4

SHORTEST PATHS: LABEL-SETTING ALGORITHMS

A journey of a thousand miles starts with a single step and if that step is the right step, it becomes the last step.
—Lao Tzu

Chapter Outline

4.1 INTRODUCTION

Shortest path problems lie at the heart of network flows. They are alluring to both researchers and to practitioners for several reasons: (1) they arise frequently in practice since in a wide variety of application settings we wish to send some material (e.g., a computer data packet, a telephone call, a vehicle) between two specified points in a network as quickly, as cheaply, or as reliably as possible; (2) they are easy to solve efficiently; (3) as the simplest network models, they capture many of the most salient core ingredients of network flows and so they provide both a benchmark and a point of departure for studying more complex network models; and (4) they arise frequently as subproblems when solving many combinatorial and network optimization problems. Even though shortest path problems are relatively easy to solve, the design and analysis of most efficient algorithms for solving them requires considerable ingenuity. Consequently, the study of shortest path problems is a natural starting point for introducing many key ideas from network flows, including the use of clever data structures and ideas such as data scaling to improve the worst-case algorithmic performance. Therefore, in this and the next chapter, we begin our discussion of network flow algorithms by studying shortest path problems.

We first set our notation and describe several assumptions that we will invoke throughout our discussion.

Notation and Assumptions

We consider a directed network $G = (N, A)$ with an *arc length* (or *arc cost*) c_{ij} associated with each arc $(i, j) \in A$. The network has a distinguished node s, called the *source*. Let $A(i)$ represent the arc adjacency list of node i and let $C = \max\{c_{ij} : (i, j) \in A\}$. We define the *length of a directed path* as the sum of the lengths of arcs in the path. The shortest path problem is to determine for every nonsource node $i \in N$ a shortest length directed path from node s to node i. Alternatively, we might view the problem as sending 1 unit of flow as cheaply as possible (with arc flow costs as c_{ij}) from node s to each of the nodes in $N - \{s\}$ in an uncapacitated network. This viewpoint gives rise to the following linear programming formulation of the shortest path problem.

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{4.1a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} n-1 & \text{for } i = s \\ -1 & \text{for all } i \in N - \{s\} \end{cases} \tag{4.1b}$$

$$x_{ij} \geq 0 \quad \text{for all } (i, j) \in A. \tag{4.1c}$$

In our study of the shortest path problem, we will impose several assumptions.

Assumption 4.1. *All arc lengths are integers.*

The integrality assumption imposed on arc lengths is necessary for some algorithms and unnecessary for others. That is, for some algorithms we can relax it and still perform the same analysis. Algorithms whose complexity bound depends on C assume integrality of the data. Note that we can always transform rational arc capacities to integer arc capacities by multiplying them by a suitably large number. Moreover, we necessarily need to convert irrational numbers to rational numbers to represent them on a computer. Therefore, the integrality assumption is really not a restrictive assumption in practice.

Assumption 4.2. *The network contains a directed path from node s to every other node in the network.*

We can always satisfy this assumption by adding a "fictitious" arc (s, i) of suitably large cost for each node i that is not connected to node s by a directed path.

Assumption 4.3. *The network does not contain a negative cycle (i.e., a directed cycle of negative length).*

Observe that for any network containing a negative cycle W, the linear programming formulation (4.1) has an unbounded solution because we can send an infinite amount of flow along W. The shortest path problem with a negative cycle is substantially harder to solve than is the shortest path problem without a negative cycle. Indeed, because the shortest path problem with a negative cycle is an $\mathcal{NP}$-complete problem, no polynomial-time algorithm for this problem is likely to exist (see Appendix B for the definition of $\mathcal{NP}$-complete problems). Negative cycles complicate matters, in part, for the following reason. All algorithms that are capable of solving shortest path problems with negative length arcs essentially determine shortest length directed walks from the source to other nodes. If the network contains no negative cycle, then some shortest length directed walk is a path (i.e., does not repeat nodes), since we can eliminate directed cycles from this walk without increasing its length. The situation for networks with negative cycles is quite different; in these situations, the shortest length directed walk might traverse a negative cycle an infinite number of times since each such repetition reduces the length of the walk. In these cases we need to prohibit walks that revisit nodes; the addition of this apparently mild stipulation has significant computational implications: With it, the shortest path problem becomes substantially more difficult to solve.

Assumption 4.4. *The network is directed.*

If the network were undirected and all arc lengths were nonnegative, we could transform this shortest path problem to one on a directed network. We described this transformation in Section 2.4. If we wish to solve the shortest path problem on an undirected network and some arc lengths are negative, the transformation described in Section 2.4 does not work because each arc with negative length would produce a negative cycle. We need a more complex transformation to handle this situation, which we describe in Section 12.7.

Various Types of Shortest Path Problems

Researchers have studied several different types of (directed) shortest path problems:

1. Finding shortest paths from one node to all other nodes when arc lengths are nonnegative
2. Finding shortest paths from one node to all other nodes for networks with arbitrary arc lengths
3. Finding shortest paths from every node to every other node
4. Various generalizations of the shortest path problem

In this and the following chapter we discuss the first three of these problem types. We refer to problem types (1) and (2) as the *single-source shortest path problem* (or, simply, the *shortest path problem*), and the problem type (3) as the *all-pairs shortest path problem.* In the exercises of this chapter we consider the following variations of the shortest path problem: (1) the maximum capacity path problem, (2) the maximum reliability path problem, (3) shortest paths with turn penalties, (4) shortest paths with an additional constraint, and (5) the resource-constrained shortest path problem.

Analog Solution of the Shortest Path Problem

The shortest path problem has a particularly simple structure that has allowed researchers to develop several intuitively appealing algorithms for solving it. The following analog model for the shortest path problem (with nonnegative arc lengths) provides valuable insight that helps in understanding some of the essential features of the shortest path problem. Consider a shortest path problem between a specified pair of nodes s and t (this discussion extends easily for the general shortest path model with multiple destination nodes and with nonnegative arc lengths). We construct a string model with nodes represented by knots, and for any arc (i, j) in A, a string with length equal to c_{ij} joining the two knots i and j. We assume that none of the strings can be stretched. After constructing the model, we hold the knot representing node s in one hand, the knot representing node t in the other hand, and pull our hands apart. One or more paths will be held tight; these are the shortest paths from node s to node t.

We can extract several insights about the shortest path problem from this simple string model:

1. For any arc on a shortest path, the string will be taut. Therefore, the shortest path distance between any two successive nodes i and j on this path will equal the length c_{ij} of the arc (i, j) between these nodes.
2. For any two nodes i and j on the shortest path (which need not be successive nodes on the path) that are connected by an arc (i, j) in A, the shortest path distance from the source to node i plus c_{ij} (a composite distance) is always as large as the shortest path distance from the source to node j. The composite distance might be larger because the string between nodes i and j might not be taut.
3. To solve the shortest path problem, we have solved an associated *maximization* problem (by pulling the string apart). As we will see in our later discussions, in general, all network flow problems modeled as minimization problems have an associated "dual" maximization problem; by solving one problem, we generally solve the other as well.

Label-Setting and Label-Correcting Algorithms

The network flow literature typically classifies algorithmic approaches for solving shortest path problems into two groups: *label setting* and *label correcting*. Both approaches are iterative. They assign tentative distance labels to nodes at each step; the distance labels are estimates of (i.e., upper bounds on) the shortest path distances. The approaches vary in how they update the distance labels from step to step and how they "converge" toward the shortest path distances. Label-setting algorithms designate one label as permanent (optimal) at each iteration. In contrast, label-correcting algorithms consider all labels as temporary until the final step, when they all become permanent. One distinguishing feature of these approaches is the class of problems that they solve. Label-setting algorithms are applicable only to (1) shortest path problems defined on acyclic networks with arbitrary arc lengths, and to (2) shortest path problems with nonnegative arc lengths. The label-correcting algorithms are more general and apply to all classes of problems, including those with negative arc lengths. The label-setting algorithms are much more efficient, that is, have much better worst-case complexity bounds; on the other hand, the label-correcting algorithms not only apply to more general classes of problems, but as we will see, they also offer more algorithmic flexibility. In fact, we can view the label-setting algorithms as special cases of the label-correcting algorithms.

In this chapter we study label-setting algorithms; in Chapter 5 we study label-correcting algorithms. We have divided our discussion in two parts for several reasons. First, we wish to emphasize the difference between these two solution approaches and the different algorithmic strategies that they employ. The two problem approaches also differ in the types of data structures that they employ. Moreover, the analysis of the two types of algorithms is quite different. The convergence proofs for label-setting algorithms are much simpler and rely on elementary combinatorial arguments. The proofs for the label-correcting algorithms tend to be much more subtle and require more careful analysis.

Chapter Overview

The basic label-setting algorithm has become known as *Dijkstra's algorithm* because Dijkstra was one of several people to discover it independently. In this chapter we study several variants of Dijkstra's algorithm. We first describe a simple implementation that achieves a time bound of $O(n^2)$. Other implementations improve on this implementation either empirically or theoretically. We describe an implementation due to Dial that achieves an excellent running time in practice. We also consider several versions of Dijkstra's algorithm that improve upon its worst-case complexity. Each of these implementations uses a *heap* (or *priority queue*) data structure. We consider several such implementations, using data structures known as binary heaps, d-heaps, Fibonacci heaps, and the recently developed radix heap. Before examining these various algorithmic approaches, we first describe some applications of the shortest path problem.

4.2 APPLICATIONS

Shortest path problems arise in a wide variety of practical problem settings, both as stand-alone models and as subproblems in more complex problem settings. For example, they arise in the telecommunications and transportation industries whenever we want to send a message or a vehicle between two geographical locations as quickly or as cheaply as possible. Urban traffic planning provides another important example: The models that urban planners use for computing traffic flow patterns are complex nonlinear optimization problems or complex equilibrium models; they build, however, on the behavioral assumption that users of the transportation system travel, with respect to prevailing traffic congestion, along shortest paths from their origins to their destinations. Consequently, most algorithmic approaches for finding urban traffic patterns solve a large number of shortest path problems as subproblems (one for each origin–destination pair in the network).

In this book we consider many other applications like this with embedded shortest path models. These many and varied applications attest to the importance of shortest path problems in practice. In Chapters 1 and 19 we discuss a number of stand-alone shortest path models in such problem contexts as urban housing, project management, inventory planning, and DNA sequencing. In this section and in the exercises in this chapter, we consider several other applications of shortest paths that are indicative of the range of applications of this core network flow model. These applications include generic mathematical applications—approximating functions, solving certain types of difference equations, and solving the so-called knapsack problem—as well as direct applications in the domains of production planning, telephone operator scheduling, and vehicle fleet planning.

Application 4.1 Approximating Piecewise Linear Functions

Numerous applications encountered within many different scientific fields use piecewise linear functions. On several occasions, these functions contain a large number of breakpoints; hence they are expensive to store and to manipulate (e.g., even to evaluate). In these situations it might be advantageous to replace the piecewise linear function by another approximating function that uses fewer breakpoints. By approximating the function we will generally be able to save on storage space and on the cost of using the function; we will, however, incur a cost because of the inaccuracy of the approximating function. In making the approximation, we would like to make the best possible trade-off between these conflicting costs and benefits.

Let $f_1(x)$ be a piecewise linear function of a scalar x. We represent the function in the two-dimensional plane: It passes through n points $a_1 = (x_1, y_1)$, $a_2 = (x_2, y_2), \ldots, a_n = (x_n, y_n)$. Suppose that we have ordered the points so that $x_1 \le x_2 \le \cdots \le x_n$. We assume that the function varies linearly between every two consecutive points x_i and x_{i+1}. We consider situations in which n is very large and for practical reasons we wish to approximate the function $f_1(x)$ by another function $f_2(x)$ that passes through only a subset of the points $a_1, a_2, \ldots, a_n$ (including a_1 and a_n). As an example, consider Figure 4.1(a): In this figure we have approximated a function $f_1(x)$ passing through 10 points by a function $f_2(x)$ drawn with dashed lines) passing through only five of the points.

This approximation results in a savings in storage space and in the use of the function. For purposes of illustration, assume that we can measure these costs by a per unit cost α associated with any single interval used in the approximation (which

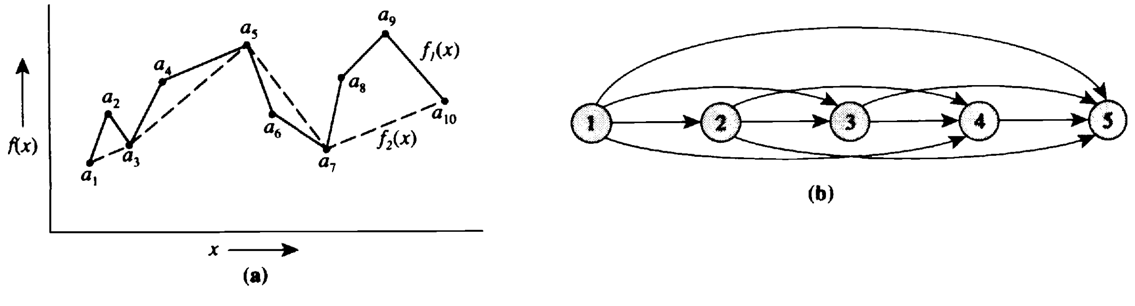

Figure 4.1 Illustrating Applications 4.1: (a) approximating the function $f_1(x)$ passing through 10 points by the function $f_2(x)$; (b) corresponding shortest path problem.

is defined by two points, a_i and a_j). As we have noted, the approximation also introduces errors that have an associated penalty. We assume that the error of an approximation is proportional to the sum of the squared errors between the actual data points and the estimated points (i.e., the penalty is $\beta \sum_{i=1}^{n} [f_1(x_i) - f_2(x_i)]^2$ for some constant β). Our decision problem is to identify the subset of points to be used to define the approximation function $f_2(x)$ so that we incur the minimum total cost as measured by the sum of the cost of storing and using the approximating function and the cost of the errors imposed by the approximation.

We will formulate this problem as a shortest path problem on a network G with n nodes, numbered 1 through n, as follows. The network contains an arc (i, j) for each pair of nodes i and j such that $i < j$. Figure 4.1(b) gives an example of the network with $n = 5$ nodes. The arc (i, j) in this network signifies that we approximate the linear segments of the function $f_1(x)$ between the points $a_i, a_{i+1}, \ldots, a_j$ by one linear segment joining the points a_i and a_j. The cost c_{ij} of the arc (i, j) has two components: the storage cost α and the penalty associated with approximating all the points between a_i and a_j by the corresponding points lying on the line joining a_i and a_j. In the interval $[x_i, x_j]$, the approximating function is $f_2(x) = f_1(x_i) + (x - x_i)[f_1(x_j) - f_1(x_i)]/(x_j - x_i)$, so the total cost in this interval is

$$c_{ij} = \alpha + \beta\left[\sum_{k=i}^{j} (f_1(x_k) - f_2(x_k))^2\right].$$

Each directed path from node 1 to node n in G corresponds to a function $f_2(x)$, and the cost of this path equals the total cost for storing this function and for using it to approximate the original function. For example, the path 1–3–5 corresponds to the function $f_2(x)$ passing through the points a_1, a_3, and a_5. As a consequence of these observations, we see that the shortest path from node 1 to node n specifies the optimal set of points needed to define the approximating function $f_2(x)$.

Application 4.2 Allocating Inspection Effort on a Production Line

A production line consists of an ordered sequence of n production stages, and each stage has a manufacturing operation followed by a potential inspection. The product enters stage 1 of the production line in batches of size $B \geq 1$. As the items within a batch move through the manufacturing stages, the operations might introduce defects. The probability of producing a defect at stage i is α_i. We assume that all of the defects are nonrepairable, so we must scrap any defective item. After each stage, we can either inspect all of the items or none of them (we do not sample the items); we assume that the inspection identifies every defective item. The production line must end with an inspection station so that we do not ship any defective units. Our decision problem is to find an optimal inspection plan that specifies at which stages we should inspect the items so that we minimize the total cost of production and inspection. Using fewer inspection stations might decrease the inspection costs, but will increase the production costs because we might perform unnecessary manufacturing operations on some units that are already defective. The optimal number of inspection stations will achieve an appropriate trade-off between these two conflicting cost considerations.

Suppose that the following cost data are available: (1) p_i, the manufacturing cost per unit in stage i; (2) f_{ij}, the fixed cost of inspecting a batch after stage j, given that we last inspected the batch after stage i; and (3) g_{ij}, the variable per unit cost for inspecting an item after stage j, given that we last inspected the batch after stage i. The inspection costs at station j depend on when the batch was inspected last, say at station i, because the inspector needs to look for defects incurred at any of the intermediate stages $i + 1, i + 2, \ldots, j$.

We can formulate this inspection problem as a shortest path problem on a network with $(n + 1)$ nodes, numbered $0, 1, \ldots, n$. The network contains an arc (i, j) for each node pair i and j for which $i < j$. Figure 4.2 shows the network for an

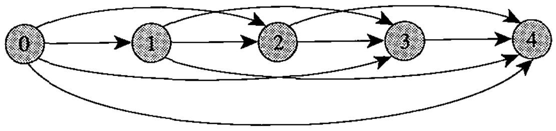

Figure 4.2 Shortest path network associated with the inspection problem.

inspection problem with four stations. Each path in the network from node 0 to node 4 defines an inspection plan. For example, the path 0–2–4 implies that we inspect the batches after the second and fourth stages. Letting $B(i) = B \prod_{k=1}^{i} (1 - \alpha_k)$ denote the expected number of nondefective units at the end of stage i, we associate the following cost c_{ij} with any arc (i, j) in the network:

$$c_{ij} = f_{ij} + B(i)g_{ij} + B(i) \sum_{k=i+1}^{j} p_k. \tag{4.2}$$

It is easy to see that c_{ij} denotes the total cost incurred in the stages $i + 1, i + 2, \ldots, j$; the first two terms on the right-hand side of (4.2) are the fixed and variable inspection costs, and the third term is the production cost incurred in these stages. This shortest path formulation permits us to solve the inspection application as a network flow problem.

