for a cycle presence in a graph, we can take advantage of the graph's representation in the form of a DFS forest. If the latter does not have back edges, the graph is clearly acyclic. If there is a back edge from some vertex u to its ancestor v (e.g., the back edge from d to a in Figure 3.10c), the graph has a cycle that comprises the path from v to u via a sequence of tree edges in the DFS forest followed by the back edge from u to v.

You will find a few other applications of DFS later in the book, although more sophisticated applications, such as finding articulation points of a graph, are not included. (A vertex of a connected graph is said to be its ***articulation point*** if its removal with all edges incident to it breaks the graph into disjoint pieces.)

Breadth-First Search

If depth-first search is a traversal for the brave (the algorithm goes as far from "home" as it can), breadth-first search is a traversal for the cautious. It proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

It is convenient to use a queue (note the difference from depth-first search!) to trace the operation of breadth-first search. The queue is initialized with the traversal's starting vertex, which is marked as visited. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

Similarly to a DFS traversal, it is useful to accompany a BFS traversal by constructing the so-called ***breadth-first search forest***. The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a ***tree edge***. If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a ***cross edge***. Figure 3.11 provides an example of a breadth-first search traversal, with the traversal queue and corresponding breadth-first search forest shown.

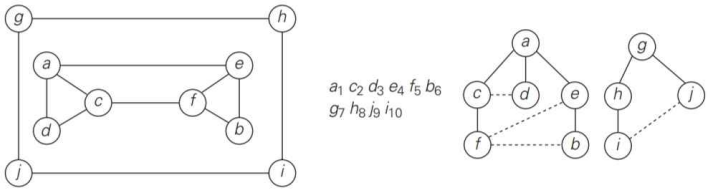


FIGURE 3.11 Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

Here is pseudocode of the breadth-first search.

