Chapter 8

(Dynamic Programming)

8

Dynamic Programming

An idea, like a ghost . . . must be spoken to a little before it will explain itself.

—Charles Dickens (1812–1870)

Dynamic programming is an algorithm design technique with a rather interesting history. It was invented by a prominent U.S. mathematician, Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes. Thus, the word "programming" in the name of this technique stands for "planning" and does not refer to computer programming. After proving its worth as an important tool of applied mathematics, dynamic programming has eventually come to be considered, at least in computer science circles, as a general algorithm design technique that does not have to be limited to special types of optimization problems. It is from this point of view that we will consider this technique here.

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

This technique can be illustrated by revisiting the Fibonacci numbers discussed in Section 2.5. (If you have not read that section, you will be able to follow the discussion anyway. But it is a beautiful topic, so if you feel a temptation to read it, do succumb to it.) The Fibonacci numbers are the elements of the sequence

$$0, \quad 1, \quad 1, \quad 2, \quad 3, \quad 5, \quad 8, \quad 13, \quad 21, \quad 34, \ldots,$$

which can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1 \tag{8.1}$$

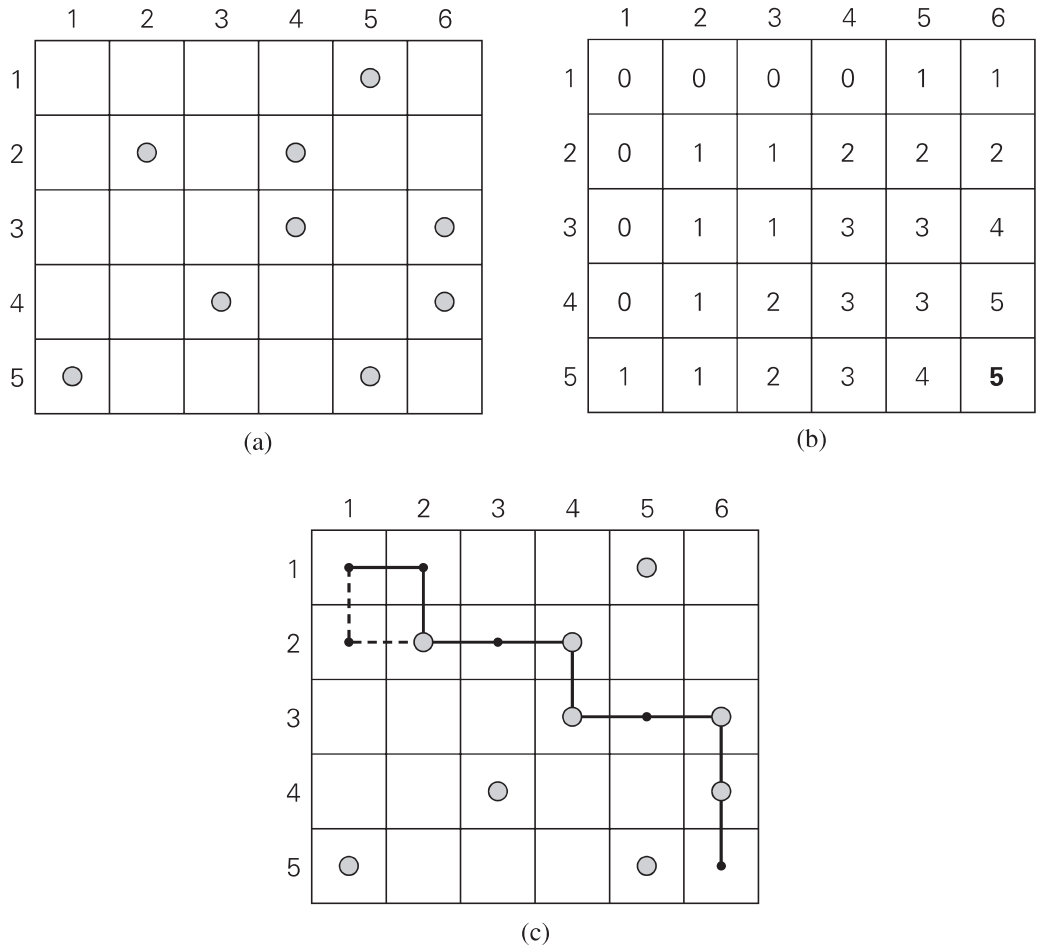

FIGURE 8.3 (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.