Application 4.3 Knapsack Problem

In Section 3.3 we introduced the knapsack problem and formulated this classical operations research model as an integer program. For convenience, let us recall the underlying motivation for this problem. A hiker must decide which goods to include in her knapsack on a forthcoming trip. She must choose from among p objects: Object i has weight w_i (in pounds) and a utility u_i to the hiker. The objective is to maximize the utility of the hiker's trip subject to the weight limitation that she can carry no more than W pounds. In Section 3.3 we described a dynamic programming algorithm for solving this problem. Here we formulate the knapsack problem as a longest path problem on an acyclic network and then show how to transform the longest path problem into a shortest path problem. This application illustrates an intimate connection between dynamic programming and shortest path problems on acyclic networks. By making the appropriate identification between the stages and "states" of any dynamic program and the nodes of a network, we can formulate essentially all deterministic dynamic programming problems as equivalent shortest path problems. For these reasons, the range of applications of shortest path problems includes most applications of dynamic programming, which is a large and extensive field in its own right.

We illustrate our formulation using a knapsack problem with four items that have the weights and utilities indicated in the accompanying table:

j	1	2	3	4
u_j	40	15	20	10
w_j	4	2	3	1

Figure 4.3 shows the longest path formulation for this sample knapsack problem, assuming that the knapsack has a capacity of $W = 6$. The network in the formulation has several layers of nodes: It has one layer corresponding to each item and one layer corresponding to a source node s and another corresponding to a sink node t. The layer corresponding to an item i has $W + 1$ nodes, $i^0, i^1, \ldots, i^W$. Node

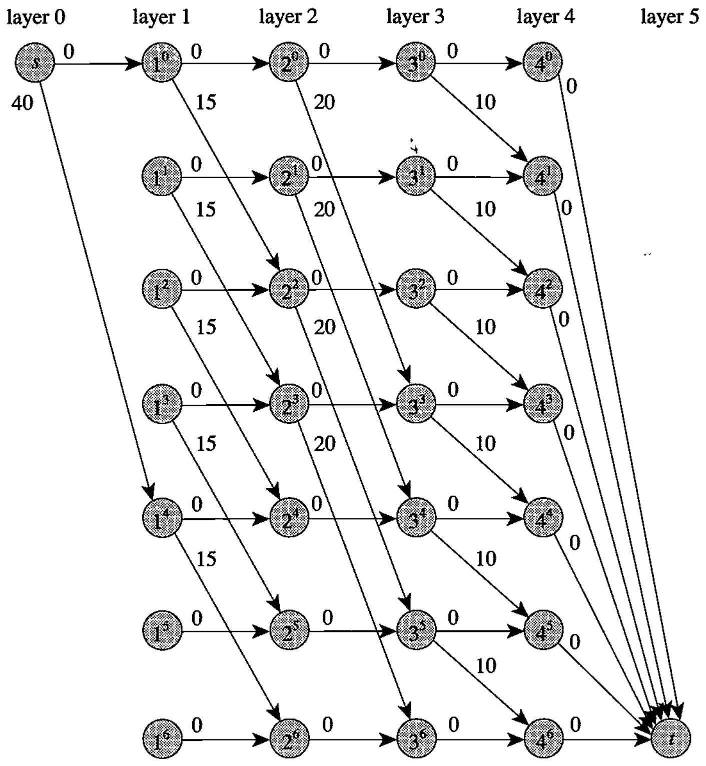

Figure 4.3 Longest path formulation of the knapsack problem.

i^k in the network signifies that the items $1, 2, \ldots, i$ have consumed k units of the knapsack's capacity. The node i^k has at most two outgoing arcs, corresponding to two decisions: (1) do not include item $(i + 1)$ in the knapsack, or (2) include item $i + 1$ in the knapsack. [Notice that we can choose the second of these alternatives only when the knapsack has sufficient spare capacity to accommodate item $(i + 1)$, i.e., $k + w_{i+1} \leq W$.] The arc corresponding to the first decision is $(i^k, (i + 1)^k)$ with zero utility and the arc corresponding to the second decision (provided that $k + w_{i+1} \leq W$) is $(i^k, (i + 1)^{k+w_{i+1}})$ with utility u_{i+1}. The source node has two incident arcs, $(s, 1^0)$ and $(s, 1^{w_1})$, corresponding to the choices of whether or not to include item 1 in an empty knapsack. Finally, we connect all the nodes in the layer corresponding to the last item to the sink node t with arcs of zero utility.

Every feasible solution of the knapsack problem defines a directed path from node s to node t; both the feasible solution and the path have the same utility. Conversely, every path from node s to node t defines a feasible solution to the knapsack problem with the same utility. For example, the path s–1^0–2^2–3^5–4^5–t implies the solution in which we include items 2 and 3 in the knapsack and exclude items 1 and 4. This correspondence shows that we can find the maximum utility selection of items by finding a maximum utility path, that is, a longest path in the network.

The longest path problem and the shortest path problem are closely related. We can transform the longest path problem to a shortest path problem by defining arc costs equal to the negative of the arc utilities. If the longest path problem contains any positive length directed cycle, the resulting shortest path problem contains a negative cycle and we cannot solve it using any of the techniques discussed in the book. However, if all directed cycles in the longest path problem have nonpositive lengths, then in the corresponding shortest path problem all directed cycles have nonnegative lengths and this problem can be solved efficiently. Notice that in the longest path formulation of the knapsack problem, the network is acyclic; so the resulting shortest path problem is efficiently solvable.

To conclude our discussion of this application, we offer a couple of concluding remarks concerning the relationship between shortest paths and dynamic programming. In Section 3.3 we solved the knapsack problem by using a recursive relationship for computing a quantity $d(i, j)$ that we defined as the maximum utility of selecting items if we restrict our selection to items 1 through i and impose a weight restriction of j. Note that $d(i, j)$ can be interpreted as the longest path length from node s to node i^j. Moreover, as we will see, the recursion that we used to solve the dynamic programming formulation of the knapsack problem is just a special implementation of one of the standard algorithms for solving shortest path problems on acyclic networks (we describe this algorithm in Section 4.4). This observation provides us with a concrete illustration of the meta statement that "(deterministic) dynamic programming is a special case of the shortest path problem."

Second, as we show in Section 4.4, shortest path problems on acyclic networks are *very* easy to solve—by methods that are *linear* in the number n of nodes and number m of arcs. Since the nodes of the network representation correspond to the "stages" and "states" of the dynamic programming formulation, the dynamic programming model will be easy to solve if the number of states and stages is not very large (i.e., do not grow exponentially fast in some underlying problem parameter).

Application 4.4 Tramp Steamer Problem

A tramp steamer travels from port to port carrying cargo and passengers. A voyage of the steamer from port i to port j earns p_{ij} units of profit and requires τ_{ij} units of time. The captain of the steamer would like to know which tour W of the steamer (i.e., a directed cycle) achieves the largest possible mean daily profit when we define the daily profit for any tour W by the expression

$$\mu(W) = \frac{\sum_{(i,j)\in W} p_{ij}}{\sum_{(i,j)\in W} \tau_{ij}}.$$

We assume that $\tau_{ij} \geq 0$ for every arc $(i, j) \in A$, and that $\sum_{(i,j)\in W} \tau_{ij} > 0$ for every directed cycle W in the network.

In Section 5.7 we study the tramp steamer problem. In this application we examine a more restricted version of the tramp steamer problem: The captain of the steamer wants to know whether some tour W will be able to achieve a mean daily profit greater than a specified threshold μ_0. We will show how to formulate this problem as a negative cycle detection problem. In this restricted version of the tramp steamer problem, we wish to determine whether the underlying network G contains a directed cycle W satisfying the following condition:

$$\frac{\sum_{(i,j)\in W} p_{ij}}{\sum_{(i,j)\in W} \tau_{ij}} > \mu_0.$$

By writing this inequality as $\sum_{(i,j)\in W} (\mu_0\tau_{ij} - p_{ij}) < 0$, we see that G contains a directed cycle W in G whose mean profit exceeds μ_0 if and only if the network contains a negative cycle when the cost of arc (i, j) is $(\mu_0\tau_{ij} - p_{ij})$. In Section 5.5 we show that label-correcting algorithms for solving the shortest path problem are able to detect negative cycles, which implies that we can solve this restricted version of the tramp steamer problem by applying a shortest path algorithm.

Application 4.5 System of Difference Constraints

In some linear programming applications, with constraints of the form $\mathcal{A}x \leq b$, the $n \times m$ constraint matrix $\mathcal{A}$ contains one $+1$ and one -1 in each row; all the other entries are zero. Suppose that the kth row has a $+1$ entry in column j_k and a -1 entry in column i_k; the entries in the vector b have arbitrary signs. Then this linear program defines the following set of m *difference constraints* in the n variables $x = (x(1), x(2), \ldots, x(n))$:

$$x(j_k) - x(i_k) \leq b(k) \qquad \text{for each } k = 1, \ldots, m. \tag{4.3}$$

We wish to determine whether the system of difference constraints given by (4.3) has a feasible solution, and if so, we want to identify a feasible solution. This model arises in a variety of applications; in Application 4.6 we describe the use of this model in the telephone operator scheduling, and in Application 19.6 we describe the use of this model in the scaling of data.

Each system of difference constraints has an associated graph G, which we call a *constraint graph*. The constraint graph has n nodes corresponding to the n variables and m arcs corresponding to the m difference constraints. We associate an arc (i_k, j_k) of length $b(k)$ in G with the constraint $x(j_k) - x(i_k) \le b(k)$. As an example, consider the following system of constraints whose corresponding graph is shown in Figure 4.4(a):

$$x(3) - x(4) \le 5, \tag{4.4a}$$

$$x(4) - x(1) \le -10, \tag{4.4b}$$

$$x(1) - x(3) \le 8, \tag{4.4c}$$

$$x(2) - x(1) \le -11, \tag{4.4d}$$

$$x(3) - x(2) \le 2. \tag{4.4e}$$

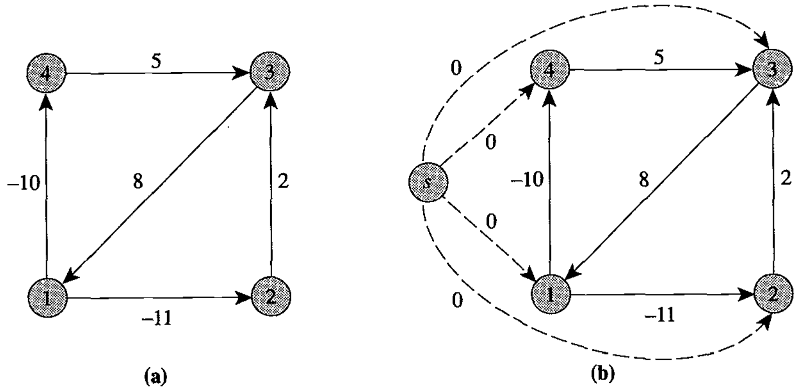

Figure 4.4 Graph corresponding to a system of difference constraints.

In Section 5.2 we show that the constraints (4.4) are identical with the optimality conditions for the shortest path problem in Figure 4.4(a) and that we can satisfy these conditions if and only if the network contains no negative (cost) cycle. The network shown in Figure 4.4(a) contains a negative cycle 1–2–3 of length -1, and the corresponding constraints [i.e., $x(2) - x(1) \le -11$, $x(3) - x(2) \le 2$, and $x(1) - x(3) \le 8$] are inconsistent because summing these constraints yields the invalid inequality $0 \le -1$.

As noted previously, we can detect the presence of a negative cycle in a network by using the label-correcting algorithms described in Chapter 5. The label-correcting algorithms do require that all the nodes are reachable by a directed path from some node, which we use as the source node for the shortest path problem. To satisfy this requirement, we introduce a new node s and join it to all the nodes in the network with arcs of zero cost. For our example, Figure 4.4(b) shows the modified network. Since all the arcs incident to node s are directed out of this node, node s is not contained in any directed cycle, so the modification does not create any new directed cycles and so does not introduce any cycles with negative costs. The label-correcting algorithms either indicate the presence of a negative cycle or provide the shortest path distances. In the former case the system of difference constraints has no solution, and in the latter case the shortest path distances constitute a solution of (4.4).

Application 4.6 Telephone Operator Scheduling

As an application of the system of difference constraints, consider the following telephone operator scheduling problem. A telephone company needs to schedule operators around the clock. Let $b(i)$ for $i = 0, 1, 2, \ldots, 23$, denote the minimum number of operators needed for the ith hour of the day [here $b(0)$ denotes number of operators required between midnight and 1 A.M.]. Each telephone operator works in a shift of 8 consecutive hours and a shift can begin at any hour of the day. The telephone company wants to determine a "cyclic schedule" that repeats daily (i.e., the number of operators assigned to the shift starting at 6 A.M. and ending at 2 P.M. is the same for each day). The optimization problem requires that we identify the fewest operators needed to satisfy the minimum operator requirement for each hour of the day. Letting y_i denote the number of workers whose shift begins at the ith hour, we can state the telephone operator scheduling problem as the following optimization model:

$$\text{Minimize} \sum_{i=0}^{23} y_i \tag{4.5a}$$

subject to

$$y_{i-7} + y_{i-6} + \cdots + y_i \geq b(i) \qquad \text{for all } i = 8 \text{ to } 23, \tag{4.5b}$$

$$y_{17+i} + \cdots + y_{23} + y_0 + \cdots + y_i \geq b(i) \qquad \text{for all } i = 0 \text{ to } 7, \tag{4.5c}$$

$$y_i \geq 0 \qquad \text{for all } i = 0 \text{ to } 23. \tag{4.5d}$$

Notice that this linear program has a very special structure because the associated constraint matrix contains only 0 and 1 elements and the 1's in each row appear consecutively. In this application we study a restricted version of the telephone operator scheduling problem: We wish to determine whether some feasible schedule uses p or fewer operators. We convert this restricted problem into a system of difference constraints by redefining the variables. Let $x(0) = y_0$, $x(1) = y_0 + y_1$, $x(2) = y_0 + y_1 + y_2, \ldots$, and $x(23) = y_0 + y_2 + \cdots + y_{23} = p$. Now notice that we can rewrite each constraint in (4.5b) as

$$x(i) - x(i-8) \geq b(i) \qquad \text{for all } i = 8 \text{ to } 23, \tag{4.6a}$$

and each constraints in (4.5c) as

$$\begin{aligned} &x(23) - x(16+i) + x(i) \\ &\qquad = p - x(16+i) + x(i) \geq b(i) \qquad \text{for all } i = 0 \text{ to } 7. \end{aligned} \tag{4.6b}$$

Finally, the nonnegativity constraints (4.5d) become

$$x(i) - x(i-1) \geq 0. \tag{4.6c}$$

By virtue of this transformation, we have reduced the restricted version of the telephone operator scheduling problem into a problem of finding a feasible solution of the system of difference constraints. We discuss a solution method for the general problem in Exercise 4.12. Exercise 9.9 considers a further generalization that incorporates costs associated with various shifts.

In the telephone operator scheduling problem, the rows of the underlying optimization model (in the variables y) satisfy a "wraparound consecutive 1's property"; that is, the variables in each row have only 0 and 1 coefficients and all of the variables with 1 coefficients are consecutive (if we consider the first and last variables to be consecutive). In the telephone operator scheduling problem, each row has exactly eight variables with coefficients of value 1. In general, as long as any optimization model satisfies the wraparound consecutive 1's property, even if the rows have different numbers of variables with coefficients of value 1, the transformation we have described would permit us to model the problem as a network flow model.

4.3 TREE OF SHORTEST PATHS

In the shortest path problem, we wish to determine a shortest path from the source node to all other $(n - 1)$ nodes. How much storage would we need to store these paths? One naive answer would be an upper bound of $(n - 1)^2$ since each path could contain at most $(n - 1)$ arcs. Fortunately, we need not use this much storage: $(n - 1)$ storage locations are sufficient to represent all these paths. This result follows from the fact that we can always find a directed out-tree rooted from the source with the property that the unique path from the source to any node is a shortest path to that node. For obvious reasons we refer to such a tree as a *shortest path tree.* Each shortest path algorithm discussed in this book is capable of determining this tree as it computes the shortest path distances. The existence of the shortest path tree relies on the following property.

Property 4.1. *If the path $s = i_1 - i_2 - \cdots - i_h = k$ is a shortest path from node s to node k, then for every $q = 2, 3, \ldots, h - 1$, the subpath $s = i_1 - i_2 - \cdots - i_q$ is a shortest path from the source node to node i_q.*

This property is fairly easy to establish. In Figure 4.5 we assume that the shortest path P_1–P_3 from node s to node k passes through some node p, but the subpath P_1 up to node p is not a shortest path to node p; suppose instead that path P_2 is a shorter path to node p. Notice that P_2–P_3 is a directed walk whose length is less than that of path P_1–P_3. Also, notice that any directed walk from node s to node k decomposes into a directed path plus some directed cycles (see Exercise 3.51), and these cycles, by our assumption, must have nonnegative length. As a result, some directed path from node s to node k is shorter than the path P_1–P_3, contradicting its optimality.

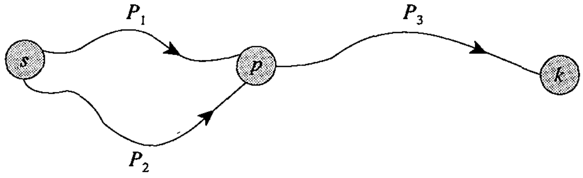

Figure 4.5 Proving Property 4.1.