ALGORITHM *BFS*(G)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable count
count ← count + 1;  mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;  mark w with count
            add w to the queue
    remove the front vertex from the queue
```

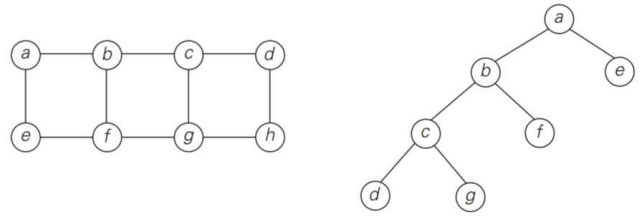


FIGURE 3.12 Illustration of the BFS-based algorithm for finding a minimum-edge path. (a) Graph. (b) Part of its BFS tree that identifies the minimum-edge path from a to g.

Breadth-first search has the same efficiency as depth-first search: it is in $\Theta(|V|^2)$ for the adjacency matrix representation and in $\Theta(|V| + |E|)$ for the adjacency list representation. Unlike depth-first search, it yields a single ordering of vertices because the queue is a FIFO (first-in first-out) structure and hence the order in which vertices are added to the queue is the same order in which they are removed from it. As to the structure of a BFS forest of an undirected graph, it can also have two kinds of edges: tree edges and cross edges. Tree edges are the ones used to reach previously unvisited vertices. Cross edges connect vertices to those visited before, but, unlike back edges in a DFS tree, they connect vertices either on the same or adjacent levels of a BFS tree.

BFS can be used to check connectivity and acyclicity of a graph, essentially in the same manner as DFS can. It is not applicable, however, for several less straightforward applications such as finding articulation points. On the other hand, it can be helpful in some situations where DFS cannot. For example, BFS can be used for finding a path with the fewest number of edges between two given vertices. To do this, we start a BFS traversal at one of the two vertices and stop it as soon as the other vertex is reached. The simple path from the root of the BFS tree to the second vertex is the path sought. For example, path $a - b - c - g$ in the graph in Figure 3.12 has the fewest number of edges among all the paths between vertices a and g. Although the correctness of this application appears to stem immediately from the way BFS operates, a mathematical proof of its validity is not quite elementary (see, e.g., [Cor09, Section 22.2]).

Table 3.1 summarizes the main facts about depth-first search and breadth-first search.

TABLE 3.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	BFS

What is the time efficiency of this algorithm? How is it compared to that of the version given in Section 4.1?

11. Let $A[0..n-1]$ be an array of n sortable elements. (For simplicity, you may assume that all the elements are distinct.) A pair $(A[i], A[j])$ is called an ***inversion*** if $i < j$ and $A[i] > A[j]$.

a. What arrays of size n have the largest number of inversions and what is this number? Answer the same questions for the smallest number of inversions.

b. Show that the average-case number of key comparisons in insertion sort is given by the formula

$$C_{avg}(n) \approx \frac{n^2}{4}.$$

12. Shellsort (more accurately Shell's sort) is an important sorting algorithm that works by applying insertion sort to each of several interleaving sublists of a given list. On each pass through the list, the sublists in question are formed by stepping through the list with an increment h_i taken from some predefined decreasing sequence of step sizes, $h_1 > \cdots > h_i > \cdots > 1$, which must end with 1. (The algorithm works for any such sequence, though some sequences are known to yield a better efficiency than others. For example, the sequence 1, 4, 13, 40, 121, . . . , used, of course, in reverse, is known to be among the best for this purpose.)

a. Apply shellsort to the list

$$S, H, E, L, L, S, O, R, T, I, S, U, S, E, F, U, L$$

b. Is shellsort a stable sorting algorithm?

c. Implement shellsort, straight insertion sort, selection sort, and bubble sort in the language of your choice and compare their performance on random arrays of sizes 10^n for $n = 2, 3, 4, 5$, and 6 as well as on increasing and decreasing arrays of these sizes.

.2 Topological Sorting

In this section, we discuss an important problem for directed graphs, with a variety of applications involving prerequisite-restricted tasks. Before we pose this problem, though, let us review a few basic facts about directed graphs themselves. A ***directed graph***, or ***digraph*** for short, is a graph with directions specified for all its edges (Figure 4.5a is an example). The adjacency matrix and adjacency lists are still two principal means of representing a digraph. There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.

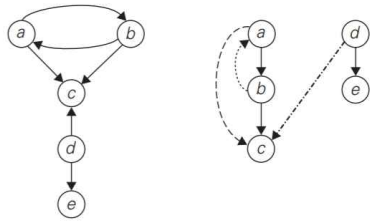


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example of Figure 4.5a, the depth-first search forest (Figure 4.5b) exhibits all four types of edges possible in a DFS forest of a directed graph: ***tree edges*** (ab, bc, de), ***back edges*** (ba) from vertices to their ancestors, ***forward edges*** (ac) from vertices to their descendants in the tree other than their children, and ***cross edges*** (dc), which are none of the aforementioned types.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, a, b, a is a directed cycle in the digraph in Figure 4.5a. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

Edge directions lead to new questions about digraphs that are either meaningless or trivial for undirected graphs. In this section, we discuss one such question. As a motivating example, consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements (Figure 4.6). In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. (Can you find such an ordering of this digraph's vertices?) This problem is called ***topological sorting***. It can be posed for an

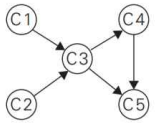


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

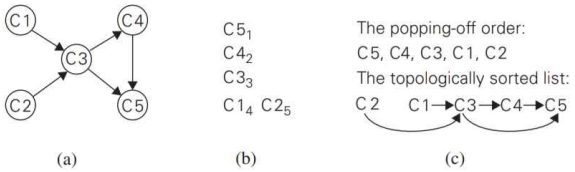


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

arbitrary digraph, but it is easy to see that the problem cannot have a solution if a digraph has a directed cycle. Thus, for topological sorting to be possible, a digraph in question must be a dag. It turns out that being a dag is not only necessary but also sufficient for topological sorting to be possible; i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution. Moreover, there are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.

The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