Exercises 8.1

1. What does dynamic programming have in common with divide-and-conquer? What is a principal difference between them?

2. Solve the instance 5, 1, 2, 10, 6 of the coin-row problem.

3. **a.** Show that the time efficiency of solving the coin-row problem by straightforward application of recurrence (8.3) is exponential.

b. Show that the time efficiency of solving the coin-row problem by exhaustive search is at least exponential.

4. Apply the dynamic programming algorithm to find all the solutions to the change-making problem for the denominations 1, 3, 5 and the amount $n = 9$.

a. Give an example of three matrices for which the number of multiplications in $(A_1 \cdot A_2) \cdot A_3$ and $A_1 \cdot (A_2 \cdot A_3)$ differ at least by a factor of 1000.

b. How many different ways are there to compute the product of n matrices?

c. Design a dynamic programming algorithm for finding an optimal order of multiplying n matrices.

8.4 Warshall's and Floyd's Algorithms

In this section, we look at two well-known algorithms: Warshall's algorithm for computing the transitive closure of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on essentially the same idea: exploit a relationship between a problem and its simpler rather than smaller version. Warshall and Floyd published their algorithms without mentioning dynamic programming. Nevertheless, the algorithms certainly have a dynamic programming flavor and have come to be considered applications of this technique.

Warshall's Algorithm

Recall that the adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its ith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex. We may also be interested in a matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph. Such a matrix, called the transitive closure of the digraph, would allow us to determine in constant time whether the jth vertex is reachable from the ith vertex.

Here are a few application examples. When a value in a spreadsheet cell is changed, the spreadsheet software must know all the other cells affected by the change. If the spreadsheet is modeled by a digraph whose vertices represent the spreadsheet cells and edges indicate cell dependencies, the transitive closure will provide such information. In software engineering, transitive closure can be used for investigating data flow and control flow dependencies as well as for inheritance testing of object-oriented software. In electronic engineering, it is used for redundancy identification and test generation for digital circuits.

DEFINITION The ***transitive closure*** of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, t_{ij} is 0.

An example of a digraph, its adjacency matrix, and its transitive closure is given in Figure 8.11.

We can generate the transitive closure of a digraph with the help of depth-first search or breadth-first search. Performing either traversal starting at the ith

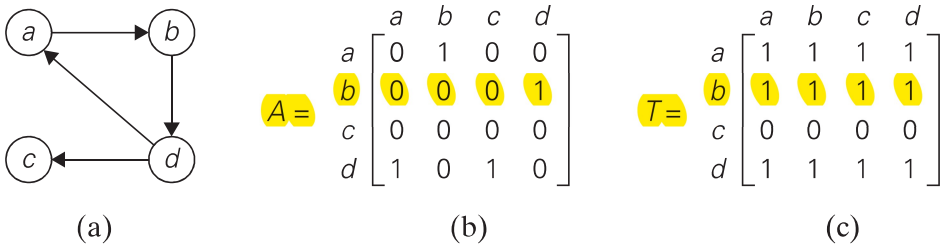

FIGURE 8.11 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

vertex gives the information about the vertices reachable from it and hence the columns that contain 1's in the ith row of the transitive closure. Thus, doing such a traversal for every vertex as a starting point yields the transitive closure in its entirety.