Let $d(\cdot)$ denote the shortest path distances. Property 4.1 implies that if P is a shortest path from the source node to some node k, then $d(j) = d(i) + c_{ij}$ for every arc $(i, j) \in P$. The converse of this result is also true; that is, if $d(j) = d(i) + c_{ij}$

for every arc in a directed path P from the source to node k, then P must be a shortest path. To establish this result, let $s = i_1 - i_2 - \cdots - i_h = k$ be the node sequence in P. Then

$$d(k) = d(i_h) = (d(i_h) - d(i_{h-1})) + (d(i_{h-1}) - d(i_{h-2})) + \cdots + (d(i_2) - d(i_1)),$$

where we use the fact that $d(i_1) = 0$. By assumption, $d(j) - d(i) = c_{ij}$ for every arc $(i, j) \in P$. Using this equality we see that

$$d(k) = c_{i_{h-1}i_h} + c_{i_{h-2}i_{h-1}} + \cdots + c_{i_1i_2} = \sum_{(i,j)\in P} c_{ij}.$$

Consequently, P is a directed path from the source node to node k of length $d(k)$. Since, by assumption, $d(k)$ is the shortest path distance to node k, P must be a shortest path to node k. We have thus established the following result.

Property 4.2. *Let the vector d represent the shortest path distances. Then a directed path P from the source node to node k is a shortest path if and only if* $d(j) = d(i) + c_{ij}$ *for every arc* $(i, j) \in P$.

We are now in a position to prove the existence of a shortest path tree. Since only a finite number of paths connect the source to every node, the network contains a shortest path to every node. Property 4.2 implies that we can always find a shortest path from the source to every other node satisfying the property that for every arc (i, j) on the path, $d(j) = d(i) + c_{ij}$. Therefore, if we perform a breadth-first search of the network using the arcs satisfying the equality $d(j) = d(i) + c_{ij}$, we must be able to reach every node. The breadth-first search tree contains a unique path from the source to every other node, which by Property 4.2 must be a shortest path to that node.

4.4 SHORTEST PATH PROBLEMS IN ACYCLIC NETWORKS

Recall that a network is said to be *acyclic* if it contains no directed cycle. In this section we show how to solve the shortest path problem on an acyclic network in $O(m)$ time even though the arc lengths might be negative. Note that no other algorithm for solving the shortest path problem on acyclic networks could be any faster (in terms of the worst-case complexity) become any algorithm for solving the problem must examine every arc, which itself would take $O(m)$ time.

Recall from Section 3.4 that we can always number (or order) nodes in an acyclic network $G = (N, A)$ in $O(m)$ time so that $i < j$ for every arc $(i, j) \in A$. This ordering of nodes is called a *topological ordering*. Conceptually, once we have determined the topological ordering, the shortest path problem is quite easy to solve by a simple dynamic programming algorithm. Suppose that we have determined the shortest path distances $d(i)$ from the source node to nodes $i = 1, 2, \ldots, k - 1$. Consider node k. The topological ordering implies that all the arcs directed into this node emanate from one of the nodes 1 through $k - 1$. By Property 4.1, the shortest path to node k is composed of a shortest path to one of the nodes $i = 1, 2, \ldots, k - 1$ together with the arc (i, k). Therefore, to compute the shortest path distance to node k, we need only select the minimum of $d(i) + c_{ik}$ for all incoming arcs (i, k). This algorithm is a *pulling* algorithm in that to find the shortest path distance to any node, it "pulls" shortest path distances forward from lower-numbered nodes. Notice that to implement this algorithm, we need to access conveniently all the arcs directed into each node. Since we frequently store the adjacency list $A(i)$ of each node i, which gives the arcs emanating out of a node, we might also like to implement a *reaching* algorithm that propagates information from each node to higher-indexed nodes, and so uses the usual adjacency list. We next describe one such algorithm.

We first set $d(s) = 0$ and the remaining distance labels to a very large number. Then we examine nodes in the topological order and for each node i being examined, we scan arcs in $A(i)$. If for any arc $(i, j) \in A(i)$, we find that $d(j) > d(i) + c_{ij}$, then we set $d(j) = d(i) + c_{ij}$. When the algorithm has examined all the nodes once in this order, the distance labels are optimal.

We use induction to show that whenever the algorithm examines a node, its distance label is optimal. Suppose that the algorithm has examined nodes $1, 2, \ldots, k$ and their distance labels are optimal. Consider the point at which the algorithm examines node $k + 1$. Let the shortest path from the source to node $k + 1$ be $s = i_1 - i_2 - \cdots - i_h - (k + 1)$. Observe that the path $i_1 - i_2 - \cdots - i_h$ must be a shortest path from the source to node i_h (by Property 4.1). The facts that the nodes are topologically ordered and that the arc $(i_h, k + 1) \in A$ imply that $i_h \in \{1, 2, \ldots, k\}$ and, by the inductive hypothesis, the distance label of node i_h is equal to the length of the path $i_1 - i_2 - \cdots - i_h$. Consequently, while examining node i_h, the algorithm must have scanned the arc $(i_h, k + 1)$ and set the distance label of node $(k + 1)$ equal to the length of the path $i_1 - i_2 - \cdots - i_h - (k + 1)$. Therefore, when the algorithm examines the node $k + 1$, its distance label is optimal. The following result is now immediate.

Theorem 4.3. *The reaching algorithm solves the shortest path problem on acyclic networks in $O(m)$ time.*

In this section we have seen how we can solve the shortest path problem on acyclic networks very efficiently using the simplest possible algorithm. Unfortunately, we cannot apply this one-pass algorithm, and examine each node and each arc exactly once, for networks containing cycles; nevertheless, we can utilize the same basic reaching strategy used in this algorithm and solve any shortest path problem with nonnegative arc lengths using a modest additional amount of work. As we will see, we incur additional work because we no longer have a set order for examining the nodes, so at each step we will need to investigate several nodes in order to determine which node to reach out from next.

4.5 DIJKSTRA'S ALGORITHM

As noted previously, Dijkstra's algorithm finds shortest paths from the source node s to all other nodes in a network with nonnegative arc lengths. Dijkstra's algorithm maintains a distance label $d(i)$ with each node i, which is an upper bound on the shortest path length to node i. At any intermediate step, the algorithm divides the nodes into two groups: those which it designates as *permanently labeled* (or permanent) and those it designates as *temporarily labeled* (or temporary). The distance label to any permanent node represents the shortest distance from the source to that node. For any temporary node, the distance label is an upper bound on the shortest path distance to that node. The basic idea of the algorithm is to fan out from node s and permanently label nodes in the order of their distances from node s. Initially, we give node s a permanent label of zero, and each other node j a temporary label equal to ∞. At each iteration, the label of a node i is its shortest distance from the source node along a path whose internal nodes (i.e., nodes other than s or the node i itself) are all permanently labeled. The algorithm selects a node i with the minimum temporary label (breaking ties arbitrarily), makes it permanent, and reaches out from that node—that is, scans arcs in $A(i)$ to update the distance labels of adjacent nodes. The algorithm terminates when it has designated all nodes as permanent. The correctness of the algorithm relies on the key observation (which we prove later) that we can always designate the node with the minimum temporary label as permanent.

Dijkstra's algorithm maintains a directed out-tree T rooted at the source that spans the nodes with finite distance labels. The algorithm maintains this tree using predecessor indices [i.e., if $(i, j) \in T$, then $pred(j) = i$]. The algorithm maintains the invariant property that every tree arc (i, j) satisfies the condition $d(j) = d(i) + c_{ij}$ with respect to the current distance labels. At termination, when distance labels represent shortest path distances, T is a shortest path tree (from Property 4.2).

Figure 4.6 gives a formal algorithmic description of Dijkstra's algorithm.

In Dijkstra's algorithm, we refer to the operation of selecting a minimum temporary distance label as a *node selection* operation. We also refer to the operation of checking whether the current labels for nodes i and j satisfy the condition $d(j) > d(i) + c_{ij}$ and, if so, then setting $d(j) = d(i) + c_{ij}$ as a *distance update* operation.

We illustrate Dijkstra's algorithm using the numerical example given in Figure 4.7(a). The algorithm permanently labels the nodes 3, 4, 2, and 5 in the given sequence: Figure 4.7(b) to (e) illustrate the operations for these iterations. Figure 4.7(f) shows the shortest path tree for this example.

```
algorithm Dijkstra;
begin
    $S := \emptyset$; $\bar{S} := N$;
    $d(i) := \infty$ for each node $i \in N$;
    $d(s) := 0$ and $pred(s) := 0$;
    while $|S| < n$ do
    begin
        let $i \in \bar{S}$ be a node for which $d(i) = \min\{d(j) : j \in \bar{S}\}$;
        $S := S \cup \{i\}$;
        $\bar{S} := \bar{S} - \{i\}$;
        for each $(i, j) \in A(i)$ do
            if $d(j) > d(i) + c_{ij}$ then $d(j) := d(i) + c_{ij}$ and $pred(j) := i$;
    end;
end;
```

Figure 4.6 Dijkstra's algorithm.

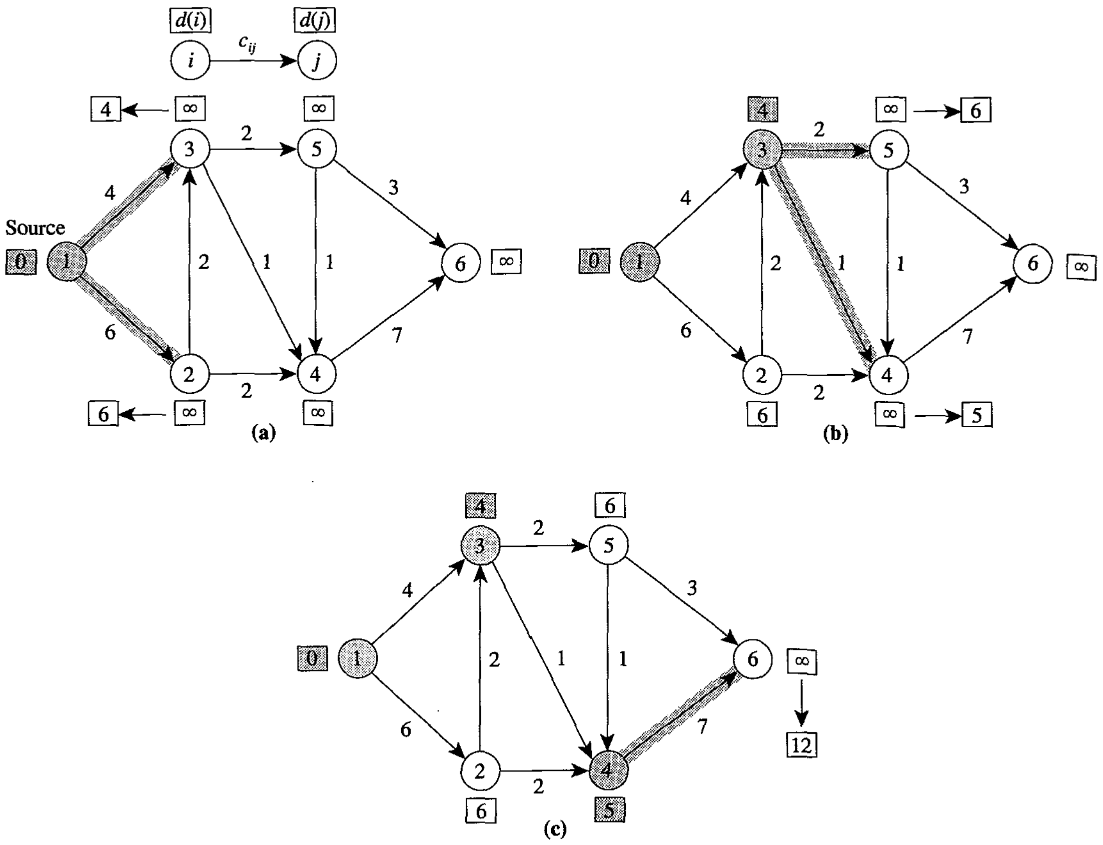

Figure 4.7 Illustrating Dijkstra's algorithm.

Correctness of Dijkstra's Algorithm

We use inductive arguments to establish the validity of Dijkstra's algorithm. At any iteration, the algorithm has partitioned the nodes into two sets, S and $\bar{S}$. Our induction hypothesis are (1) that the distance label of each node in S is optimal, and (2) that the distance label of each node in $\bar{S}$ is the shortest path length from the source provided that each internal node in the path lies in S. We perform induction on the cardinality of the set S.

To prove the first inductive hypothesis, recall that at each iteration the algorithm transfers a node i in the set $\bar{S}$ with smallest distance label to the set S. We need to show that the distance label $d(i)$ of node i is optimal. Notice that by our induction hypothesis, $d(i)$ is the length of a shortest path to node i among all paths that do not contain any node in $\bar{S}$ as an internal node. We now show that the length of any path from s to i that contains some nodes in $\bar{S}$ as an internal node will be at least $d(i)$. Consider any path P from the source to node i that contains at least one node in $\bar{S}$ as an internal node. The path P can be decomposed into two segments P_1 and P_2: the path segment P_1 does not contain any node in $\bar{S}$ as an internal node, but terminates at a node k in $\bar{S}$ (see Figure 4.8). By the induction hypothesis, the length of the path P_1 is at least $d(k)$ and since node i is the smallest distance label

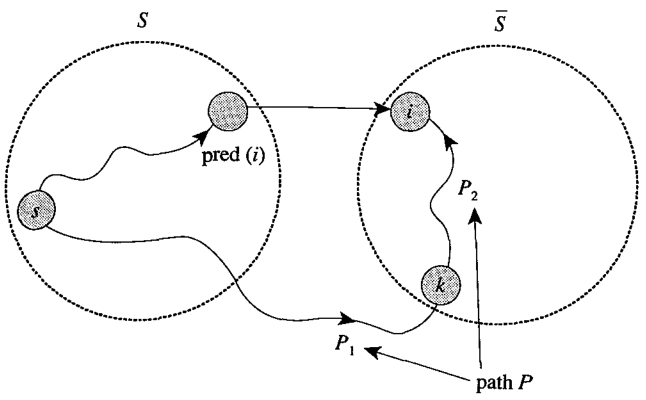

Figure 4.8 Proving Dijkstra's algorithm.

in $\bar{S}$, $d(k) \geq d(i)$. Therefore, the path segment P_1 has length at least $d(i)$. Furthermore, since all arc lengths are nonnegative, the length of the path segment P_2 is nonnegative. Consequently, length of the path P is at least $d(i)$. This result establishes the fact that $d(i)$ is the shortest path length of node i from the source node.

We next show that the algorithm preserves the second induction hypothesis. After the algorithm has labeled a new node i permanently, the distance labels of some nodes in $\bar{S} - \{i\}$ might decrease, because node i could become an internal node in the tentative shortest paths to these nodes. But recall that after permanently labeling node i, the algorithm examines each arc $(i, j) \in A(i)$ and if $d(j) > d(i) + c_{ij}$, then it sets $d(j) = d(i) + c_{ij}$ and pred$(j) = i$. Therefore, after the distance update operation, by the induction hypothesis the path from node j to the source node defined by the predecessor indices satisfies Property 4.2 and so the distance label of each node in $\bar{S} - \{i\}$ is the length of a shortest path subject to the restriction that each internal node in the path must belong to $S \cup \{i\}$.

Running Time of Dijkstra's Algorithm

We now study the worst-case complexity of Dijkstra's algorithm. We might view the computational time for Dijkstra's algorithm as allocated to the following two basic operations:

1. *Node selections.* The algorithm performs this operation n times and each such operation requires that it scans each temporarily labeled node. Therefore, the total node selection time is $n + (n - 1) + (n - 2) + \cdots + 1 = O(n^2)$.
2. *Distance updates.* The algorithm performs this operation $|A(i)|$ times for node i. Overall, the algorithm performs this operation $\sum_{i \in N} |A(i)| = m$ times. Since each distance update operation requires $O(1)$ time, the algorithm requires $O(m)$ total time for updating all distance labels.

We have established the following result.

Theorem 4.4. *Dijkstra's algorithm solves the shortest path problem in $O(n^2)$ time.*

The $O(n^2)$ time bound for Dijkstra's algorithm is the best possible for completely dense networks [i.e., $m = \Omega(n^2)$], but can be improved for sparse networks. Notice that the times required by the node selections and distance updates are not balanced. The node selections require a total of $O(n^2)$ time, and the distance updates require only $O(m)$ time. Researchers have attempted to reduce the node selection time without substantially increasing the time for updating the distances. Consequently, they have, using clever data structures, suggested several implementations of the algorithm. These implementations have either dramatically reduced the running time of the algorithm in practice or improved its worst-case complexity. In Section 4.6 we describe Dial's algorithm, which is an excellent implementation of Dijkstra's algorithm in practice. Sections 4.7 and 4.8 describe several implementations of Dijkstra's algorithm with improved worst-case complexity.

Reverse Dijkstra's Algorithm

In the (forward) Dijkstra's algorithm, we determine a shortest path from node s to every other node in $N - \{s\}$. Suppose that we wish to determine a shortest path from every node in $N - \{t\}$ to a sink node t. To solve this problem, we use a slight modification of Dijkstra's algorithm, which we refer to as the *reverse Dijkstra's algorithm*. The reverse Dijkstra's algorithm maintains a distance $d'(j)$ with each node j, which is an upper bound on the shortest path length from node j to node t. As before, the algorithm designates a set of nodes, say S', as permanently labeled and the remaining set of nodes, say $\bar{S}'$, as temporarily labeled. At each iteration, the algorithm designates a node with the minimum temporary distance label, say $d'(j)$, as permanent. It then examines each incoming arc (i, j) and modifies the distance label of node i to $\min\{d'(i), c_{ij} + d'(j)\}$. The algorithm terminates when all the nodes have become permanently labeled.

Bidirectional Dijkstra's Algorithm

In some applications of the shortest path problem, we need not determine a shortest path from node s to every other node in the network. Suppose, instead, that we want to determine a shortest path from node s to a specified node t. To solve this problem and eliminate some computations, we could terminate Dijkstra's algorithm as soon as it has selected t from $\bar{S}$ (even though some nodes are still temporarily labeled). The bidirectional Dijkstra's algorithm, which we describe next, allows us to solve this problem even faster in practice (though not in the worst case).