Why does the algorithm work? When a vertex v is popped off a DFS stack, no vertex u with an edge from u to v can be among the vertices popped off before v. (Otherwise, (u, v) would have been a back edge.) Hence, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

Figure 4.7 illustrates an application of this algorithm to the digraph in Figure 4.6. Note that in Figure 4.7c, we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a convenient way to check visually the correctness of a solution to an instance of the topological sorting problem.

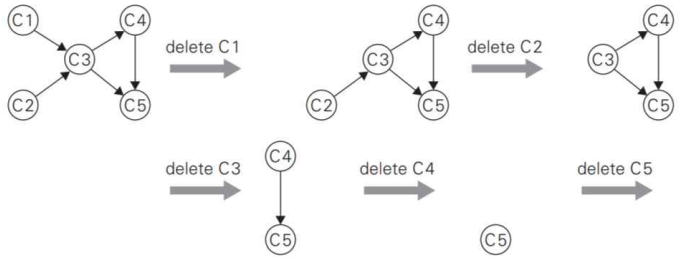


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved—see Problem 6a in this section's exercises.) The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure 4.8.

Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

The tiny size of the example we used might create a wrong impression about the topological sorting problem. But imagine a large project—e.g., in construction, research, or software development—that involves a multitude of interrelated tasks with known prerequisites. The first thing to do in such a situation is to make sure that the set of given prerequisites is not contradictory. The convenient way of doing this is to solve the topological sorting problem for the project's digraph. Only then can one start thinking about scheduling tasks to, say, minimize the total completion time of the project. This would require, of course, other algorithms that you can find in general books on operations research or in special ones on CPM (Critical Path Method) and PERT (Program Evaluation and Review Technique) methodologies.

As to applications of topological sorting in computer science, they include instruction scheduling in program compilation, cell evaluation ordering in spreadsheet formulas, and resolving symbol dependencies in linkers.

Exercises 4.2

$E(T')$ then T is clearly of minimum cost. If $E(T) \neq E(T')$ then let e be a minimum cost edge such that $e \in E(T)$ and $e \notin E(T')$. Clearly, such an e must exist. The inclusion of e into T' creates a unique cycle (Exercise 20). Let $e, e_1, e_2, \ldots, e_k$ be this unique cycle. At least one of the e_i's, $1 \leq i \leq k$ is not in $E(T)$ as otherwise T will also contain the cycle $e, e_1, e_2, \ldots, e_k$. Let e_j be an edge on this cycle such that $e_j \notin E(T)$. If e_j is of lesser cost than e then Kruskal's algorithm would consider e_j before e and include e_j into T. To see this note that all edges in $E(T)$ of cost less than the cost of e are also in $E(T')$ and do not form a cycle with e_j. So $c(e_j) \geq c(e)$ ($c(\cdot)$ is the edge-cost function).

Now, reconsider the graph with edge set $E(T') \cup \{e\}$. Removal of any edge on the cycle $e, e_1, e_2, \ldots, e_k$ will leave behind a tree T'' (Exercise 20). In particular, if we delete the edge e_j then the resulting tree T'' will have a cost no more than the cost of T' (as $c(e_j) \geq c(e)$). Hence, T'' is also a minimum cost tree.

By repeatedly using the transformation described above, tree T' can be transformed into the spanning tree T without any increase in cost. Hence, T is a minimum cost spanning tree. □

4.7 SINGLE SOURCE SHORTEST PATHS

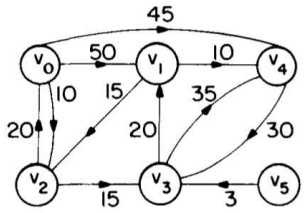
Graphs may be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges may then be assigned weights which might be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to city B would be interested in answers to the following questions:

(i) Is there a path from A to B?
(ii) If there is more than one path from A to B, which is the shortest path?

The problems defined by (i) and (ii) above are special cases of the path problem we shall be studying in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path will be referred to as the *source* and the last vertex the *destination*. The graphs will be digraphs to allow for one way streets. In the problem we shall consider, we are given a directed graph $G = (V, E)$, a weighting function $c(e)$ for the edges of G and a source vertex v_0. The

problem is to determine the shortest paths from v_0 to *all* the remaining vertices of G. It is assumed that all the weights are positive.

Example 4.11 Consider the directed graph of Figure 4.10(a). The numbers on the edges are the weights. If v_0 is the source vertex, then the shortest path from v_0 to v_1 is $v_0v_2v_3v_1$. The length of this path is 10 + 15 + 20 = 45. Even though there are three edges on this path, it is shorter than the path v_0v_1 which is of length 50. There is no path from v_0 to v_5. Figure 4.10(b) lists the shortest paths from v_0 to v_1, v_2, v_3 and v_4. The paths have been listed in nondecreasing order of path length. □



(a)

	Path	Length
1)	v_0v_2	10
2)	$v_0v_2v_3$	25
3)	$v_0v_2v_3v_1$	45
4)	v_0v_4	45

(b)

Figure 4.10 Graph and shortest paths from v_0 to all destinations

In order to formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also conceive of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the lengths of all paths so far generated. In order for this measure to be minimized, each individual path must be of minimum length. Using this optimization measure, if we have already constructed i shortest paths then the next path to be constructed should be the next shortest minimum length path. The greedy way (and also a systematic way) to generate the shortest paths from v_0 to the remaining vertices would be to generate these paths in nondecreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated and so on. For the graph of Figure 4.10(a) the nearest vertex to v_0 is v_2 ($c(v_0, v_2) = 10$). The path v_0v_2 will be the first path generated. The second nearest vertex to v_0 is v_3 and the distance between v_0 andv_3 is 25. The path $v_0v_2v_3$ will be the next path generated. In order to generate the shortest paths in this order, we need to be able to deter-

requires a minor extension to this algorithm and is left as an exercise. In procedure SHORTEST_PATHS (Algorithm 4.11) it is assumed that the n vertices of G are numbered 1 through n. The set S is maintained as a bit array with $S(i) = 0$ if vertex i is not in S and $S(i) = 1$ if it is. It is assumed that the graph itself is represented by its cost adjacency matrix with COST(i, j) being the weight of the edge (i, j). COST(i, j) will be set to some large number, $+\infty$, in case the edge (i, j) is not in $E(G)$. For $i = j$, COST(i, j) may be set to any nonnegative number without affecting the outcome of the algorithm.