Since this method traverses the same digraph several times, we should hope that a better algorithm can be found. Indeed, such an algorithm exists. It is called ***Warshall's algorithm*** after Stephen Warshall, who discovered it [War62]. It is convenient to assume that the digraph's vertices and hence the rows and columns of the adjacency matrix are numbered from 1 to n. Warshall's algorithm constructs the transitive closure through a series of $n \times n$ boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}. \tag{8.9}$$

Each of these matrices provides certain information about directed paths in the digraph. Specifically, the element $r_{ij}^{(k)}$ in the ith row and jth column of matrix $R^{(k)}$ $(i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n)$ is equal to 1 if and only if there exists a directed path of a positive length from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. Thus, the series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph. (Recall that the adjacency matrix contains the information about one-edge paths, i.e., paths with no intermediate vertices.) $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate; thus, with more freedom, so to speak, it may contain more 1's than $R^{(0)}$. In general, each subsequent matrix in series (8.9) has one more vertex to use as intermediate for its paths than its predecessor and hence may, but does not have to, contain more 1's. The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

The central point of the algorithm is that we can compute all the elements of each matrix $R^{(k)}$ from its immediate predecessor $R^{(k-1)}$ in series (8.9). Let $r_{ij}^{(k)}$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k:

$$v_i, \text{ a list of intermediate vertices each numbered not higher than } k, v_j. \tag{8.10}$$

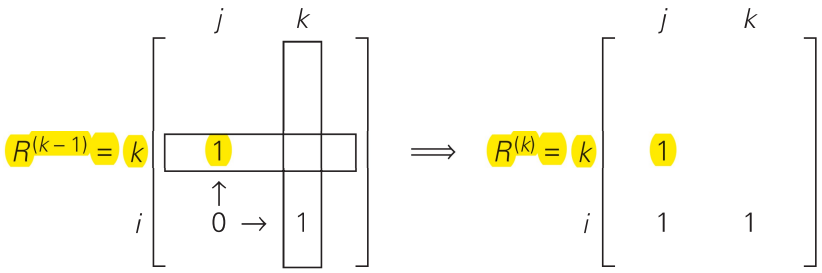

FIGURE 8.12 Rule for changing zeros in Warshall's algorithm.

Two situations regarding this path are possible. In the first, the list of its intermediate vertices does not contain the kth vertex. Then this path from v_i to v_j has intermediate vertices numbered not higher than $k-1$, and therefore $r_{ij}^{(k-1)}$ is equal to 1 as well. The second possibility is that path (8.10) does contain the kth vertex v_k among the intermediate vertices. Without loss of generality, we may assume that v_k occurs only once in that list. (If it is not the case, we can create a new path from v_i to v_j with this property by simply eliminating all the vertices between the first and last occurrences of v_k in it.) With this caveat, path (8.10) can be rewritten as follows:

$$v_i, \text{ vertices numbered} \leq k-1, v_k, \text{ vertices numbered} \leq k-1, v_j.$$

The first part of this representation means that there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ (hence, $r_{ik}^{(k-1)} = 1$), and the second part means that there exists a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$ (hence, $r_{kj}^{(k-1)} = 1$).

What we have just proved is that if $r_{ij}^{(k)} = 1$, then either $r_{ij}^{(k-1)} = 1$ or both $r_{ik}^{(k-1)} = 1$ and $r_{kj}^{(k-1)} = 1$. It is easy to see that the converse of this assertion is also true. Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right). \tag{8.11}$$

Formula (8.11) is at the heart of Warshall's algorithm. This formula implies the following rule for generating elements of matrix $R^{(k)}$ from elements of matrix $R^{(k-1)}$, which is particularly convenient for applying Warshall's algorithm by hand:

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$. This rule is illustrated in Figure 8.12.

As an example, the application of Warshall's algorithm to the digraph in Figure 8.11 is shown in Figure 8.13.