In the bidirectional Dijkstra's algorithm, we simultaneously apply the forward Dijkstra's algorithm from node s and reverse Dijkstra's algorithm from node t. The algorithm alternatively designates a node in $\bar{S}$ and a node in $\bar{S}'$ as permanent until both the forward and reverse algorithms have permanently labeled the same node, say node k (i.e., $S \cap S' = \{k\}$). At this point, let $P(i)$ denote the shortest path from node s to node $i \in S$ found by the forward Dijkstra's algorithm, and let $P'(j)$ denote the shortest path from node $j \in S'$ to node t found by the reverse Dijkstra's algorithm. A straightforward argument (see Exercise 4.52) shows that the shortest path from node s to node t is either the path $P(k) \cup P'(k)$ or a path $P(i) \cup \{(i, j)\} \cup P'(j)$ for some arc (i, j), $i \in S$ and $j \in S'$. This algorithm is very efficient because it tends to permanently label few nodes and hence never examines the arcs incident to a large number of nodes.

4.6 DIAL'S IMPLEMENTATION

The bottleneck operation in Dijkstra's algorithm is node selection. To improve the algorithm's performance, we need to address the following question. Instead of scanning all temporarily labeled nodes at each iteration to find the one with the minimum distance label, can we reduce the computation time by maintaining distances in some sorted fashion? Dial's algorithm tries to accomplish this objective, and reduces the algorithm's computation time in practice, using the following fact:

Property 4.5. *The distance labels that Dijkstra's algorithm designates as permanent are nondecreasing.*

This property follows from the fact that the algorithm permanently labels a node i with a smallest temporary label $d(i)$, and while scanning arcs in $A(i)$ during the distance update operations, never decreases the distance label of any temporarily labeled node below $d(i)$ because arc lengths are nonnegative.

Dial's algorithm stores nodes with finite temporary labels in a sorted fashion. It maintains $nC + 1$ sets, called *buckets*, numbered $0, 1, 2, \ldots, nC$: Bucket k stores all nodes with temporary distance label equal to k. Recall that C represents the largest arc length in the network, and therefore nC is an upper bound on the distance label of any finitely labeled node. We need not store nodes with infinite temporary distance labels in any of the buckets—we can add them to a bucket when they first receive a finite distance label. We represent the content of bucket k by the set *content(k)*.

In the node selection operation, we scan buckets numbered $0, 1, 2, \ldots,$ until we identify the first nonempty bucket. Suppose that bucket k is the first nonempty bucket. Then each node in *content(k)* has the minimum distance label. One by one, we delete these nodes from the bucket, designate them as permanently labeled, and scan their arc lists to update the distance labels of adjacent nodes. Whenever we update the distance label of a node i from d_1 to d_2, we move node i from *content*(d_1) to *content*(d_2). In the next node selection operation, we resume the scanning of buckets numbered $k + 1, k + 2, \ldots$ to select the next nonempty bucket. Property 4.5 implies that the buckets numbered $0, 1, 2, \ldots, k$ will always be empty in the subsequent iterations and the algorithm need not examine them again.

As a data structure for storing the content of the buckets, we store each set *content(k)* as a doubly linked list (see Appendix A). This data structure permits us to perform each of the following operations in $O(1)$ time: (1) checking whether a bucket is empty or nonempty, (2) deleting an element from a bucket, and (3) adding an element to a bucket. With this data structure, the algorithm requires $O(1)$ time for each distance update, and thus a total of $O(m)$ time for all distance updates. The bottleneck operation in this implementation is scanning $nC + 1$ buckets during node selections. Consequently, the running time of Dial's algorithm is $O(m + nC)$.

Since Dial's algorithm uses $nC + 1$ buckets, its memory requirements can be prohibitively large. The following fact allows us to reduce the number of buckets to $C + 1$.

Property 4.6. *If $d(i)$ is the distance label that the algorithm designates as permanent at the beginning of an iteration, then at the end of that iteration, $d(j) \leq d(i) + C$ for each finitely labeled node j in $\bar{S}$.*

This fact follows by noting that (1) $d(l) \leq d(i)$ for each node $l \in S$ (by Property 4.5), and (2) for each finitely labeled node j in $\bar{S}$, $d(j) = d(l) + c_{lj}$ for some node $l \in S$ (by the property of distance updates). Therefore, $d(j) = d(l) + c_{lj} \leq d(i) + C$. In other words, all finite temporary labels are bracketed from below by $d(i)$ and from above by $d(i) + C$. Consequently, $C + 1$ buckets suffice to store nodes with finite temporary distance labels.

Dial's algorithm uses $C + 1$ buckets numbered $0, 1, 2, \ldots, C$, which we might view as arranged in a circular fashion as in Figure 4.9. We store a temporarily labeled node j with distance label $d(j)$ in the bucket $d(j) \bmod (C + 1)$. Consequently, during the entire execution of the algorithm, bucket k stores nodes with temporary distance labels k, $k + (C + 1)$, $k + 2(C + 1)$, and so on; however, because of Property 4.6, at any point in time, this bucket will hold only nodes with the same distance label. This storage scheme also implies that if bucket k contains a node with the minimum distance label, then buckets $k + 1, k + 2, \ldots, C, 0, 1, 2, \ldots, k - 1$ store nodes in increasing values of the distance labels.

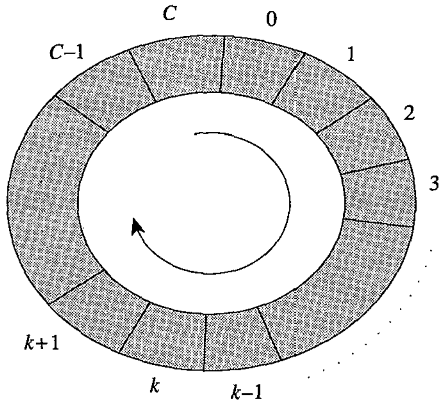

Figure 4.9 Bucket arrangement in Dial's algorithm.

Dial's algorithm examines the buckets sequentially, in a wraparound fashion, to identify the first nonempty bucket. In the next iteration, it reexamines the buckets starting at the place where it left off previously. A potential disadvantage of Dial's algorithm compared to the original $O(n^2)$ implementation of Dijkstra's algorithm is that it requires a large amount of storage when C is very large. In addition, because the algorithm might wrap around as many as $n - 1$ times, the computational time could be large. The algorithm runs in $O(m + nC)$ time, which is not even polynomial, but rather, is pseudopolynomial. For example, if $C = n^4$, the algorithm runs in $O(n^5)$ time, and if $C = 2^n$, the algorithm requires exponential time in the worst case. However, the algorithm typically does not achieve the bound of $O(m + nC)$ time. For most applications, C is modest in size, and the number of passes through all of the buckets is much less than $n - 1$. Consequently, the running time of Dial's algorithm is much better than that indicated by its worst-case complexity.

4.7 HEAP IMPLEMENTATIONS

This section requires that the reader is familiar with heap data structures. We refer an unfamiliar reader to Appendix A, where we describe several such data structures.

A *heap* (or *priority queue*) is a data structure that allows us to perform the following operations on a collection H of *objects*, each with an associated real number called its *key*. More properly, a priority queue is an abstract data type, and is usually implemented using one of several heap data structures. However, in this treatment we are using the words "heap" and "priority queue" interchangeably.

create-heap(H). Create an empty heap.

find-min(i, H). Find and return an object i of minimum key.

insert(i, H). Insert a new object i with a predefined key.

decrease-key(value, i, H). Reduce the key of an object i from its current value to *value*, which must be smaller than the key it is replacing.

delete-min(i, H). Delete an object i of minimum key.

If we implement Dijkstra's algorithm using a heap, H would be the collection of nodes with finite temporary distance labels and the key of a node would be its distance label. Using a heap, we could implement Dijkstra's algorithm as described in Figure 4.10.

As is clear from this description, the heap implementation of Dijkstra's algorithm performs the operations find-min, delete-min, and insert at most n times and the operation decrease-key at most m times. We now analyze the running times of Dijkstra's algorithm implemented using different types of heaps: binary heaps, d-heaps, Fibonacci heaps, and another data structure suggested by Johnson. We describe the first three of these four data structures in Appendix A and provide a reference for the fourth data structure in the reference notes.

```
algorithm heap-Dijkstra;
begin
    create-heap(H);
    $d(j) := \infty$ for all $j \in N$;
    $d(s) := 0$ and pred($s$) := 0;
    insert($s$, H);
    while $H \neq \emptyset$ do
    begin
        find-min($i$, H);
        delete-min($i$, H);
        for each $(i, j) \in A(i)$ do
        begin
            value := $d(i) + c_{ij}$;
            if $d(j) >$ value then
                if $d(j) = \infty$ then $d(j)$ := value, pred($j$) := $i$, and insert ($j$, H)
                else set $d(j)$ := value, pred($j$) := $i$, and decrease-key(value, $i$, H);
        end;
    end;
end;
```

Figure 4.10 Dijkstra's algorithm using a heap.

Binary heap implementation. As discussed in Appendix A, a binary heap data structure requires $O(\log n)$ time to perform insert, decrease-key, and delete-min, and it requires $O(1)$ time for the other heap operations. Consequently, the binary heap version of Dijkstra's algorithm runs in $O(m \log n)$ time. Notice that the binary heap implementation is slower than the original implementation of Dijkstra's algorithm for completely dense networks [i.e., $m = \Omega(n^2)$], but is faster when $m = O(n^2/\log n)$.

***d*-Heap implementation.** For a given parameter $d \geq 2$, the d-heap data structure requires $O(\log_d n)$ time to perform the insert and decrease-key operations; it requires $O(d \log_d n)$ time for delete-min, and it requires $O(1)$ steps for the other heap operations. Consequently, the running time of this version of Dijkstra's algorithm is $O(m \log_d n + nd \log_d n)$. To obtain an optimal choice of d, we equate the two terms (see Section 3.2), giving $d = \max\{2, \lceil m/n \rceil\}$. The resulting running time is $O(m \log_d n)$. Observe that for very sparse networks [i.e., $m = O(n)$], the running time of the d-heap implementation is $O(n \log n)$. For nonsparse networks [i.e., $m = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$], the running time of d-heap implementation is $O(m \log_d n) = O((m \log n)/(\log d)) = O((m \log n)/(\log n^{\epsilon})) = O((m \log n)/(\epsilon \log n)) = O(m/\epsilon) = O(m)$. The last equality is true since ϵ is a constant. Thus the running time is $O(m)$, which is optimal.

Fibonacci heap implementation. The Fibonacci heap data structure performs every heap operation in $O(1)$ amortized time except delete-min, which requires $O(\log n)$ time. Consequently the running time of this version of Dijkstra's algorithm is $O(m + n \log n)$. This time bound is consistently better than that of binary heap and d-heap implementations for all network densities. This implementation is also currently the best strongly polynomial-time algorithm for solving the shortest path problem.

Johnson's implementation. Johnson's data structure (see the reference notes) is applicable only when all arc lengths are integer. This data structure requires $O(\log \log C)$ time to perform each heap operation. Consequently, this implementation of Dijkstra's algorithm runs in $O(m \log \log C)$ time.

We next discuss one more heap implementation of Dijkstra's algorithm, known as the *radix heap implementation*. The radix heap implementation is one of the more recent implementations; its running time is $O(m + n \log(nC))$.

4.8 RADIX HEAP IMPLEMENTATION

The radix heap implementation of Dijkstra's algorithm is a hybrid of the original $O(n^2)$ implementation and Dial's implementation (the one that uses $nC + 1$ buckets). These two implementations represent two extremes. The original implementation considers all the temporarily labeled nodes together (in one large bucket, so to speak) and searches for a node with the smallest label. Dial's algorithm uses a large number of buckets and separates nodes by storing any two nodes with different labels in different buckets. The radix heap implementation improves on these methods by adopting an intermediate approach: It stores many, but not all, labels in a bucket. For example, instead of storing only nodes with a temporary label k in the kth bucket, as in Dial's implementation, we might store temporary labels from $100k$ to $100k + 99$ in bucket k. The different temporary labels that can be stored in a bucket make up the *range* of the bucket; the cardinality of the range is called its *width*. For the preceding example, the range of bucket k is $[100k, 100k + 99]$ and its width is 100. Using widths of size k permits us to reduce the number of buckets needed by a factor of k. But to find the smallest distance label, we need to search all of the elements in the smallest indexed nonempty bucket. Indeed, if k is arbitrarily large, we need only one bucket, and the resulting algorithm reduces to Dijkstra's original implementation.

Using a width of 100, say, for each bucket reduces the number of buckets, but still requires us to search through the lowest-numbered nonempty bucket to find the node with minimum temporary label. If we could devise a variable width scheme, with a width of 1 for the lowest-numbered bucket, we could conceivably retain the advantages of both the wide bucket and narrow bucket approaches. The radix heap algorithm we consider next uses variable widths and changes the ranges dynamically. In the version of the radix heap that we present:

1. The widths of the buckets are 1, 1, 2, 4, 8, 16, . . . , so that the number of buckets needed is only $O(\log(nC))$.
2. We dynamically modify the ranges of the buckets and we reallocate nodes with temporary distance labels in a way that stores the minimum distance label in a bucket whose width is 1.

Property 1 allows us to maintain only $O(\log(nC))$ buckets and thereby overcomes the drawback of Dial's implementation of using too many buckets. Property 2 permits us, as in Dial's algorithm, to avoid the need to search the entire bucket to find a node with the minimum distance label. When implemented in this way, this version of the radix heap algorithm has a running time of $O(m + n \log(nC))$.

To describe the radix heap in more detail, we first set some notation. For a given shortest path problem, the radix heap consists of $1 + \lceil \log(nC) \rceil$ buckets. The buckets are numbered $0, 1, 2, \ldots, K = \lceil \log(nC) \rceil$. We represent the range of bucket k by *range(k)* which is a (possibly empty) closed interval of integers. We store a temporary node i in bucket k if $d(i) \in \text{range}(k)$. We do not store permanent nodes. The set *content(k)* denotes the nodes in bucket k. The algorithm will change the ranges of the buckets dynamically, and each time it changes the ranges, it redistributes the nodes in the buckets. Initially, the buckets have the following ranges:

$$\begin{aligned}
\text{range}(0) &= [0];\\
\text{range}(1) &= [1];\\
\text{range}(2) &= [2, 3];\\
\text{range}(3) &= [4, 7];\\
\text{range}(4) &= [8, 15];\\
&\vdots\\
\text{range}(K) &= [2^{K-1}, 2^K - 1].
\end{aligned}$$

These ranges change as the algorithm proceeds; however, the widths of the buckets never increase beyond their initial widths.

As we have noted the fundamental difficulty associated with using bucket widths larger than 1, as in the radix heap algorithm, is that we have to examine every node in the bucket containing a node with the minimum distance label and this time might be "too large" from a worst-case perspective. The radix heap algorithm overcomes this difficulty in the following manner. Suppose that at some stage the minimum indexed nonempty bucket is bucket 4, whose range is [8, 15]. The algorithm would examine every node in content(4) to identify a node with the smallest distance label. Suppose that the smallest distance label of a node in content(4) is 9. Property 4.5 implies that no temporary distance label will ever again be less than 9 and, consequently, we will never again need the buckets 0 to 3. Rather than leaving these buckets idle, the algorithm redistributes the range [9, 15] to the previous buckets, resulting in the ranges range(0) = [9], range(1) = [10], range(2) = [11, 12], range(3) = [13,15] and range(4) = $\varnothing$. Since the range of bucket 4 is now empty, the algorithm shifts (or redistributes) the nodes in content(4) into the appropriate buckets (0, 1, 2, and 3). Thus each of the nodes in bucket 4 moves to a lower-indexed bucket and all nodes with the smallest distance label move to bucket 0, which has width 1.

To summarize, whenever the algorithm finds that nodes with the minimum distance label are in a bucket with width larger than 1, it examines all nodes in the bucket to identify a node with minimum distance label. Then the algorithm redistributes the bucket ranges and shifts each node in the bucket to the lower-indexed bucket. Since the radix heap contains K buckets, a node can shift at most K times, and consequently, the algorithm will examine any node at most K times. Hence the total number of node examinations is $O(nK)$, which is not "too large."

We now illustrate the radix heap data structure on the shortest path example given in Figure 4.11 with $s = 1$. In the figure, the number beside each arc indicates its length. For this problem $C = 20$ and $K = \lceil \log(120) \rceil = 7$. Figure 4.12 specifies the distance labels determined by Dijkstra's algorithm after it has examined node 1; it also shows the corresponding radix heap.

To select the node with the smallest distance label, we scan the buckets 0, 1, 2, . . . , K to find the first nonempty bucket. In our example, bucket 0 is nonempty. Since bucket 0 has width 1, every node in this bucket has the same (minimum) distance label. So the algorithm designates node 3 as permanent, deletes node 3 from the radix heap, and scans the arc (3, 5) to change the distance label of node 5 from

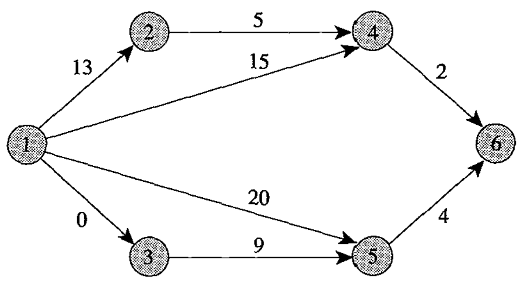

Figure 4.11 Shortest path example.

node i	1	2	3	4	5	6
label $d(i)$	0	13	0	15	20	∞

bucket k	0	1	2	3	4	5	6	7
range(k)	[0]	[1]	[2, 3]	[4, 7]	[8, 15]	[16, 31]	[32, 63]	[64, 127]
content(k)	{3}	∅	∅	∅	{2, 4}	{5}	∅	

Figure 4.12 Initial radix heap.