```
     procedure SHORTEST-PATHS(v, COST, DIST, n)
       //DIST(j), 1 ≤ j ≤ n is set to the length of the shortest path//
       //from vertex v to vertex j in a digraph G with n vertices.//
       //DIST(v) is set to zero. G is represented by its cost adjacency//
       //matrix, COST(n, n)//
       boolean S(1:n); real COST(1:n, 1:n), DIST(1:n)
       integer u, v, n, num, i, w
1      for i ← 1 to n do  //initialize set S to empty//
2        S(i) ← 0; DIST(i) ← COST(v, i)
3      repeat
4      S(v) ← 1; DIST(v) ← 0      //put vertex v in set S//
5      for num ← 2 to n − 1 do  //determine n − 1 paths from vertex v//
6        choose u such that DIST(u) = min{ DIST(w)}
                                      S(w) = 0
7        S(u) ← 1  //put vertex u in set S//
8        for all w with S(w) = 0 do  //update distances//
9          DIST(w) ← min(DIST(w), DIST(u) + COST(u, w))
10       repeat
11     repeat
12   end SHORTEST-PATHS
```

Algorithm 4.11 Greedy algorithm to generate shortest paths

Analysis of Algorithm SHORTEST-PATHS

From our earlier discussion, it is easy to see that the algorithm is correct. The time taken by the algorithm on a graph with n vertices is $O(n^2)$. To see this note that the **for** loop of line 1 takes $\theta(n)$ time. The **for** loop of line 5 is executed $n - 2$ times. Each execution of this loop requires $O(n)$ time at

11. *Steiner tree* Four villages are located at the vertices of a unit square in the Euclidean plane. You are asked to connect them by the shortest network of roads so that there is a path between every pair of the villages along those roads. Find such a network.

12. Write a program generating a random maze based on
   a. Prim's algorithm.
   b. Kruskal's algorithm.

3 Dijkstra's Algorithm

In this section, we consider the ***single-source shortest-paths problem***: for a given vertex called the ***source*** in a weighted connected graph, find shortest paths to all its other vertices. It is important to stress that we are not interested here in a single shortest path that starts at the source and visits all the other vertices. This would have been a much more difficult problem (actually, a version of the traveling salesman problem introduced in Section 3.4 and discussed again later in the book). The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

A variety of practical applications of the shortest-paths problem have made the problem a very popular object of study. The obvious but probably most widely used applications are transportation planning and packet routing in communication networks, including the Internet. Multitudes of less obvious applications include finding shortest paths in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling. In the world of entertainment, one can mention pathfinding in video games and finding best solutions to puzzles using their state-space graphs (see Section 6.6 for a very simple example of the latter).

There are several well-known algorithms for finding sh... includ... Floyd's algorithm for the more general all-pairs shortest-... iscu... in Chapter 8. Here, we consider the best-known algorith... e-so... shortest-paths problem, called ***Dijkstra's algorithm***.[4] This algorithm is applicab... to undirected and directed graphs with nonnegative weights only. Since in most applications this condition is satisfied, the limitation has not impaired the popularity of Dijkstra's algorithm.

Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source. First, it finds the shortest path from the source

4. Edsger W. Dijkstra (1930–2002), a noted Dutch pioneer of the science and industry of computing, discovered this algorithm in the mid-1950s. Dijkstra said about his algorithm: "This was the first graph problem I ever posed myself and solved. The amazing thing was that I didn't publish it. It was not amazing at the time. At the time, algorithms were hardly considered a scientific topic."

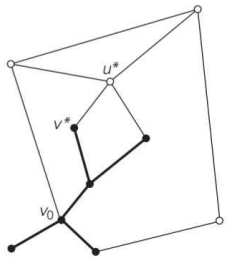


FIGURE 9.10 Idea of Dijkstra's algorithm. The subtree of the shortest paths already found is shown in bold. The next nearest to the source v_0 vertex, u^*, is selected by comparing the lengths of the subtree's paths increased by the distances to vertices adjacent to the subtree's vertices.

to a vertex nearest to it, then to a second nearest, and so on. In general, before its ith iteration commences, the algorithm has already identified the shortest paths to $i - 1$ other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph (Figure 9.10). Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i. The set of vertices adjacent to the vertices in T_i can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source. (Actually, all the other vertices can be treated as fringe vertices connected to tree vertices by edges of infinitely large weights.) To identify the ith nearest vertex, the algorithm computes, for every fringe vertex u, the sum of the distance to the nearest tree vertex v (given by the weight of the edge (v, u)) and the length d_v of the shortest path from the source to v (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