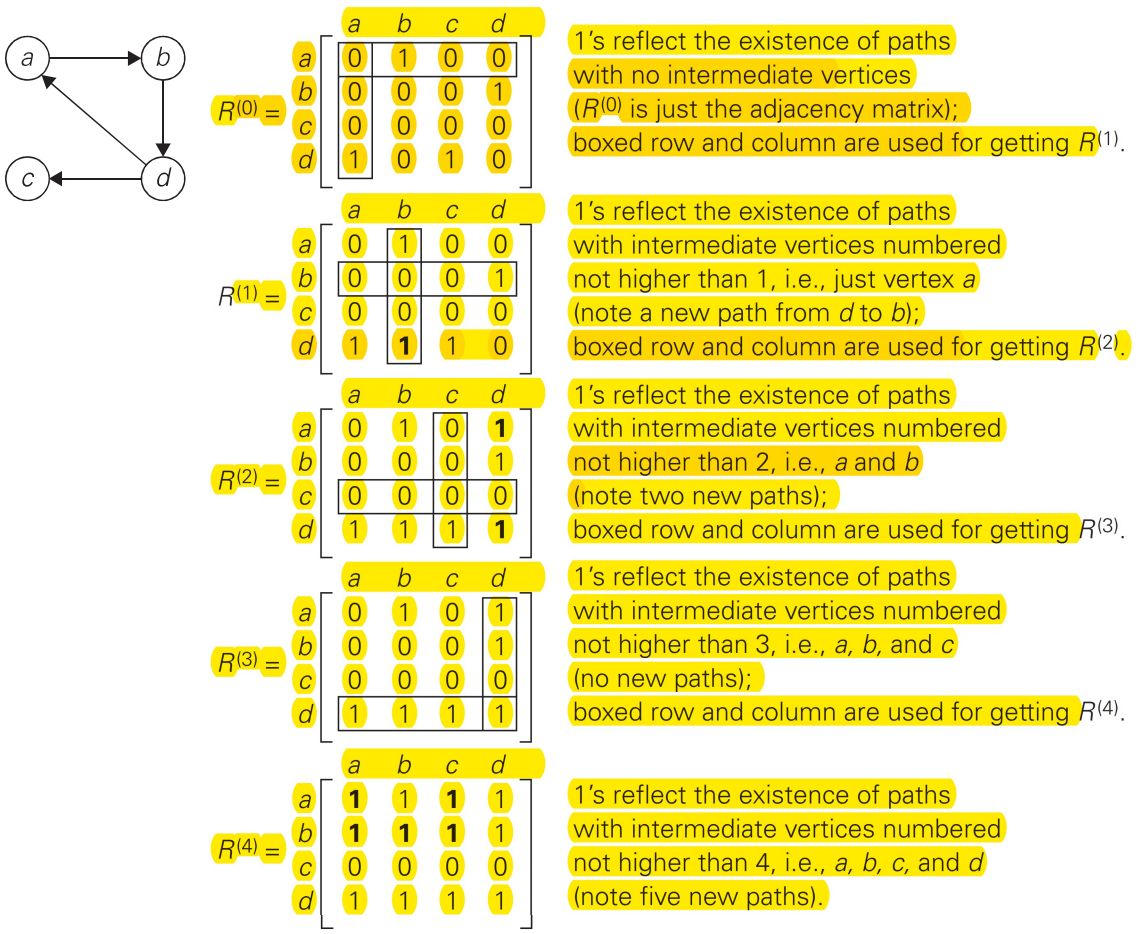

FIGURE 8.13 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

Here is pseudocode of Warshall's algorithm.

ALGORITHM *Warshall*$(A[1..n, 1..n])$
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Several observations need to be made about Warshall's algorithm. First, it is remarkably succinct, is it not? Still, its time efficiency is only $\Theta(n^3)$. In fact, for sparse graphs represented by their adjacency lists, the traversal-based algorithm

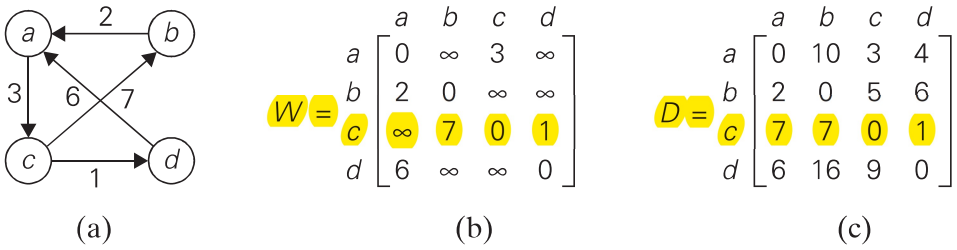

FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

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].

that they do not contain a cycle of a negative length. (The distance between any two vertices in such a cycle can be made arbitrarily small by repeating the cycle enough times.) The algorithm can be enhanced to find not only the lengths of the shortest paths for all vertex pairs but also the shortest paths themselves (Problem 10 in this section's exercises).