20 to 9. We check whether the new distance label of node 5 is contained in the range of its present bucket, which is bucket 5. It is not. Since its distance label has decreased, node 5 should move to a lower-indexed bucket. So we sequentially scan the buckets from right to left, starting at bucket 5, to identify the first bucket whose range contains the number 9, which is bucket 4. Node 5 moves from bucket 5 to bucket 4. Figure 4.13 shows the new radix heap.

node i	2	4	5	6
label $d(i)$	13	15	9	∞

bucket k	0	1	2	3	4	5	6	7
range(k)	[0]	[1]	[2, 3]	[4, 7]	[8, 15]	[16, 31]	[32, 63]	[64, 127]
content(k)	∅	∅	∅	∅	{2, 4, 5}	∅	∅	∅

Figure 4.13 Radix heap at the end of iteration 1.

We again look for the node with the smallest distance label. Scanning the buckets sequentially, we find that bucket $k = 4$ is the first nonempty bucket. Since the range of this bucket contains more than one integer, the first node in the bucket need not have the minimum distance label. Since the algorithm will never use the ranges range(0), . . . , range($k - 1$) for storing temporary distance labels, we can redistribute the range of bucket k into the buckets 0, 1, . . . , $k - 1$, and reinsert its nodes into the lower-indexed buckets. In our example, the range of bucket 4 is [8, 15], but the smallest distance label in this bucket is 9. We therefore redistribute the range [9, 15] over the lower-indexed buckets in the following manner:

$$\begin{aligned}
\text{range}(0) &= [9], \\
\text{range}(1) &= [10], \\
\text{range}(2) &= [11, 12], \\
\text{range}(3) &= [13, 15], \\
\text{range}(4) &= \varnothing.
\end{aligned}$$

Other ranges do not change. The range of bucket 4 is now empty, and we must reassign the contents of bucket 4 to buckets 0 through 3. We do so by successively selecting nodes in bucket 4, sequentially scanning the buckets 3, 2, 1, 0 and inserting the node in the appropriate bucket. The resulting buckets have the following contents:

$$\begin{aligned}
\text{content}(0) &= \{5\}, \\
\text{content}(1) &= \varnothing, \\
\text{content}(2) &= \varnothing, \\
\text{content}(3) &= \{2, 4\}, \\
\text{content}(4) &= \varnothing.
\end{aligned}$$

This redistribution necessarily empties bucket 4 and moves the node with the smallest distance label to bucket 0.

We are now in a position to outline the general algorithm and analyze its complexity. We first consider moving nodes between the buckets. Suppose that $j \in$ content(k) and that we are re-assigning node j to a lower-numbered bucket (because either $d(j)$ decreases or we are redistributing the useful range of bucket k and removing the nodes from this bucket). If $d(j) \notin$ range(k), we sequentially scan lower-numbered buckets from right to left and add the node to the appropriate bucket. Overall, this operation requires $O(m + nK)$ time. The term m reflects the number of distance updates, and the term nK arises because every time a node moves, it moves to a lower-indexed bucket: Since there are $K + 1$ buckets, a node can move at most K times. Therefore, $O(nK)$ is a bound on the total number of node movements.

Next we consider the node selection operation. Node selection begins by scanning the buckets from left to right to identify the first nonempty bucket, say bucket k. This operation requires $O(K)$ time per iteration and $O(nK)$ time in total. If $k = 0$ or $k = 1$, any node in the selected bucket has the minimum distance label. If $k \geq 2$, we redistribute the "useful" range of bucket k into the buckets $0, 1, \ldots, k - 1$ and reinsert its contents in those buckets. If the range of bucket k is $[l, u]$ and the smallest distance label of a node in the bucket is $d_{\min}$, the useful range of the bucket is $[d_{\min}, u]$.

The algorithm redistributes the useful range in the following manner: We assign the first integer to bucket 0, the next integer to bucket 1, the next two integers to bucket 2, the next four integers to bucket 3, and so on. Since bucket k has width less than 2^{k-1}, and since the widths of the first k buckets can be as large as $1, 1, 2, \ldots, 2^{k-2}$ for a total potential width of 2^{k-1}, we can redistribute the useful range of bucket k over the buckets $0, 1, \ldots, k - 1$ in the manner described. This redistribution of ranges and the subsequent reinsertions of nodes empties bucket k and moves the nodes with the smallest distance labels to bucket 0. The redistribution of ranges requires $O(K)$ time per iteration and $O(nK)$ time over all the iterations. As we have already shown, the algorithm requires $O(nK)$ time in total to move nodes and reinsert them in lower-indexed buckets. Consequently, the running time of the algorithm is $O(m + nK)$. Since $K = \lceil \log(nC) \rceil$, the algorithm runs in $O(m + n \log(nC))$ time. We summarize our discussion as follows.

Theorem 4.7. *The radix heap implementation of Dijkstra's algorithm solves the shortest path problem in $O(m + n \log(nC))$ time.*

This algorithm requires $1 + \lceil \log(nC) \rceil$ buckets. As in Dial's algorithm, Property 4.6 permits us to reduce the number of buckets to $1 + \lceil \log C \rceil$. This refined implementation of the algorithm runs in $O(m + n \log C)$ time. Using a Fibonacci heap data structure within the radix heap implementation, it is possible to reduce this bound further to $O(m + n \sqrt{\log C})$, which gives one of the fastest polynomial-time algorithm to solve the shortest path problem with nonnegative arc lengths.

4.9 SUMMARY

The shortest path problem is a core model that lies at the heart of network optimization. After describing several applications, we developed several algorithms for solving shortest path problems with nonnegative arc lengths. These algorithms, known as *label-setting algorithms*, assign tentative distance labels to the nodes and then iteratively identify a true shortest path distance (a permanent label) to one or more nodes at each step. The shortest path problem with arbitrary arc lengths requires different solution approaches; we address this problem class in Chapter 5.

The basic shortest path problem that we studied requires that we determine a shortest (directed) path from a source node s to each node $i \in N - \{s\}$. We showed how to store these $(n - 1)$ shortest paths compactly in the form of a directed out-tree rooted at node s, called the tree of shortest paths. This result uses the fact that if P is a shortest path from node s to some node j, then any subpath of P from node s to any of its internal nodes is also a shortest path to this node.

We began our discussion of shortest path algorithms by describing an $O(m)$ algorithm for solving the shortest path problem in acyclic networks. This algorithm computes shortest path distances to the nodes as it examines them in a topological order. This discussion illustrates a fact that we will revisit many times throughout this book: It is often possible to develop very efficient algorithms when we restrict the underlying network by imposing special structure on the data or on the network's topological structure (as in this case).

We next studied Dijkstra's algorithm, which is a natural and simple algorithm for solving shortest path problems with nonnegative arc lengths. After describing the original implementation of Dijkstra's algorithm, we examined several other implementations that either improve on its running time in practice or improve on its worst-case complexity. We considered the following implementations: Dial's implementation, a d-heap implementation, a Fibonacci heap implementation, and a radix heap implementation. Figure 4.14 summarizes the basic features of these implementations.

Algorithm	Running time	Features
Original implementation	$O(n^2)$	1. Selects a node with the minimum temporary distance label, designating it as permanent, and examines arcs incident to it to modify other distance labels. 2. Very easy to implement. 3. Achieves the best available running time for dense networks.
Dial's implementation	$O(m + nC)$	1. Stores the temporary labeled nodes in a sorted order in unit length buckets and identifies the minimum temporary distance label by sequentially examining the buckets. 2. Easy to implement and has excellent empirical behavior. 3. The algorithm's running time is pseudopolynomial and hence is theoretically unattractive.
d-Heap implementation	$O(m \log_d n)$, where $d = m/n$	1. Uses the d-heap data structure to maintain temporary labeled nodes. 2. Linear running time whenever $m = \Omega(n^{1+\epsilon})$ for any positive $\epsilon > 0$.
Fibonacci heap implementation	$O(m + n \log n)$	1. Uses the Fibonacci heap data structure to maintain temporary labeled nodes. 2. Achieves the best available strongly polynomial running time for solving shortest paths problems. 3. Intricate and difficult to implement.
Radix heap implementation	$O(m + n \log(nC))$	1. Uses a radix heap to implement Dijkstra's algorithm. 2. Improves Dial's algorithm by storing temporarily labeled nodes in buckets with varied widths. 3. Achieves an excellent running time for problems that satisfy the similarity assumption.

Figure 4.14 Summary of different implementations of Dijkstra's algorithm.

REFERENCE NOTES

The shortest path problem and its generalizations have a voluminous research literature. As a guide to these results before 1984, we refer the reader to the extensive bibliography compiled by Deo and Pang [1984]. In this discussion we present some selected references; additional references can be found in the survey papers of Ahuja, Magnanti, and Orlin [1989, 1991].

The first label-setting algorithm was suggested by Dijkstra [1959] and, independently, by Dantzig [1960], and Whiting and Hillier [1960]. The original implementation of Dijkstra's algorithm runs in $O(n^2)$ time, which is the optimal running time for fully dense networks [those with $m = \Omega(n^2)$] because any algorithm must examine every arc. However, the use of heaps permits us to obtain improved running times for sparse networks. The d-heap implementation of Dijkstra's algorithm with

$d = \max\{2, \lceil m/n \rceil\}$ runs in $O(m \log_d n)$ time and is due to Johnson [1977a]. The Fibonacci heap implementation, due to Fredman and Tarjan [1984], runs in $O(m + n \log n)$ time. Johnson [1982] suggested the $O(m \log \log C)$ implementation of Dijkstra's algorithm, based on earlier work by Boas, Kaas, and Zijlstra [1977]. Gabow's [1985] scaling algorithm, discussed in Exercise 5.51, is another efficient shortest path algorithm.

Dial [1969] (and also, independently, Wagner [1976]) suggested the $O(m + nC)$ implementation of Dijkstra's algorithm that we discussed in Section 4.6. Dial, Glover, Karney, and Klingman [1979] proposed an improved version of Dial's implementation that runs better in practice. Although Dial's implementation is only pseudopolynomial time, it has led to algorithms with better worst-case behavior. Denardo and Fox [1979] suggested several such improvements. The radix heap implementation that we described in Section 4.8 is due to Ahuja, Mehlhorn, Orlin, and Tarjan [1990]; we can view it as an improved version of Denardo and Fox's implementations. Our description of the radix heap implementation runs in $O(m + n \log(nC))$ time. Ahuja et al. [1990] also suggested several improved versions of the radix heap implementation that run in $O(m + n \log C)$, $O(m + (n \log C)/(\log \log C))$, $O(m + n \sqrt{\log C})$ time.

Currently, the best time bound for solving the shortest path problem with nonnegative arc lengths is $O(\min\{m + n \log n, m \log \log C, m + n \sqrt{\log C}\})$; this expression contains three terms because different time bounds are better for different values of n, m, and C. We refer to the overall time bound as $S(n, m, C)$; Fredman and Tarjan [1984], Johnson [1982], and Ahuja et al. [1990] have obtained the three bounds it contains. The best strongly polynomial-time bound for solving the shortest path problem with nonnegative arc lengths is $O(m + n \log n)$, which we subsequently refer to as $S(n, m)$.

Researchers have extensively tested label-setting algorithms empirically. Some of the more recent computational results can be found in Gallo and Pallottino [1988], Hung and Divoky [1988], and Divoky and Hung [1990]. These results suggest that Dial's implementation is the fastest label-setting algorithm for most classes of networks tested. Dial's implementation is, however, slower than some of the label-correcting algorithms that we discuss in Chapter 5.

The applications of the shortest path problem that we described in Section 4.2 are adapted from the following papers:

1. Approximating piecewise linear functions (Imai and Iri [1986])
2. Allocating inspection effort on a production line (White [1969])
3. Knapsack problem (Fulkerson [1966])
4. Tramp steamer problem (Lawler [1966])
5. System of difference constraints (Bellman [1958])
6. Telephone operator scheduling (Bartholdi, Orlin, and Ratliff [1980])

Elsewhere in this book we have described other applications of the shortest path problem. These applications include (1) reallocation of housing (Application 1.1, Wright [1975]), (2) assortment of steel beams (Application 1.2, Frank [1965]), (3) the paragraph problem (Exercise 1.7), (4) compact book storage in libraries (Exercise 4.3, Ravindran [1971]), (5) the money-changing problem (Exercise 4.5), (6) cluster analysis (Exercise 4.6), (7) concentrator location on a line (Exercises 4.7 and 4.8, Balakrishnan, Magnanti, and Wong [1989b]), (8) the personnel planning problem (Exercise 4.9, Clark and Hastings [1977]), (9) single-duty crew scheduling (Exercise 4.13, Veinott and Wagner [1962]), (10) equipment replacement (Application 9.6, Veinott and Wagner [1962]), (11) asymmetric data scaling with lower and upper bounds (Application 19.5, Orlin and Rothblum [1985]), (12) DNA sequence alignment (Application 19.7, Waterman [1988]), (13) determining minimum project duration (Application 19.9), (14) just-in-time scheduling (Application 19.10, Elmaghraby [1978], Levner and Nemirovsky [1991]), (15) dynamic lot sizing (Applications 19.19, Application 19.20, Application 19.21, Veinott and Wagner [1962], Zangwill [1969]), and (16) dynamic facility location (Exercise 19.22).

The literature considers many other applications of shortest paths that we do not cover in this book. These applications include (1) assembly line balancing (Gutjahr and Nemhauser [1964]), (2) optimal improvement of transportation networks (Goldman and Nemhauser [1967]), (3) machining process optimization (Szadkowski [1970]), (4) capacity expansion (Luss [1979]), (5) routing in computer communication networks (Schwartz and Stern [1980]), (6) scaling of matrices (Golitschek and Schneider [1984]), (7) city traffic congestion (Zawack and Thompson [1987]), (8) molecular confirmation (Dress and Havel [1988]), (9) order picking in an isle (Goetschalckx and Ratliff [1988]), and (10) robot design (Haymond, Thornton, and Warner [1988]).

Shortest path problems often arise as important subroutines within algorithms for solving many different types of network optimization problems. These applications are too numerous to mention. We do describe several such applications in subsequent chapters, however, when we show that shortest path problems are key subroutines in algorithms for the minimum cost flow problem (see Chapter 9), the assignment problem (see Section 12.4), the constrained shortest path problem (see Section 16.4), and the network design problem (see Application 16.4).

EXERCISES

4.1. Mr. Dow Jones, 50 years old, wishes to place his IRA (Individual Retirement Account) funds in various investment opportunities so that at the age of 65 years, when he withdraws the funds, he has accrued maximum possible amount of money. Assume that Mr. Jones knows the investment alternatives for the next 15 years: their maturity (in years) and the appreciation they offer. How would you formulate this investment problem as a shortest path problem, assuming that at any point in time, Mr. Jones invests all his funds in a single investment alternative.

4.2. Beverly owns a vacation home in Cape Cod that she wishes to rent for the period May 1 to August 31. She has solicited a number of bids, each having the following form: the day the rental starts (a rental day starts at 3 P.M.), the day the rental ends (checkout time is noon), and the total amount of the bid (in dollars). Beverly wants to identify a selection of bids that would maximize her total revenue. Can you help her find the best bids to accept?

4.3. **Compact book storage in libraries** (Ravindran [1971]). A library can store books according to their subject or author classification, or by their size, or by any other method that permits an orderly retrieval of the books. This exercise concerns an optimal storage of books by their size to minimize the storage cost for a given collection of books.

Suppose that we know the heights and thicknesses of all the books in a collection (assuming that all widths fit on the same shelving, we consider only a two-dimensional problem and ignore book widths). Suppose that we have arranged the book heights in ascending order of their n known heights $H_1, H_2, \ldots, H_n$; that is, $H_1 < H_2 < \cdots < H_n$. Since we know the thicknesses of the books, we can compute the required length of shelving for each height class. Let L_i denote the length of shelving for books of height H_i. If we order shelves of height H_i for length x_i, we incur cost equal to $F_i + C_i x_i$; F_i is a fixed ordering cost (and is independent of the length ordered) and C_i is the cost of the shelf per unit length. Notice that in order to save the fixed cost of ordering, we might not order shelves of every possible height because we can use a shelf of height H_i to store books of smaller heights. We want to determine the length of shelving for each height class that would minimize the total cost of the shelving. Formulate this problem as a shortest path problem.

4.4. Consider the compact book storage problem discussed in Exercise 4.3. Show that the storage problem is trivial if the fixed cost of ordering shelves is zero. Next, solve the compact book storage problem with the following data.

i	1	2	3	4	5	6
H_i	5 in.	6 in.	7 in.	9 in.	12 in.	14 in.
L_i	100	300	200	300	500	100
E_i	1000	1200	1100	1600	1800	2000
C_i	5	6	7	9	12	14

4.5. **Money-changing problem.** The money-changing problem requires that we determine whether we can change a given number p into coins of known denominations $a_1, a_2, \ldots, a_k$. For example, if $k = 3$, $a_1 = 3$, $a_2 = 5$, $a_3 = 7$, we can change all the numbers in the set $\{8, 12, 54\}$; on the other hand, we cannot change the number 4. In general, the money-changing problem asks whether $p = \sum_{i=1}^{k} a_i x_i$ for some nonnegative integers $x_1, x_2, \ldots, x_k$.

(a) Describe a method for identifying all numbers in a given range of numbers $[l, u]$ that we can change.

(b) Describe a method that identifies whether we can change a given number p, and if so, then identifies a denomination with the least number of coins.

4.6. **Cluster analysis.** Consider a set of n scalar numbers $a_1, a_2, \ldots, a_n$ arranged in nondecreasing order of their values. We wish to partition these numbers into clusters (or groups) so that (1) each cluster contains at least p numbers; (2) each cluster contains consecutive numbers from the list $a_1, a_2, \ldots, a_n$; and (3) the sum of the squared deviation of the numbers from their cluster means is as small as possible. Let $\bar{a}(S) = (\sum_{i \in S} a_i)/|S|$ denote the mean of a set S of numbers defining a cluster. If the number a_k belongs to cluster S, the squared deviation of the number a_k from the cluster mean is $(a_k - \bar{a}(S))^2$. Show how to formulate this problem as a shortest path problem. Illustrate your formulation using the following data: $p = 2$, $n = 6$, $a_1 = 0.5$, $a_2 = 0.8$, $a_3 = 1.1$, $a_4 = 1.5$, $a_5 = 1.6$, and $a_6 = 2.0$.