To facilitate the algorithm's operations, we label each vertex with two labels. The numeric label d indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree, d indicates the length of the shortest path from the source to that vertex. The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the source s and vertices that are adjacent to none of the current tree vertices.) With such labeling, finding the next nearest vertex u^* becomes a simple task of finding a fringe vertex with the smallest d value. Ties can be broken arbitrarily.

After we have identified a vertex u^* to be added to the tree, we need to perform two operations:

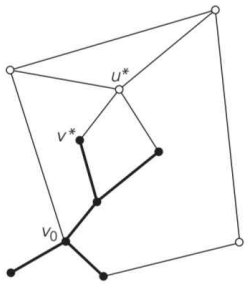


FIGURE 9.10 Idea of Dijkstra's algorithm. The subtree of the shortest paths already found is shown in bold. The next nearest to the source v_0 vertex, u^*, is selected by comparing the lengths of the subtree's paths increased by the distances to vertices adjacent to the subtree's vertices.

to a vertex nearest to it, then to a second nearest, and so on. In general, before its ith iteration commences, the algorithm has already identified the shortest paths to $i - 1$ other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph (Figure 9.10). Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i. The set of vertices adjacent to the vertices in T_i can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source. (Actually, all the other vertices can be treated as fringe vertices connected to tree vertices by edges of infinitely large weights.) To identify the ith nearest vertex, the algorithm computes, for every fringe vertex u, the sum of the distance to the nearest tree vertex v (given by the weight of the edge (v, u)) and the length d_v of the shortest path from the source to v (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

To facilitate the algorithm's operations, we label each vertex with two labels. The numeric label d indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree, d indicates the length of the shortest path from the source to that vertex. The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the source s and vertices that are adjacent to none of the current tree vertices.) With such labeling, finding the next nearest vertex u^* becomes a simple task of finding a fringe vertex with the smallest d value. Ties can be broken arbitrarily

After we have identified a vertex u^* to be added to the tree, we ne rm two operations:

- Move u^* from the fringe to the set of tree vertices.
- For each remaining fringe vertex u that is connected to u^* by an edge of weight $w(u^*, u)$ such that $d_{u^*} + w(u^*, u) < d_u$, update the labels of u by u^* and $d_{u^*} + w(u^*, u)$, respectively.

Figure 9.11 demonstrates the application of Dijkstra's algorithm to a specific graph.

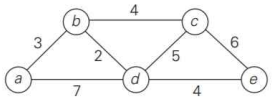
The labeling and mechanics of Dijkstra's algorithm are quite similar to those used by Prim's algorithm (see Section 9.1). Both of them construct an expanding subtree of vertices by selecting the next vertex from the priority queue of the remaining vertices. It is important not to mix them up, however. They solve different problems and therefore operate with priorities computed in a different manner: Dijkstra's algorithm compares path lengths and therefore must add edge weights, while Prim's algorithm compares the edge weights as given.

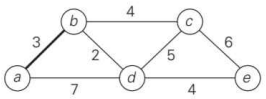
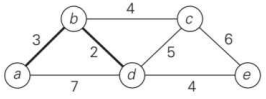
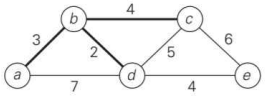
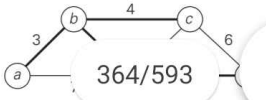
Now we can give pseudocode of Dijkstra's algorithm. It is spelled out—in more detail than Prim's algorithm was in Section 9.1—in terms of explicit operations on two sets of labeled vertices: the set V_T of vertices for which a shortest path has already been found and the priority queue Q of the fringe vertices. (Note that in the following pseudocode, V_T contains a given source vertex and the fringe contains the vertices adjacent to it *after* iteration 0 is completed.)

ALGORITHM *Dijkstra*(G, s)

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ with nonnegative weights
//       and its vertex s
//Output: The length d_v of a shortest path from s to v
//        and its penultimate vertex p_v for every vertex v in V
Initialize(Q)  //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;  p_v ← null
    Insert(Q, v, d_v)  //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)  //update priority of s v
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)  //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is adjacent to u* do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);  p_u ← u*
            Decrease(Q, u, d_u)
```