Floyd's algorithm computes the distance matrix of a weighted graph with n vertices through a series of $n \times n$ matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}. \quad \textbf{(8.12)}$$

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question. Specifically, the element $d_{ij}^{(k)}$ in the ith row and the jth column of matrix $D^{(k)}$ ($i, j = 1, 2, \ldots, n$, $k = 0, 1, \ldots, n$) is equal to the length of the shortest path among all paths from the ith vertex to the jth vertex with each intermediate vertex, if any, numbered not higher than k. In particular, the series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is simply the weight matrix of the graph. The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing other than the distance matrix being sought.

As in Warshall's algorithm, we can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$ in series (8.12). Let $d_{ij}^{(k)}$ be the element in the ith row and the jth column of matrix $D^{(k)}$. This means that $d_{ij}^{(k)}$ is equal to the length of the shortest path among all paths from the ith vertex v_i to the jth vertex v_j with their intermediate vertices numbered not higher than k:

$$v_i, \text{ a list of intermediate vertices each numbered not higher than } k, v_j. \quad \textbf{(8.13)}$$

We can partition all such paths into two disjoint subsets: those that do not use the kth vertex v_k as intermediate and those that do. Since the paths of the first subset have their intermediate vertices numbered not higher than $k-1$, the shortest of them is, by definition of our matrices, of length $d_{ij}^{(k-1)}$.

What is the length of the shortest path in the second subset? If the graph does not contain a cycle of a negative length, we can limit our attention only to the paths in the second subset that use vertex v_k as their intermediate vertex exactly once (because visiting v_k more than once can only increase the path's length). All such paths have the following form:

$$v_i, \text{ vertices numbered} \le k-1, v_k, \text{ vertices numbered} \le k-1, v_j.$$

In other words, each of the paths is made up of a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ and a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$. The situation is depicted symbolically in Figure 8.15.

Since the length of the shortest path from v_i to v_k among the paths that use intermediate vertices numbered not higher than $k-1$ is equal to $d_{ik}^{(k-1)}$ and the length of the shortest path from v_k to v_j among the paths that use intermediate

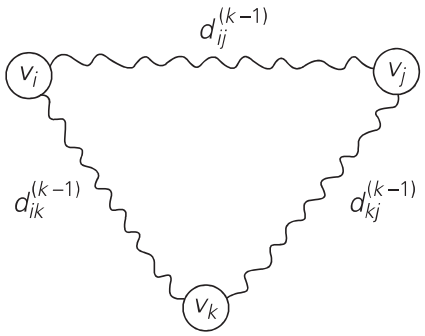

FIGURE 8.15 Underlying idea of Floyd's algorithm.

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:

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

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.