4.7. **Concentrator location on a line** (Balakrishnan, Magnanti, and Wong [1989]). In the telecommunication industry, telephone companies typically connect each customer directly to a switching center, which is a device that routes calls between the users in the system. Alternatively, to use fewer cables for routing the telephone calls, a company can combine the calls of several customers in a message compression device known as a *concentrator* and then use a single cable to route all of the calls transmitted by those users to the switching center. Constructing a concentrator at any node in the telephone network incurs a node-specific cost and assigning each customer to any concentrator incurs a "homing cost" that depends on the customer and the concentrator location. Suppose that all of the customers lie on a path and that we wish to identify the optimal location of concentrators to service these customers (assume that we must assign each customer to one of the concentrators). Suppose further that the set of customers allocated to any concentrator must be contiguous on the path (many telephone companies use this customer grouping policy). How would you find the optimal location of a single concentrator that serves any contiguous set of customers? Show how to use the solution of these single-location subproblems (one for each interval of customers) to solve the concentrator location problem on the path as a shortest path problem.

4.8. Modified concentrator location problem. Show how to formulate each of the following variants of the concentrator location problem that we consider in Exercise 4.7 as a shortest path problem. Assume in each case that all the customer lie on a path.

(a) The cost of connecting each customer to a concentrator is negligible, but each concentrator can handle at most five customers.

(b) Several types of concentrators are available at each node; each type of concentrator has its own cost and its own capacity (which is the maximum number of customers it can accommodate).

(c) In the situations considered in Exercise 4.7 and in parts (a) and (b) of this exercise, no customer can be assigned to a concentrator more that 1200 meters from the concentrator (because of line degradation of transmitted signals).

4.9. Personnel planning problem (Clark and Hastings [1977]). A construction company's work schedule on a certain site requires the following number of skilled personnel, called *steel erectors*, in the months of March through August:

Month	Mar.	Apr.	May	June	July	Aug.
Personnel	4	6	7	4	6	2

Personnel work at the site on the monthly basis. Suppose that three steel erectors are on the site in February and three steel erectors must be on site in September. The problem is to determine how many workers to have on site in each month in order to minimize costs, subject to the following conditions:

Transfer costs. Adding a worker to this site costs \$100 per worker and redeploying a worker to another site costs \$160.

Transfer rules. The company can transfer no more than three workers at the start of any month, and under a union agreement, it can redeploy no more than one-third of the current workers in any trade from a site at the end of any month.

Shortage time and overtime. The company incurs a cost of \$200 per worker per month for having a surplus of steel erectors on site and a cost of \$200 per worker per month for having a shortage of workers at the site (which must be made up in overtime). Overtime cannot exceed 25 percent of the regular work time.

Formulate this problem as a shortest path problem and solve it. (*Hint*: Give a dynamic programming-based formulation and use as many nodes for each month as the maximum possible number of steel erectors.)

4.10. Multiple-knapsack problem. In the shortest path formulation of the knapsack problem discussed in Application 4.3, an item is either placed in the knapsack or not. Consequently, each $x_j \in \{0, 1\}$. Consider a situation in which the hiker can place multiple copies of an item in her knapsack (i.e., $x_j \in \{0, 1, 2, 3, \ldots\}$). How would you formulate this problem as a shortest path problem? Illustrate your formulation on the example given in Application 4.3.

4.11. Modified system of difference constraints. In discussing system of difference constraints in Application 4.5, we assumed that each constraint is of the form $x(j_k) - x(i_k) \le b(k)$. Suppose, instead, that some constraints are of the form $x(j_k) \le b(k)$ or $x(i_k) \ge b(k)$. Describe how you would solve this modified system of constraints using a shortest path algorithm.

4.12. Telephone operator scheduling. In our discussion of the telephone operator scheduling problem in Application 4.6, we described a method for solving a restricted problem of determining whether some feasible schedule uses at most p operators. Describe a polynomial-time algorithm for determining a schedule with the fewest operators that uses the restricted problem as a subproblem.

4.13. Single-dnty crew scheduling. The following table illustrates a number of possible duties for the drivers of a bus company. We wish to ensure, at the lowest possible cost, that at least one driver is on duty for each hour of the planning period (9 A.M. to 5 P.M.). Formulate and solve this scheduling problem as a shortest path problem.

Duty hours	9–1	9–11	12–3	12–5	2–5	1–4	4–5
Cost	30	18	21	38	20	22	9

4.14. Solve the shortest path problems shown in Figure 4.15 using the original implementation of Dijkstra's algorithm. Count the number of distance updates.

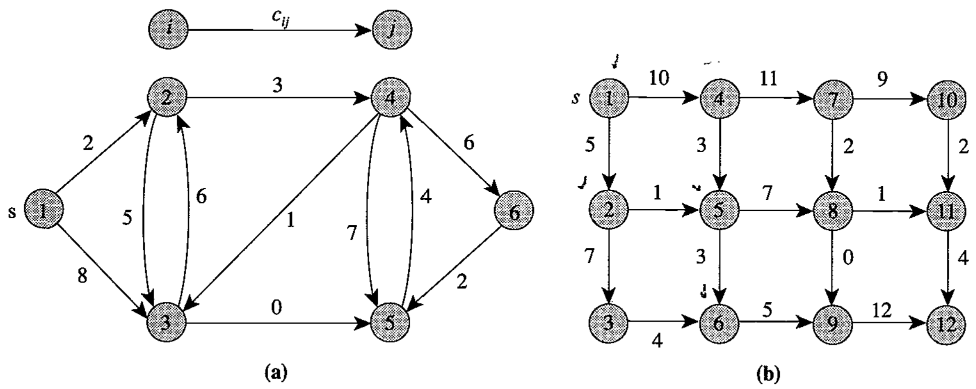

Figure 4.15 Some shortest path networks.

4.15. Solve the shortest path problem shown in Figure 4.15(a) using Dial's implementation of Dijkstra's algorithm. Show all of the buckets along with their content after the algorithm has examined the most recent permanently labeled node at each step.

4.16. Solve the shortest path problem shown in Figure 4.15(a) using the radix heap algorithm.

4.17. Consider the network shown in Figure 4.16. Assign integer lengths to the arcs in the network so that for every $k \in [0, 2^K - 1]$, the network contains a directed path of length k from the source node to sink node.

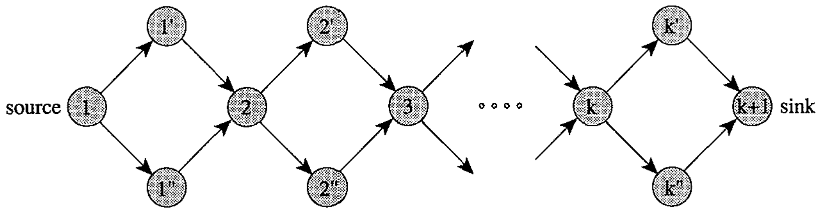

Figure 4.16 Network for Exercise 4.17.

4.18. Suppose that all the arcs in a network G have length 1. Show that Dijkstra's algorithm examines nodes for this network in the same order as the breadth-first search algorithm described in Section 3.4. Consequently, show that it is possible to solve the shortest path problem in this unit length network in $O(m)$ time.

4.19. Construct an example of the shortest path problem with some negative arc lengths, but no negative cycle, that Dijkstra's algorithm will solve correctly. Construct another example that Dijkstra's algorithm will solve incorrectly.

4.20. (Malik, Mittal, and Gupta [1989]) Consider a network without any negative cost cycle. For every node $j \in N$, let $d^s(j)$ denote the length of a shortest path from node s to node j and let $d^t(j)$ denote the length of a shortest path from node j to node t.

(a) Show that an arc (i, j) is on a shortest path from node s to node t if and only if $d^s(t) = d^s(i) + c_{ij} + d^t(j)$.

(b) Show that $d^s(t) = \min\{d^s(i) + c_{ij} + d^t(j) : (i, j) \in A\}$.

4.21. Which of the following claims are true and which are false? Justify your answer by giving a proof or by constructing a counterexample.

(a) If all arcs in a network have different costs, the network has a unique shortest path tree.

(b) In a directed network with positive arc lengths, if we eliminate the direction on every arc (i.e., make it undirected), the shortest path distances will not change.

(c) In a shortest path problem, if each arc length increases by k units, shortest path distances increase by a multiple of k.

(d) In a shortest path problem, if each arc length decreases by k units, shortest path distances decrease by a multiple of k.

(e) Among all shortest paths in a network, Dijkstra's algorithm always finds a shortest path with the least number of arcs.

4.22. Suppose that you are given a shortest path problem in which all arc lengths are the same. How will you solve this problem in the least possible time?

4.23. In our discussion of shortest path algorithms, we often assumed that the underlying network has no parallel arcs (i.e., at most one arc has the same tail and head nodes). How would you solve a problem with parallel arcs? (*Hint*: If the network contains k parallel arcs directed from node i to node j, show that we can eliminate all but one of these arcs.)

4.24. Suppose that you want to determine a path of shortest length that can start at either of the nodes s_1 or s_2 and can terminate at either of the nodes t_1 and t_2. How would you solve this problem?

4.25. Show that in the shortest path problem if the length of some arc decreases by k units, the shortest path distance between any pair of nodes decreases by at most k units.

4.26. **Most vital arc problem.** A *vital arc* of a network is an arc whose removal from the network causes the shortest distance between two specified nodes, say node s and node t, to increase. A most vital arc is a vital arc whose removal yields the greatest increase in the shortest distance from node s to node t. Assume that the network is directed, arc lengths are positive, and some arc is vital. Prove that the following statements are true or show through counterexamples that they are false.

(a) A most vital arc is an arc with the maximum value of c_{ij}.

(b) A most vital arc is an arc with the maximum value of c_{ij} on some shortest path from node s to node t.

(c) An arc that does not belong to any shortest path from node s to node t cannot be a most vital arc.

(d) A network might contain several most vital arcs.

4.27. Describe an algorithm for determining a most vital arc in a directed network. What is the running time of your algorithm?

4.28. A *longest path* is a directed path from node s to node t with the maximum length. Suggest an $O(m)$ algorithm for determining a longest path in an acyclic network with nonnegative arc lengths. Will your algorithm work if the network contains directed cycles?

4.29. Dijkstra's algorithm, as stated in Figure 4.6, identifies a shortest directed path from node s to every node $j \in N - \{s\}$. Modify this algorithm so that it identifies a shortest directed path from each node $j \in N - \{t\}$ to node t.

4.30. Show that if we add a constant α to the length of every arc emanating from the source node, the shortest path tree remains the same. What is the relationship between the shortest path distances of the modified problem and those of the original problem?

4.31. Can adding a constant α to the length of every arc emanating from a nonsource node produce a change in the shortest path tree? Justify your answer.

4.32. Show that Dijkstra's algorithm runs correctly even when a network contains negative cost arcs, provided that all such arcs emanate from the source node. (*Hint*: Use the result of Exercise 4.30.)

4.33. **Improved Dial's implementation** (Denardo and Fox [1979]). This problem discusses a practical speed-up of Dial's implementation. Let $c_{\min} = \min\{c_{ij} : (i, j) \in A\}$ and $w = \max\{1, c_{\min}\}$. Consider a version of Dial's implementation in which we use buckets of width w. Show that the algorithm will never decrease the distance label of any node in the least index nonempty bucket; consequently, we can permanently label any node in this bucket. What is the running time of this version of Dial's implementation?

4.34. Suppose that we arrange all directed paths from node s to node t in nondecreasing order of their lengths, breaking ties arbitrarily. The kth shortest path problem is to identify a path that can be at the kth place in this order. Describe an algorithm to find the kth shortest path for $k = 2$. (*Hint*: The second shortest path must differ from the first shortest path by at least one arc.)

4.35. Suppose that every directed cycle in a graph G has a positive length. Show that a shortest directed walk from node s to node t is always a path. Construct an example for which the first shortest directed walk is a path, but the second shortest directed walk is not a path.

4.36. Describe a method for identifying the first K shortest paths from node s to node t in an acyclic directed network. The running time of your algorithm should be polynomial in terms of n, m, and K. (*Hint*: For each node j, keep track of the first K shortest paths from node s to node j. Also, use the results in Exercise 4.34.)

4.37. **Maximum capacity path problem.** Let $c_{ij} \geq 0$ denote the capacity of an arc in a given network. Define the *capacity* of a directed path P as the minimum arc capacity in P. The *maximum capacity path problem* is to determine a maximum capacity path from a specified source node s to every other node in the network. Modify Dijkstra's algorithm so that it solves the maximum capacity path problem. Justify your algorithm.

4.38. Let (i_1, j_1), (i_2, j_2), . . . , (i_m, j_m) denote the arcs of a network in nondecreasing order of their arc capacities. Show that the maximum capacity path from node s to any node j remains unchanged if we modify some or all of the arc capacities but maintain the same (capacity) order for the arcs. Use this result to show that if we already have a sorted list of the arcs, we can solve the maximum capacity path problem in $O(m)$ time. (*Hint*: Modify arc capacities so that they are all between 1 and m. Then use a variation of Dial's implementation.)

4.39. Maximum reliability path problems. In the network G we associate a reliability $0 < \mu_{ij} \le 1$ with every arc $(i, j) \in A$; the reliability measures the probability that the arc will be operational. We define the reliability of a directed path P as the product of the reliability of arcs in the path [i.e., $\mu(P) = \prod_{(i,j) \in P} \mu_{ij}$]. The maximum reliability path problem is to identify a directed path of maximum reliability from the source node s to every other node in the network.

(a) Show that if we are allowed to take logarithms, we can reduce the maximum reliability path problem to a shortest path problem.

(b) Suppose that you are not allowed to take logarithms because they yield irrational data. Specify an $O(n^2)$ algorithm for solving the maximum reliability path problem and prove the correctness of this algorithm. (*Hint*: Modify Dijkstra's algorithm.)

(c) Will your algorithms in parts (a) and (b) work if some of the coefficients μ_{ij} are strictly greater than 1?

4.40. Shortest paths with turn penalties. Figure 4.15(b) gives a road network in which all road segments are parallel to either the x-axis or the y-axis. The figure also gives the traversal costs of arcs. Suppose that we incur an additional cost (or penalty) of α units every time we make a left turn. Describe an algorithm for solving the shortest path problem with these turn penalties and apply it to the shortest path example in Figure 4.15(b). Assume that $\alpha = 5$. [*Hint*: Create a new graph G^* with a node $i - j$ corresponding to each arc $(i, j) \in A$ and with each pair of nodes $i - j$ and $j - k$ in N joined by an arc. Assign appropriate arc lengths to the new graph.]

4.41. Max-min result. We develop a max-min type of result for the maximum capacity path problem that we defined in Exercise 4.37. As in that exercise, suppose that we wish to find the maximum capacity path from node s to node t. We say that a cut $[S, \bar{S}]$ is an *s–t cut* if $s \in S$ and $t \in \bar{S}$. Define the *bottleneck value* of an s–t cut as the largest arc capacity in the cut. Show that the capacity of the maximum capacity path from node s to node t equals the minimum bottleneck value of a cut.

4.42. A farmer wishes to transport a truckload of eggs from one city to another city through a given road network. The truck will incur a certain amount of breakage on each road segment; let w_{ij} denote the fraction of the eggs broken if the truck traverses the road segment (i, j). How should the truck be routed to minimize the total breakage? How would you formulate this problem as a shortest path problem.

4.43. A^* algorithm. Suppose that we want to identify a shortest path from node s to node t, and not necessarily from s to any other node, in a network with nonnegative arc lengths. In this case we can terminate Dijkstra's algorithm whenever we permanently label node t. This exercise studies a modification of Dijkstra's algorithm that would speed up the algorithm in practice by designating node t as a permanent labeled node more quickly. Let $h(i)$ be a lower bound on the length of the shortest path from node i to node t and suppose that the lower bounds satisfy the conditions $h(i) \le h(j) + c_{ij}$ for all $(i, j) \in A$. For instance, if nodes are points in a two-dimensional plane with coordinates (x_i, y_i) and arc lengths equal Euclidean distances between points, then $h(i) = [(x_i - x_t)^2 + (y_i - y_t)^2]^{1/2}$ (i.e., the Euclidean distance from i to t) is a valid lower bound on the length of the shortest path from node i to node t.

(a) Let $c_{ij}^h = c_{ij} + h(j) - h(i)$ for all $(i, j) \in A$. Show that replacing the arc lengths c_{ij} by c_{ij}^h does not affect the shortest paths between any pair of nodes.

(b) If we apply Dijkstra's algorithm with c_{ij}^h as arc lengths, why should this modification improve the empirical behavior of the algorithm? [*Hint*: What is its impact if each $h(i)$ represents actual shortest path distances from node i to node t?]

4.44. Arc tolerances. Let T be a shortest path tree of a network. Define the *tolerances* of an arc (i, j) as the maximum increase, α_{ij}, and the maximum decrease, β_{ij}, that the arc can tolerate without changing the tree of shortest paths.

(a) Show that if the arc $(i, j) \notin T$, then $\alpha_{ij} = +\infty$ and β_{ij} will be a finite number. Describe an $O(1)$ method for computing β_{ij}.
(b) Show that if the arc $(i, j) \in T$, then $\beta_{ij} = +\infty$ and α_{ij} will be a finite number. Describe an $O(m)$ method for computing α_{ij}.

4.45. (a) Describe an algorithm that will determine a shortest walk from a source node s to a sink node t subject to the additional condition that the walk must visit a specified node p. Will this walk always be a path?
(b) Describe an algorithm for determining a shortest walk from node s to node t that must visit a specified arc (p, q).