The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. For the reasons explained in the analysis of Prim's algorithm in Section 9.1, it is



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b:	$a - b$	of length 3
from a to d:	$a - b - d$	of length 5
from a to c:	$a - b - c$	of length 7
from a to e:	$a - b - d - e$	of length 9

FIGURE 9.11 Application of Dijkstra's algorithm. The next closest vertex is shown in bold.

the expected decode time is $\Sigma_{1 \le i \le n+1} q_i d_i$ where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that $\Sigma_{1 \le i \le n+1} q_i d_i$ is also the expected length of a transmitted message. Hence the code which minimizes expected decode time also minimizes the expected length of a message.

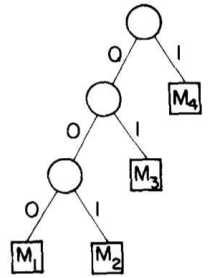


Figure 4.5 Huffman codes

4.6 MINIMUM SPANNING TREES

Definition Let $G = (V, E)$ be an undirected connected graph. A subgraph $T = (V, E')$ of G is a *spanning tree* of G iff T is a tree.

Example 4.8 Figure 4.6 shows the complete graph on 4 nodes together with three of its spanning trees. □

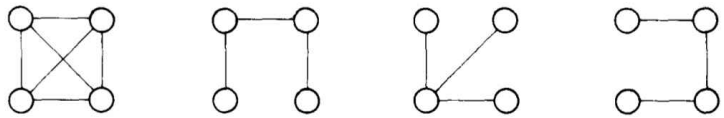


Figure 4.6 An undirected graph and three of its spanning trees

Spanning trees can be used to obtain an independent set of circuit equations for an electrical network. First, a spanning tree for the electrical network is obtained. Let B be the set of network edges not in the spanning tree. Adding an edge from B to the spanning tree creates a cycle. Different edges from B result in different cycles. Kirchoff's second law is used on

each cycle to obtain a circuit equation. The cycles obtained in this way are independent (i.e., none of these cycles can be obtained by taking a linear combination of the remaining cycles) as each contains an edge from B which is not contained in any other cycle. Hence, the circuit equations so obtained are also independent. In fact, it may be shown that the cycles obtained by introducing the edges of B one at a time into the resulting spanning tree form a cycle basis and so all other cycles in the graph can be constructed by taking a linear combination of the cycles in the basis (see Harary in the references for further details).

It is not difficult to imagine other applications for spanning trees. One that is of interest arises from the property that a spanning tree is a minimal subgraph G' of G such that $V(G') = V(G)$ and G' is connected (by a minimal subgraph, we mean one with the fewest number of edges). Any connected graph with n vertices must have at least $n - 1$ edges and all connected graphs with $n - 1$ edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting 2 cities, then the minimum number of links needed to connect the n cities is $n - 1$. The spanning trees of G will represent all feasible choices.

In any practical situation, however, the edges will have weights assigned to them. These weights might represent the cost of construction, the length of the link, etc. Given such a weighted graph one would then wish to select for construction a set of communication links that would connect all the cities and have minimum total cost or be of minimum total length. In either case the links selected will have to form a tree (assuming all weights are positive). In case this is not so, then the selection of links contains a cycle. Removal of any one of the links on this cycle will result in a link selection of less cost connecting all cities. We are therefore interested in finding a spanning tree of G with minimum cost. (The cost of a spanning

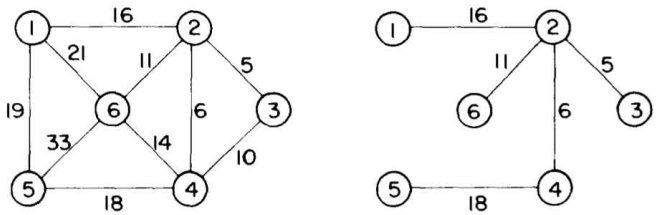


Figure 4.7 A graph and one of its minimum costs spanning trees

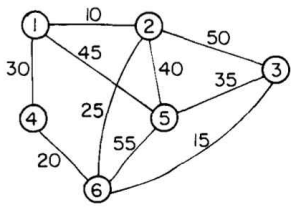


Figure 4.8(a) Graph for Examples 4.9 and 4.10

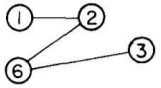
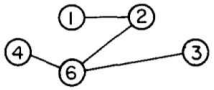
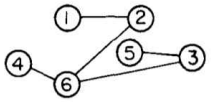
Edge	Cost	Spanning tree
(1,2)	10	
(2,6)	25	
(3,6)	15	
(6,4)	20	
(1,4)	reject	
(3,5)	35	