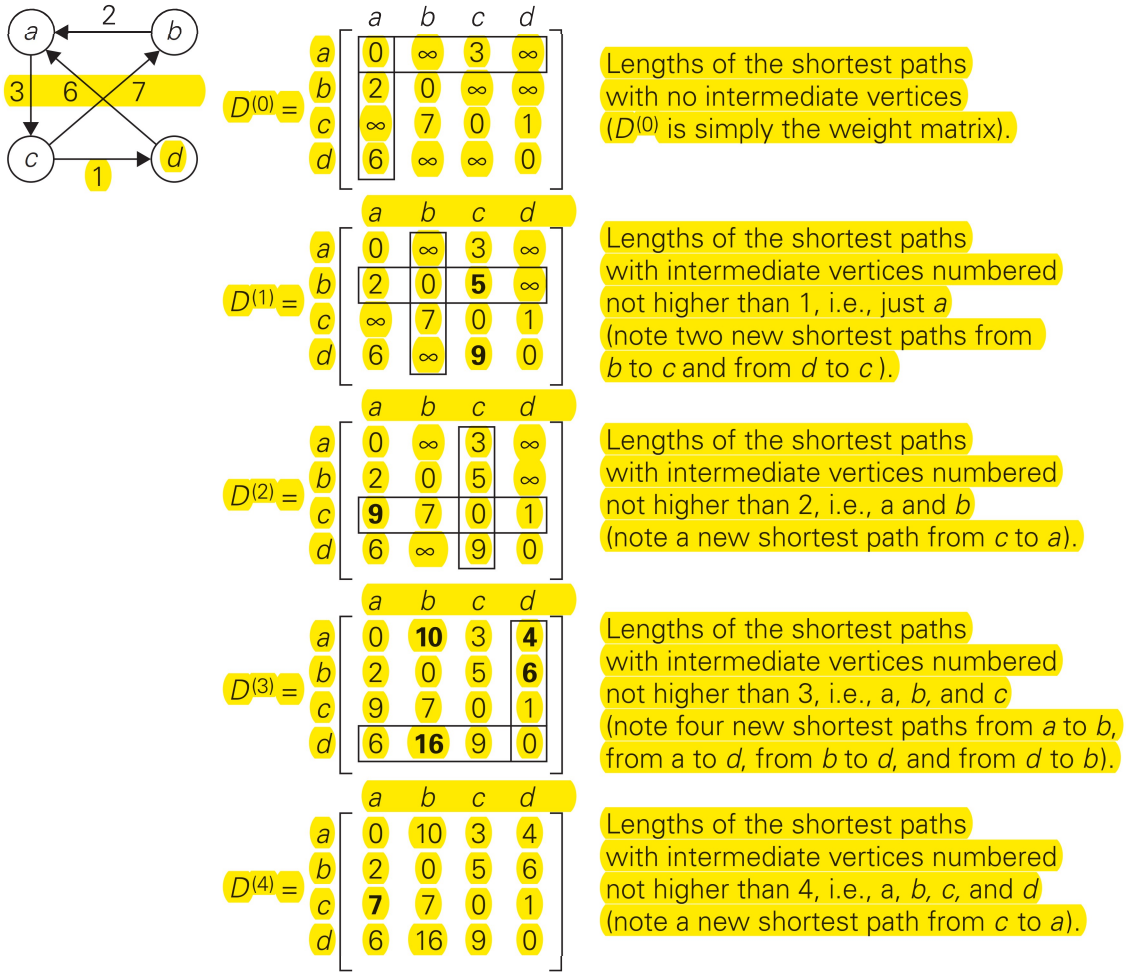

FIGURE 8.16 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

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.

3. Explain how to implement Warshall's algorithm without using extra memory for storing elements of the algorithm's intermediate matrices.
4. Explain how to restructure the innermost loop of the algorithm *Warshall* to make it run faster at least on some inputs.
5. Rewrite pseudocode of Warshall's algorithm assuming that the matrix rows are represented by bit strings on which the bitwise *or* operation can be performed.
6. **a.** Explain how Warshall's algorithm can be used to determine whether a given digraph is a dag (directed acyclic graph). Is it a good algorithm for this problem?
   b. Is it a good idea to apply Warshall's algorithm to find the transitive closure of an undirected graph?
7. Solve the all-pairs shortest-path problem for the digraph with the following weight matrix:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

8. Prove that the next matrix in sequence (8.12) of Floyd's algorithm can be written over its predecessor.
9. Give an example of a graph or a digraph with negative weights for which Floyd's algorithm does not yield the correct result.
10. Enhance Floyd's algorithm so that shortest paths themselves, not just their lengths, can be found.
11. *Jack Straws* In the game of Jack Straws, a number of plastic or wooden "straws" are dumped on the table and players try to remove them one by one without disturbing the other straws. Here, we are only concerned with whether various pairs of straws are connected by a path of touching straws. Given a list of the endpoints for $n > 1$ straws (as if they were dumped on a large piece of graph paper), determine all the pairs of straws that are connected. Note that touching is connecting, but also that two straws can be connected indirectly via other connected straws. [1994 East-Central Regionals of the ACM International Collegiate Programming Contest]

SUMMARY

- *Dynamic programming* is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the