4.46. **Constrained shortest path problem.** Suppose that we associate two integer numbers with each arc in a network G: the arc's length c_{ij} and its traversal time $\tau_{ij} > 0$ (we assume that the traversal times are integers). The *constrained shortest path problem* is to determine a shortest length path from a source node s to every other node with the additional constraint that the traversal time of the path does not exceed τ_0. In this exercise we describe a dynamic programming algorithm for solving the constrained shortest path problem. Let $d_j(\tau)$ denote the length of a shortest path from node s to node j subject to the condition that the traversal time of the path does not exceed τ. Suppose that we set $d_j(\tau) = \infty$ for $\tau < 0$. Justify the following equations:

$$d_s(0) = 0,$$
$$d_j(\tau) = \min[d_j(\tau - 1), \min_k\{d_k(\tau - \tau_{kj}) + c_{kj}\}].$$

Use these equations to design an algorithm for the constrained shortest path problem and analyze its running time.

4.47. **Generalized knapsack problem.** In the knapsack problem discussed in Application 4.3, suppose that each item j has three associated numbers: *value* v_j, *weight* w_j, and *volume* r_j. We want to maximize the value of the items put in the knapsack subject to the condition that the total weight of the items is at most W and the total volume is at most R. Formulate this problem as a shortest path problem with an additional constraint.

4.48. Consider the generalized knapsack problem studied in Exercise 4.47. Extend the formulation in Application 4.3 in order to transform this problem into a longest path problem in an acyclic network.

4.49. Suppose that we associate two numbers with each arc (i, j) in a directed network $G = (N, A)$: the arc's length c_{ij} and its reliability r_{ij}. We define the reliability of a directed path P as the product of the reliabilities of arcs in the path. Describe a method for identifying a shortest length path from node s to node t whose reliability is at least r.

4.50. **Resource-constrained shortest path problem.** Suppose that the traversal time τ_{ij} of an arc (i, j) in a network is a function $f_{ij}(d)$ of the discrete amount of a resource d that we consume while traversing the arc. Suppose that we want to identify the shortest directed path from node s to node t subject to a budget D on the amount of the resource we can consume. (For example, we might be able to reduce the traversal time of an arc by using more fuel, and we want to travel from node s to node t before we run out of fuel.) Show how to formulate this problem as a shortest path problem. Assume that $d = 3$. (*Hint*: Give a dynamic programming-based formulation.)

4.51. **Modified function approximation problem.** In the function approximation problem that we studied in Application 4.1, we approximated a given piecewise linear function $f_1(x)$ by another piecewise linear function $f_2(x)$ in order to minimize a weighted function of the two costs: (1) the cost required to store the data needed to represent the function $f_2(x)$, and (2) the errors introduced by the approximating $f_1(x)$ by $f_2(x)$. Suppose that, instead, we wish to identify a subset of at most p points so that the function $f_2(x)$ defined by these points minimizes the errors of the approximation (i.e., $\sum_{k=1}^{n} [f_1(x_k) - f_2(x_k)]^2$). That is, instead of imposing a cost on the use of any breakpoint in the approximation, we impose a limit on the number of breakpoints we can use. How would you solve this problem?

4.52. Bidirectional Dijkstra's algorithm (Helgason, Kennington, and Stewart [1988]). Show that the bidirectional shortest path algorithm described in Section 4.5 correctly determines a shortest path from node s to node t. [*Hint*: At the termination of the algorithm, let S and S' be the sets of nodes that the forward and reverse versions of Dijkstra's algorithm have designated as permanently labeled. Let $k \in S \cap S'$. Let P^* be some shortest path from node s to node t; suppose that the first q nodes of P^* are in S and that the $(q + 1)$st node of P^* is not in S. Show first that some shortest path from node s to node t has the same first q nodes as P^* and has its $(q + 1)$st node in S'. Next show that some shortest path has the same first q nodes as P^* and each subsequent node in S'.]

4.53. Shortest paths in bipartite networks (Orlin [1988]). In this exercise we discuss an improved algorithm for solving shortest path problem in "unbalanced" bipartite networks $G = (N_1 \cup N_2, A)$, that is, those satisfying the condition that $n_1 = |N_1| \ll |N_2| = n_2$. Assume that the degree of any node in N_2 is at most K for some constant K, and that all arc costs are nonnegative. Shortest path problems with this structure arise in the context of solving the minimum cost flow problem (see Section 10.6). Let us define a graph $G' = (N_1, A')$ whose arc set A' is defined as the following set of arcs: For every pair of arcs (i, j) and (j, k) in A, A' has an arc (i, k) of cost equal to $c_{ij} + c_{jk}$.

(a) Show how to solve the shortest path problem in G by solving a shortest path problem in G'. What is the resulting running time of solving the shortest path problem in G in terms of the parameters n, m and K?

(b) A network G is *semi-bipartite* if we can partition its node set N into the subsets N_1 and N_2 so that no arc has both of its endpoints in N_2. Assume again that $|N_1| \ll |N_2|$ and the degree of any node in N_2 is at most K. Suggest an improved algorithm for solving shortest path problems in semi-bipartite networks.

5

SHORTEST PATHS: LABEL-CORRECTING ALGORITHMS

To get to heaven, turn right and keep straight ahead.
—Anonymous

Chapter Outline

5.1 INTRODUCTION

In Chapter 4 we saw how to solve shortest path problems very efficiently when they have special structure: either a special network topology (acyclic networks) or a special cost structure (nonnegative arc lengths). When networks have arbitrary costs and arbitrary topology, the situation becomes more complicated. As we noted in Chapter 4, for the most general situations—that is, general networks with negative cycles—finding shortest paths appears to be very difficult. In the parlance of computational complexity theory, these problems are NP-complete, so they are equivalent to solving many of the most noted and elusive problems encountered in the realm of combinatorial optimization and integer programming. Consequently, we have little hope of devising polynomial-time algorithms for the most general problem setting. Instead, we consider a tractable compromise somewhere between the special cases we examined in Chapter 4 and the most general situations: namely, algorithms that either identify a negative cycle, when one exists, or if the underlying network contains no negative cycle, solves the shortest path problem.

Essentially, all shortest path algorithms rely on the same important concept: distance labels. At any point during the execution of an algorithm, we associate a numerical value, or distance label, with each node. If the label of any node is infinite, we have yet to find a path joining the source node and that node. If the label is finite, it is the distance from the source node to that node along some path. The most basic algorithm that we consider in this chapter, the generic label-correcting algorithm, reduces the distance label of one node at each iteration by considering only local information, namely the length of the single arc and the current distance labels of its incident nodes. Since we can bound the sum of the distance labels from above and below in terms of the problem data, then under the assumption of integral costs, the distance labels will be integral and so the generic algorithm will always be finite. As is our penchant in this book, however, we wish to discover algorithms that are not only finite but that require a number of computations that grow as a (small) polynomial in the problem's size.

We begin the chapter by describing optimality conditions that permit us to assess when a set of distance labels are optimal—that is, are the shortest path distances from the source node. These conditions provide us with a termination criterion, or optimality certificate, for telling when a feasible solution to our problem is optimal and so we need perform no further computations. The concept of optimality conditions is a central theme in the field of optimization and will be a recurring theme throughout our treatment of network flows in this book. Typically, optimality conditions provide us with much more than a termination condition; they often provide considerable problem insight and also frequently suggest algorithms for solving optimization problems. When a tentative solution does not satisfy the optimality conditions, the conditions often suggest how we might modify the current solution so that it becomes "closer" to an optimal solution, as measured by some underlying metric. Our use of the shortest path optimality conditions in this chapter for developing label-correcting algorithms demonstrates the power of optimality conditions in guiding the design of solution algorithms.

Although the general label-correcting algorithm is finite, it requires $O(n^2C)$ computations to solve shortest path problems on networks with n nodes and with a bound of C on the maximum absolute value of any arc length. This bound is not very satisfactory because it depends linearly on the values of the arc costs. One of the advantages of the generic label-correcting algorithm is its flexibility: It offers considerable freedom in the tactics used for choosing arcs that will lead to improvements in the shortest path distances. To develop algorithms that are better in theory and in practice, we consider specific strategies for examining the arcs. One "balancing" strategy that considers arcs in a sequential wraparound fashion requires only $O(nm)$ computations. Another implementation that gives priority to arcs emanating from nodes whose labels were changed most recently, the so-called dequeue implementation, has performed very well in practice even though it has poor worst-case performance. In Section 5.4 we study both of these modified versions of the generic label-correcting algorithm.

We next consider networks with negative cycles and show how to make several types of modifications to the various label-correcting algorithms so that they can detect the presence of negative cycles, if the underlying network contains any. One nice feature of these methods is that they do not add to the worst-case computational complexity of any of the label-correcting algorithms.

We conclude this chapter by considering algorithms for finding shortest paths between all pairs of nodes in a network. We consider two approaches to this problem. One approach repeatedly applies the label-setting algorithm that we considered in Chapter 4, with each node serving as the source node. As the first step in this procedure, we apply the label-correcting algorithm to find the shortest paths from one arbitrary node, and use the results of this shortest path computation to redefine the costs so that they are all nonnegative and so that the subsequent n single-source problems are all in a form so that we can apply more efficient label-setting algorithms. The computational requirements for this algorithm is essentially the same as that required to solve n shortest path problems with nonnegative arc lengths and depends on which label-setting algorithm we adopt from those that we described in Chapter 4. The second approach is a label-correcting algorithm that simultaneously finds the shortest path distances between all pairs of nodes. This algorithm is very easy to implement; it uses a clever dynamic programming recursion and is able to solve the all-pairs shortest path problem in $O(n^3)$ computations.

5.2 *OPTIMALITY CONDITIONS*

As noted previously, label-correcting algorithms maintain a distance label $d(j)$ for every node $j \in N$. At intermediate stages of computation, the distance label $d(j)$ is an estimate of (an upper bound on) the shortest path distance from the source node s to node j, and at termination it is the shortest path distance. In this section we develop necessary and sufficient conditions for a set of distance labels to represent shortest path distances. Let $d(j)$ for $j \neq s$ denote the length of a shortest path from the source node to the node j [we set $d(s) = 0$]. If the distance labels are shortest path distances, they must satisfy the following necessary optimality conditions:

$$d(j) \le d(i) + c_{ij}, \qquad \text{for all } (i, j) \in A. \tag{5.1}$$

These inequalities state that for every arc (i, j) in the network, the length of the shortest path to node j is no greater than the length of the shortest path to node i plus the length of the arc (i, j). For, if not, some arc $(i, j) \in A$ must satisfy the condition $d(j) > d(i) + c_{ij}$; in this case, we could improve the length of the shortest path to node j by passing through node i, thereby contradicting the optimality of distance labels $d(j)$.

These conditions also are sufficient for optimality, in the sense that if each $d(j)$ represents the length of some directed path from the source node to node j and this solution satisfies the conditions (5.1), then it must be optimal. To establish this result, consider any solution $d(j)$ satisfying (5.1). Let $s = i_1 - i_2 - \dots - i_k = j$ be any directed path P from the source to node j. The conditions (5.1) imply that

$$\begin{aligned} d(j) = d(i_k) &\le d(i_{k-1}) + c_{i_{k-1}i_k}, \\ d(i_{k-1}) &\le d(i_{k-2}) + c_{i_{k-2}i_{k-1}}, \\ &\vdots \\ d(i_2) &\le d(i_1) + c_{i_1i_2} = c_{i_1i_2}. \end{aligned}$$

The last equality follows from the fact that $d(i_1) = d(s) = 0$. Adding these inequalities, we find that

$$d(j) = d(i_k) \le c_{i_{k-1}i_k} + c_{i_{k-2}i_{k-1}} + c_{i_{k-3}i_{k-2}} + \dots + c_{i_1i_2} = \sum_{(i,j)\in P} c_{ij}.$$

Thus $d(j)$ is a lower bound on the length of any directed path from the source to node j. Since $d(j)$ is the length of some directed path from the source to node j, it also is an upper bound on the shortest path length. Therefore, $d(j)$ is the shortest path length, and we have established the following result.

Theorem 5.1 (Shortest Path Optimality Conditions). *For every node $j \in N$, let $d(j)$ denote the length of some directed path from the source node to node j. Then the numbers $d(j)$ represent shortest path distances if and only if they satisfy the following shortest path optimality conditions:*

$$d(j) \leq d(i) + c_{ij} \qquad \text{for all } (i, j) \in A. \tag{5.2}$$ ◆

Let us define the reduced arc length c_{ij}^d of an arc (i, j) with respect to the distance labels $d(\cdot)$ as $c_{ij}^d = c_{ij} + d(i) - d(j)$. The following properties about the reduced arc lengths will prove to be useful in our later development.

Property 5.2

(a) *For any directed cycle W, $\sum_{(i,j)\in W} c_{ij}^d = \sum_{(i,j)\in W} c_{ij}$.*
(b) *For any directed path P from node k to node l, $\sum_{(i,j)\in P} c_{ij}^d = \sum_{(i,j)\in P} c_{ij} + d(k) - d(l)$.*
(c) *If $d(\cdot)$ represent shortest path distances, $c_{ij}^d \geq 0$ for every arc $(i, j) \in A$.*

The proof of the first two results is similar to the proof of Property 2.5 in Section 2.4. The third result follows directly from Theorem 5.1.

We next note that if the network contains a negative cycle, then no set of distance labels $d(\cdot)$ satisfies (5.2). For suppose that W is a directed cycle in G. Property 5.2(c) implies that $\sum_{(i,j)\in W} c_{ij}^d \geq 0$. Property 5.2(a) implies that $\sum_{(i,j)\in W} c_{ij}^d = \sum_{(i,j)\in W} c_{ij} \geq 0$, and therefore W cannot be a negative cycle. Thus if the network were to contain a negative cycle, no distance labels could satisfy (5.2). We show in the next section that if the network does not contain a negative cycle, some shortest path distances do satisfy (5.2).

For those familiar with linear programming, we point out that the shortest path optimality conditions can also be viewed as the linear programming optimality conditions. In the linear programming formulation of the shortest path problem, the negative of the shortest path distances [i.e., $-d(j)$] define the optimal dual variables, and the conditions (5.2) are equivalent to the fact that in the optimal solution, reduced costs of all primal variables are nonnegative. The presence of a negative cycle implies the unboundedness of the primal problem and hence the infeasibility of the dual problem.

5.3 GENERIC LABEL-CORRECTING ALGORITHMS

In this section we study the generic label-correcting algorithm. We shall study several special implementations of the generic algorithm in the next section. Our discussion in this and the next section assumes that the network does not contain any negative cycle; we consider the case of negative cycles in Section 5.5.

The generic label-correcting algorithm maintains a set of distance labels $d(\cdot)$ at every stage. The label $d(j)$ is either ∞, indicating that we have yet to discover a directed path from the source to node j, or it is the length of some directed path from the source to node j. For each node j we also maintain a predecessor index, $pred(j)$, which records the node prior to node j in the current directed path of length $d(j)$. At termination, the predecessor indices allow us to trace the shortest path from the source node back to node j. The generic label-correcting algorithm is a general procedure for successively updating the distance labels until they satisfy the shortest path optimality conditions (5.2). Figure 5.1 gives a formal description of the generic label-correcting algorithm.

```
algorithm label-correcting;
begin
    $d(s) := 0$ and pred$(s) := 0$;
    $d(j) := \infty$ for each $j \in N - \{s\}$;
    while some arc $(i, j)$ satisfies $d(j) > d(i) + c_{ij}$ do
    begin
        $d(j) := d(i) + c_{ij}$;
        pred$(j) := i$;
    end;
end;
```

Figure 5.1 Generic label-correcting algorithm.

By definition of reduced costs, the distance labels $d(\cdot)$ satisfy the optimality conditions if $c_{ij}^d \geq 0$ for all $(i, j) \in A$. The generic label-correcting algorithm selects an arc (i, j) violating its optimality condition (i.e., $c_{ij}^d < 0$) and uses it to update the distance label of node j. This operation decreases the distance label of node j and makes the reduced arc length of arc (i, j) equal to zero.

We illustrate the generic label correcting algorithm on the network shown in Figure 5.2(a). If the algorithm selects the arcs (1, 3), (1, 2), (2, 4), (4, 5), (2, 5), and (3, 5) in this sequence, we obtain the distance labels shown in Figure 5.2(b) through (g). At this point, no arc violates its optimality condition and the algorithm terminates.

The algorithm maintains a predecessor index for every finitely labeled node. We refer to the collection of arcs (pred$(j), j$) for every finitely labeled node j (except the source node) as the *predecessor graph*. The predecessor graph is a directed out-tree T rooted at the source that spans all nodes with finite distance labels. Each distance update using the arc (i, j) produces a new predecessor graph by deleting the arc (pred$(j), j$) and adding the arc (i, j). Consider, for example, the graph shown in Figure 5.3(a): the arc (6, 5) enters, the arc (3, 5) leaves, and we obtain the graph shown in Figure 5.3(b).

The label-correcting algorithm satisfies the invariant property that for every arc (i, j) in the predecessor graph, $c_{ij}^d \leq 0$. We establish this result by performing induction on the number of iterations. Notice that the algorithm adds an arc (i, j) to the predecessor graph during a distance update, which implies that after this update $d(j) = d(i) + c_{ij}$, or $c_{ij} + d(i) - d(j) = c_{ij}^d = 0$. In subsequent iterations, $d(i)$ might decrease and so c_{ij}^d might become negative. Next observe that if $d(j)$ decreases during the algorithm, then for some arc (i, j) in the predecessor graph c_{ij}^d may become positive, thereby contradicting the invariant property. But observe that in this case, we immediately delete arc (i, j) from the graph and so maintain the invariant property. For an illustration, see Figure 5.3: in this example, adding arc (6, 5) to the graph decreases $d(5)$, thereby making $c_{58}^d < 0$. This step increases c_{35}^d, but arc (3, 5) immediately leaves the tree.