Figure 4.8(b) Stages in Prim's Algorithm

The time required by procedure PRIM is readily seen to be $\theta(n^2)$ where n is the number of vertices in the graph G. To see this note that line 3 takes $\theta(e)$ ($e = |E|$) time and line 4 takes $\theta(1)$ time. The loop of lines 6-9

```
line  procedure PRIM(E, COST, n, T, mincost)
      //E is the set of edges in G//
      //COST(n, n) is the cost adjacency matrix of an n vertex graph//
      //such that COST(i, j) is either a positive real number or + ∞ if//
      //no edge (i, j) exists. A minimum spanning tree is computed and//
      //stored as a set of edges in the array T(1:n - 1, 2). (T(i, 1),//
      //T(i, 2)) is an edge in the min-cost spanning tree. The final cost//
      //is assigned to mincost//
 1      real COST(n, n), mincost;
 2      integer NEAR(n), n, i, j, k, l, T(1:n - 1, 2);
 3      (k, l) ← edge with minimum cost
 4      mincost ← COST(k, l)
 5      (T(1, 1), T(1, 2)) ← (k, l)
 6      for i ← 1 to n do //initialize NEAR//
 7        if COST(i, l) < COST(i, k) then NEAR(i) ← l
 8                                   else NEAR(i) ← k endif
 9      repeat
10      NEAR(k) ← NEAR(l) ← 0
11      for i ← 2 to n - 1 do  //find n - 2 additional edges for T//
12      let j be an index such that NEAR (j) ≠ 0 and COST(j, NEAR (j))
          is minimum
13      (T(i, 1), T(i, 2)) ← (j, NEAR(j))
14      mincost ← mincost + COST(j, NEAR(j))
15      NEAR(j) ← 0
16      for k ← 1 to n do  //update NEAR//
17        if NEAR (k) ≠ 0 and COST(k, NEAR(k)) > COST(k, j)
18          then NEAR (k) ← j
19        endif
20      repeat
21    repeat
22    if mincost ≥ ∞ then print ('no spanning tree') endif
23  end PRIM
```

Algorithm 4.8 Prim's minimum spanning tree algorithm

takes $\theta(n)$ time. Line 12 and the loop of lines 16–20 require $\theta(n)$ time. So, each iteration of the loop of lines 11–21 takes $\theta(n)$ time. The total time for the loop of therefore $\theta(n^2)$. Hence, procedure PRIM has a time complexity that is $\theta(n^2)$.

The algorithm may be speeded a bit by making the observation that a minimum spanning tree includes for each vertex v a minimum cost edge

shows the forest represented by T during the various stages of this putation. The spanning tree obtained has a cost of 105. □

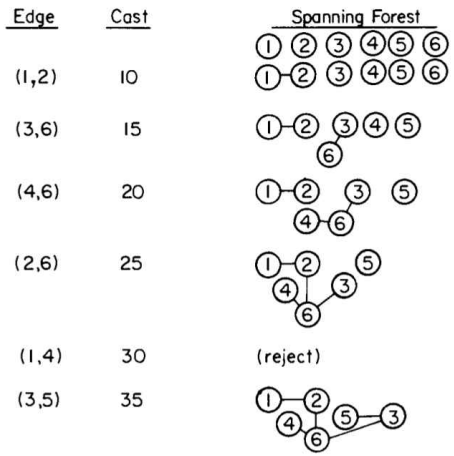


Figure 4.9 Stages in Kruskal's algorithm

For clarity, Kruskal's algorithm is written out more forma Algorithm 4.9. Initially E is the set of all edges in G. The only functi wish to perform on this set are: (i) determine an edge with minimu (line 3), and (ii) delete this edge (line 4). Both these functions c performed efficiently if the edges in E are maintained as a sorted sequ list. Actually, it is not essential to sort all the edges so long as th edge for line 3 can be determined easily. If the edges are maintaine min-heap then the next edge to consider can be obtained in $O(\log e$ if G has e edges. The construction of the heap itself takes $O(e)$ time.

If $j \neq k$ then vertices u and v are in different sets (and so in different trees) and edge (u, v) is included into T. The sets containing u and v are combined (line 12). If $u = v$ the edge (u, v) is discarded as its inclusion into T will create a cycle. Line 15 determines whether a spanning tree was found. It follows that $i \neq n - 1$ iff the graph G is not connected. One may verify that the computing time is $O(e \log e)$ where e is the number of edges in G ($e = |E|$).

```
line  procedure KRUSKAL (E, COST, n, T, mincost)
        //E is the set of edges in G. G has n vertices.        's th
        //cost of edge (u, v). T is the set of edges in        pa
        //ning tree and mincost is its cost//
 1      real mincost, COST (1:n, 1:n)
 2      integer PARENT(1:n), T(1:n - 1, 2), n
 3      construct a heap out of the edge costs using HEAPIFY
 4      PARENT ← -1 //each vertex is in a different set//
 5      i ← mincost ← 0
 6      while i < n - 1 and heap not empty do
 7        delete a minimum cost edge (u, v) from the heap and reheapify
          using ADJUST
 8        j ← FIND(u); k ← FIND(v)
 9        if j ≠ k then i ← i + 1
10                      T(i, 1) ← u; T(i, 2) ← v
11                      mincost ← mincost + COST(u, v)
12                      call UNION(j, k)
13        endif
14      repeat
15      if i ≠ n - 1 then print ('no spanning tree') endif
16      return
17    end KRUSKAL
```

Algorithm 4.10 Kruskal's Algorithm

Theorem 4.8 Kruskal's algorithm generates a minimum cost spanning tree for every connected undirected graph G.

Proof: Let G be any undirected connected graph. Let T be the spanning tree for G generated by Kruskal's algorithm. Let T' be a minimum cost spanning tree for G. We shall show that both T and T' have the same cost.

Let $E(T)$ and $E(T')$ respectively be the edges in T and T'. If n is the number of vertices in G then both T and T' have $n - 1$ edges. If $E(T) =$

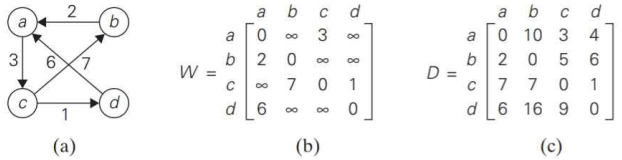


FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

mentioned at the beginning of this section has a better asymptotic efficiency than Warshall's algorithm (why?). We can speed up the above implementation of Warshall's algorithm for some inputs by restructuring its innermost loop (see Problem 4 in this section's exercises). Another way to make the algorithm run faster is to treat matrix rows as bit strings and employ the bitwise *or* operation available in most modern computer languages.

As to the space efficiency of Warshall's algorithm, the situation is similar to that of computing a Fibonacci number and some other dynamic programming algorithms. Although we used separate matrices for recording intermediate results of the algorithm, this is, in fact, unnecessary. Problem 3 in this section's exercises asks you to find a way of avoiding this wasteful use of the computer memory. Finally, we shall see below how the underlying idea of Warshall's algorithm can be applied to the more general problem of finding lengths of shortest paths in weighted graphs.

Floyd's Algorithm for the All-Pairs Shortest-Paths Problem

Given a weighted connected graph (undirected or directed), the ***all-pairs shortest-paths problem*** asks to find the distances—i.e., the lengths of the shortest paths—from each vertex to all other vertices. This is one of several variations of the problem involving shortest paths in graphs. Because of its important applications to communications, transportation networks, and operations research, it has been thoroughly studied over the years. Among recent applications of the all-pairs shortest-path problem is precomputing distances for motion planning in computer games.

It is convenient to record the lengths of shortest paths in an $n \times n$ matrix D called the ***distance matrix***: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex. For an example, see Figure 8.14.

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called ***Floyd's algorithm*** after its co-inventor Robert W. Floyd.[1] It is applicable to both undirected and directed weighted graphs provided

[1] Floyd explicitly referenced Warshall's paper in presenting his algorithm [Flo62]. Three years earlier, Bernard Roy published essentially the same algorithm in the proceedings of the French Academy of Sciences [Roy59].

length of the shortest path from v_k to v_j among the paths that use intermediate

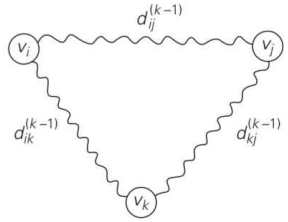


FIGURE 8.15 Underlying idea of Floyd's algorithm.

vertices numbered not higher than $k-1$ is equal to $d_{kj}^{(k-1)}$, the length of the shortest path among the paths that use the kth vertex is equal to $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$. Taking into account the lengths of the shortest paths in both subsets leads to the following recurrence:

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\ \ d_{ij}^{(0)} = w_{ij}. \qquad \textbf{(8.14)}$$

To put it another way, the element in row i and column j of the current distance matrix $D^{(k-1)}$ is replaced by the sum of the elements in the same row i and the column k and in the same column j and the row k if and only if the latter sum is smaller than its current value.

The application of Floyd's algorithm to the graph in Figure 8.14 is illustrated in Figure 8.16.

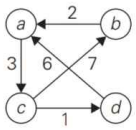
Here is pseudocode of Floyd's algorithm. It takes advantage of the fact that the next matrix in sequence (8.12) can be written over its predecessor.

ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W  //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Obviously, the time efficiency of Floyd's algorithm is cubic—as is the time efficiency of Warshall's algorithm. In the next chapter, we examine Dijkstra's algorithm—another method for finding shortest paths.

efficiency of Warshall's algorithm. In the next chapter, we examine Dijkstra's algorithm—another method for finding shortest paths.



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FIGURE 8.16 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

Exercises 8.4

1. Apply Warshall's algorithm to find the transitive closure of the digraph defined by the following adjacency matrix:

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

2. a. Prove that the time efficiency of Warshall's algorithm is cubic.

b. Explain why the time efficiency class of Warshall's algorithm is inferior to that of the traversal-based algorithm for sparse graphs represented by their adjacency lists.