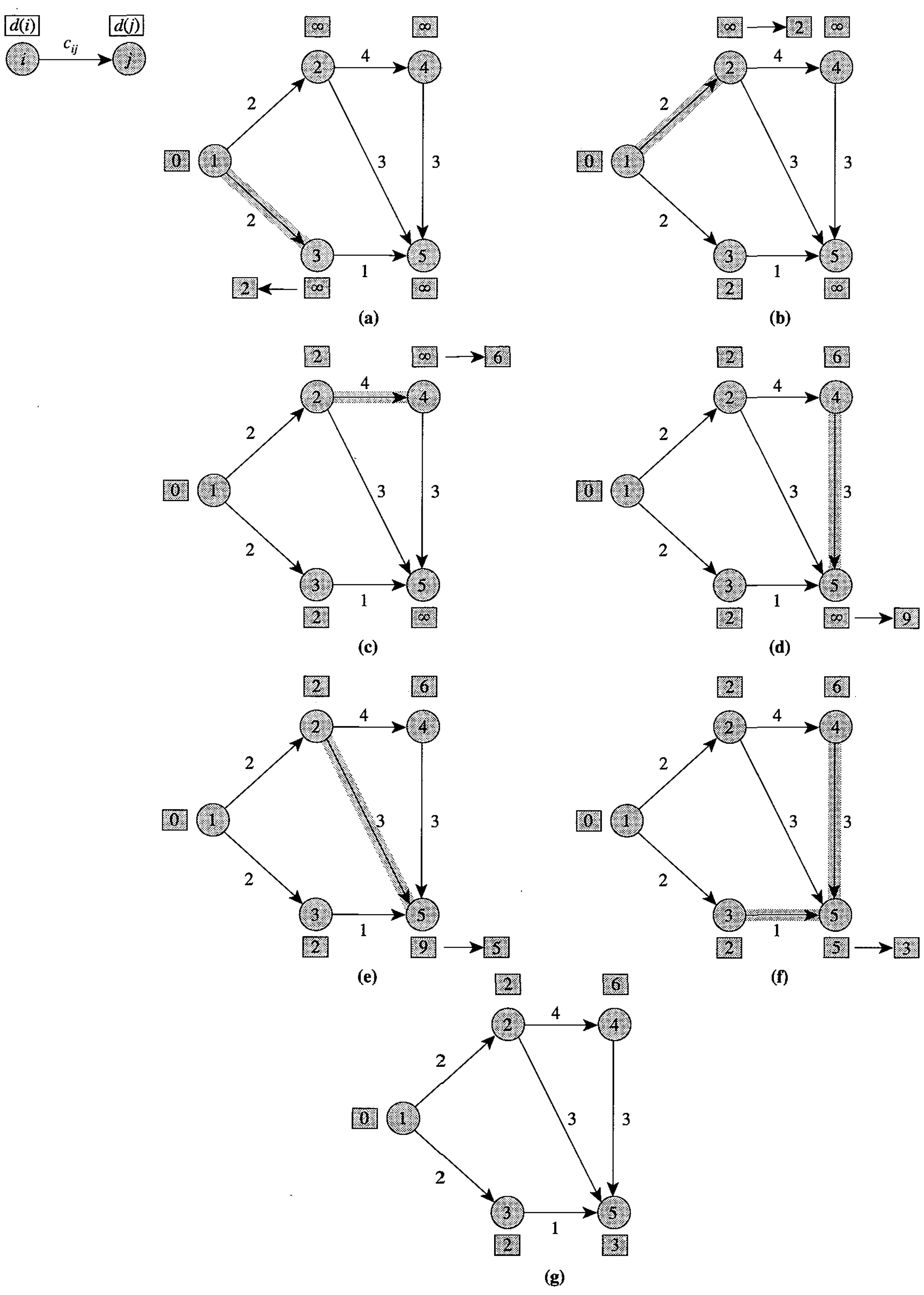

Figure 5.2 Illustrating the generic label-correcting algorithm.

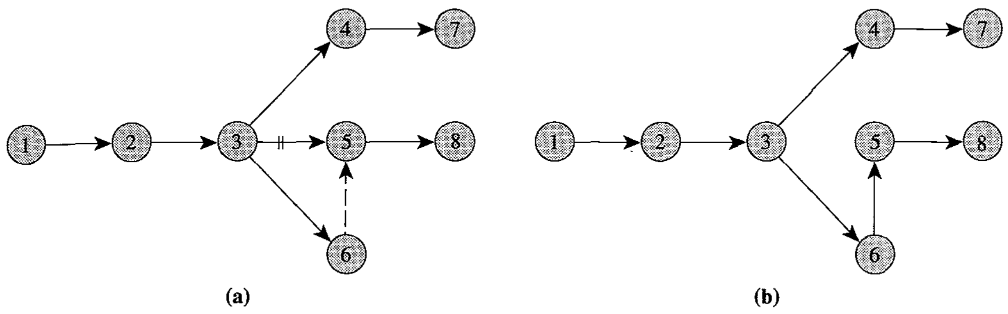
Figure 5.3 Showing that the predecessor graph is a directed out-tree.

We note that the predecessor indices might not necessarily define a tree. To illustrate this possibility, we use the situation shown in Figure 5.4(a). Suppose that arc (6, 2) satisfies $d(2) > d(6) + c_{62}$ (or $c^d_{62} < 0$) and we update the distance label of node 2. This operation modifies the predecessor index of node 2 from 1 to 6 and the graph defined by the predecessor indices is no longer a tree. Why has this happened? The predecessor indices do not define a tree because the network contained a negative cycle. To see that this is the case, notice from Property 5.1 that for the cycle 2–3–6–2, $c_{23} + c_{36} + c_{62} = c^d_{23} + c^d_{36} + c^d_{62} < 0$, because $c^d_{23} \le 0$, $c^d_{36} \le 0$, and $c^d_{62} < 0$. Therefore, the cycle 2–3–6–2 is a negative cycle. This discussion shows that in the absence of negative cycles, we will never encounter a situation shown in Figure 5.4(b) and the predecessor graph will always be a tree.

The predecessor graph contains a unique directed path from the source node to every node k and the length of this path is at most $d(k)$. To verify this result, let P be the path from the source to node k. Since every arc in the predecessor graph has a nonpositive reduced arc length, $\sum_{(i,j)\in P} c^d_{ij} \le 0$. Property 5.2(b) implies that $0 \ge \sum_{(i,j)\in P} c^d_{ij} = \sum_{(i,j)\in P} c_{ij} + d(s) - d(k) = \sum_{(i,j)\in P} c_{ij} - d(k)$. Alternatively, $\sum_{(i,j)\in P} c_{ij} \le d(k)$. When the label-correcting algorithm terminates, each arc in the predecessor graph has a zero reduced arc length (why?), which implies that the length of the path from the source to every node k equals $d(k)$. Consequently, when the algorithm terminates, the predecessor graph is a shortest path tree. Recall from Section 4.3 that a shortest path tree is a directed out-tree rooted at the source with the property that the unique path from the source to any node is a shortest path to that node.

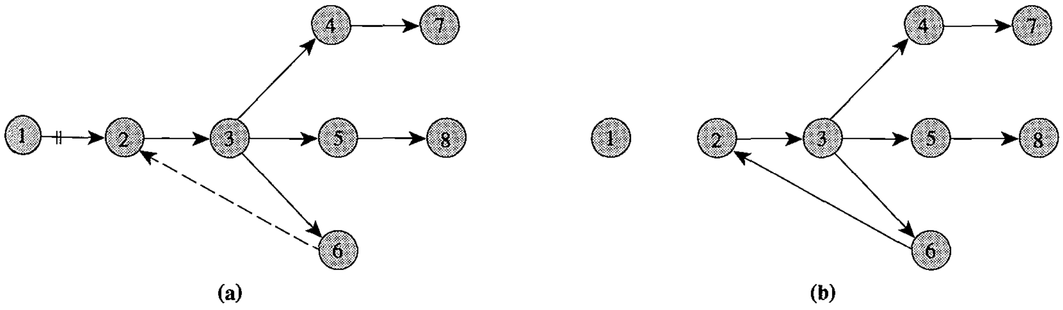
Figure 5.4 Formation of a cycle in a predecessor graph.

It is easy to show that the algorithm terminates in a finite number of iterations. We prove this result when the data are integral; Exercise 5.8 discusses situations when the data are nonintegral. Observe that each $d(j)$ is bounded from above by nC (because a path contains at most $n - 1$ arcs, each of length at most C) and is bounded from below by $-nC$. Therefore, the algorithm updates any label $d(j)$ at most $2nC$ times because each update of $d(j)$ decreases it by at least 1 unit. Consequently, the total number of distance label updates is at most $2n^2C$. Each iteration updates a distance label, so the algorithm performs $O(n^2C)$ iterations. The algorithm also terminates in $O(2^n)$ steps. (See Exercise 5.8.)

Modified Label-Correcting Algorithm

The generic label-correcting algorithm does not specify any method for selecting an arc violating the optimality condition. One obvious approach is to scan the arc list sequentially and identify any arc violating this condition. This procedure is very time consuming because it requires $O(m)$ time per iteration. We shall now describe an improved approach that reduces the workload to an average of $O(m/n)$ time per iteration.

Suppose that we maintain a list, LIST, of all arcs that *might* violate their optimality conditions. If LIST is empty, clearly we have an optimal solution. Otherwise, we examine this list to select an arc, say (i, j), violating its optimality condition. We remove arc (i, j) from LIST, and if this arc violates its optimality condition we use it to update the distance label of node j. Notice that any decrease in the distance label of node j decreases the reduced lengths of all arcs emanating from node j and some of these arcs might violate the optimality condition. Also notice that decreasing $d(j)$ maintains the optimality condition for all incoming arcs at node j. Therefore, if $d(j)$ decreases, we must add arcs in $A(j)$ to the set LIST. Next, observe that whenever we add arcs to LIST, we add *all* arcs emanating from a single node (whose distance label decreases). This suggests that instead of maintaining a list of all arcs that might violate their optimality conditions, we may maintain a list of *nodes* with the property that if an arc (i, j) violates the optimality condition, LIST must contain node i. Maintaining a node list rather than the arc list requires less work and leads to faster algorithms in practice. This is the essential idea behind the modified label-correcting algorithm whose formal description is given in Figure 5.5.

We call this algorithm the *modified label-correcting algorithm*. The correctness of the algorithm follows from the property that the set LIST contains every node i that is incident to an arc (i, j) violating the optimality condition. By performing induction on the number of iterations, it is easy to establish the fact that this property remains valid throughout the algorithm. To analyze the complexity of the algorithm, we make several observations. Notice that whenever the algorithm updates $d(j)$, it adds node j to LIST. The algorithm selects this node in a later iteration and scans its arc list $A(j)$. Since the algorithm can update the distance label $d(j)$ at most $2nC$ times, we obtain a bound of $\sum_{i \in N} (2nC) \mid A(i) \mid = O(nmC)$ on the total number of arc scannings. Therefore, this version of the generic label-correcting algorithm runs in $O(nmC)$ time. When C is exponentially large, the running time is $O(2^n)$. (See Exercise 5.8.)

```
algorithm modified label-correcting;
begin
    $d(s) := 0$ and pred$(s) := 0$;
    $d(j) := \infty$ for each node $j \in N - \{s\}$;
    LIST $:= \{s\}$;
    while LIST $\neq \emptyset$ do
    begin
        remove an element $i$ from LIST;
        for each arc $(i, j) \in A(i)$ do
        if $d(j) > d(i) + c_{ij}$ then
        begin
            $d(j) := d(i) + c_{ij}$;
            pred$(j) := i$;
            if $j \notin$ LIST then add node $j$ to LIST;
        end;
    end;
end;
```

Figure 5.5 Modified label-correcting algorithm.

5.4 *SPECIAL IMPLEMENTATIONS OF THE MODIFIED LABEL-CORRECTING ALGORITHM*

One nice feature of the generic (or the modified) label-correcting algorithm is its flexibility: We can select arcs that do not satisfy the optimality condition in any order and still assure finite convergence of the algorithm. One drawback of this general algorithmic strategy, however, is that without a further restriction on the choice of arcs in the generic label-correcting algorithm (or nodes in the modified label-correcting algorithm), the algorithm does not necessarily run in polynomial time. Indeed, if we apply the algorithm to a pathological set of data and make a poor choice at every iteration, the number of steps can grow exponentially with n. (Since the algorithm is a pseudopolynomial-time algorithm, these instances must have exponentially large values of C. See Exercises 5.27 and 5.28 for a family of such instances.) These examples show that to obtain polynomially bounded label-correcting algorithms, we must organize the computations carefully. If we apply the modified label-correcting algorithm to a problem with nonnegative arc lengths and we always examine a node from LIST with the minimum distance label, the resulting algorithm is the same as Dijkstra's algorithm discussed in Section 4.5. In this case our selection rule guarantees that the algorithm examines at most n nodes, and the algorithm can be implemented to run in $O(n^2)$ time. Similarly, when applying the modified label-correcting algorithm to acyclic networks, if we examine nodes in LIST in the topological order, shortest path algorithm becomes the one that we discussed in Section 4.4, so it is a polynomial-time algorithm.

In this section we study two new implementations of the modified label-correcting algorithm. The first implementation runs in $O(nm)$ time and is currently the best strongly polynomial-time implementation for solving the shortest path problem with negative arc lengths. The second implementation is not a polynomial-time method, but is very efficient in practice.

O(nm) *Implementation*

We first describe this implementation for the generic label-correcting algorithm. In this implementation, we arrange arcs in A in some specified (possibly arbitrary) order. We then make passes through A. In each pass we scan arcs in A, one by one, and check the condition $d(j) > d(i) + c_{ij}$. If the arc satisfies this condition, we update $d(j) = d(i) + c_{ij}$. We stop when no distance label changes during an entire pass.

Let us show that this algorithm performs at most $n - 1$ passes through the arc list. Since each pass requires $O(1)$ computations for each arc, this conclusion implies the $O(nm)$ time bound for the algorithm. We claim that at the end of the kth pass, the algorithm will compute shortest path distances for all nodes that are connected to the source node by a shortest path consisting of k or fewer arcs. We prove this claim by performing induction on the number of passes. Our claim is surely true for $k = 1$. Now suppose that the claim is true for the kth pass. Thus $d(j)$ is the shortest path length to node j provided that some shortest path to node j contains k or fewer arcs, and is an upper bound on the shortest path length otherwise.

Consider a node j that is connected to the source node by a shortest path $s = i_0 - i_1 - i_2 - \cdots - i_k - i_{k+1} = j$ consisting of $k + 1$ arcs, but has no shortest path containing fewer than $k + 1$ arcs. Notice that the path $i_0 - i_1 - \cdots - i_k$ must be a shortest path from the source to node i_k, and by the induction hypothesis, the distance label of node i_k at the end of the kth pass must be equal to the length of this path. Consequently, when we examine arc (i_k, i_{k+1}) in the $(k + 1)$th pass, we set the distance label of node i_{k+1} equal to the length of the path $i_0 - i_1 - \cdots - i_k - i_{k+1}$. This observation establishes that our induction hypothesis will be true for the $(k + 1)$th pass as well.

We have shown that the label correcting algorithm requires $O(nm)$ time as long as at each pass we examine all the arcs. It is not necessary to examine the arcs in any particular order.

The version of the label-correcting algorithm we have discussed considers every arc in A during every pass. It need not do so. Suppose that we order the arcs in the arc list by their tail nodes so that all arcs with the same tail node appear consecutively on the list. Thus, while scanning arcs, we consider one node at a time, say node i, scan arcs in $A(i)$, and test the optimality condition. Now suppose that during one pass through the arc list, the algorithm does not change the distance label of node i. Then during the next pass, $d(j) \le d(i) + c_{ij}$ for every $(i, j) \in A(i)$ and the algorithm need not test these conditions. Consequently, we can store all nodes whose distance labels change during a pass, and consider (or examine) only those nodes in the next pass. One plausible way to implement this approach is to store the nodes in a list whose distance labels change in a pass and examine this list in the first-in, first-out (FIFO) order in the next pass. If we follow this strategy in every pass, the resulting implementation is exactly the same as the modified label-correcting algorithm stated in Figure 5.5 provided that we maintain LIST as a queue (i.e., select nodes from the front of LIST and add nodes to the rear of LIST). We call this algorithm the *FIFO label-correcting algorithm* and summarize the preceding discussion as the following theorem.

Theorem 5.3. The FIFO label-correcting algorithm solves the shortest path problem in $O(nm)$ time.

Dequeue Implementation

The modification of the modified label-correcting algorithm we discuss next has a pseudopolynomial worst-case behavior but is very efficient in practice. Indeed, this version of the modified label-correcting algorithm has proven in practice to be one of the fastest algorithms for solving the shortest path problems in sparse networks. We refer to this implementation of the modified label-correcting algorithm as the *dequeue implementation.*

This implementation maintains LIST as a *dequeue*. A dequeue is a data structure that permits us to store a list so that we can add or delete elements from the front as well as the rear of the list. A dequeue can easily be implemented using an array or a linked list (see Appendix A). The dequeue implementation always selects nodes from the front of the dequeue, but adds nodes either at the front or at the rear. If the node has been in the LIST earlier, the algorithm adds it to the front; otherwise, it adds the node to the rear. This heuristic rule has the following intuitive justification. If a node i has appeared previously in LIST, some nodes, say i_1, i_2, . . . , i_k, might have node i as its predecessor. Suppose further that LIST contains the nodes $i_1, i_2, \ldots, i_k$ when the algorithm updates $d(i)$ again. It is then advantageous to update the distance labels of nodes $i_1, i_2, \ldots, i_k$ from node i as soon as possible rather than first examining the nodes $i_1, i_2, \ldots, i_k$ and then reexamine them when their distance labels eventually decrease due to decrease in $d(i)$. Adding node i to the front of LIST tends to correct the distance labels of nodes $i_1, i_2, \ldots, i_k$ quickly and reduces the need to reexamine nodes. Empirical studies have observed similar behavior and found that the dequeue implementation examines fewer nodes than do most other label-correcting algorithms.

5.5 DETECTING NEGATIVE CYCLES

So far we have assumed that the network contains no negative cycle and described algorithms that solve the shortest path problem. We now describe modifications required in these algorithms that would permit us to detect the presence of a negative cycle, if one exists.

We first study the modifications required in the generic label-correcting algorithm. We have observed in Section 5.2 that if the network contains a negative cycle, no set of distance labels will satisfy the optimality condition. Therefore, the label-correcting algorithm will keep decreasing distance labels indefinitely and will never terminate. But notice that $-nC$ is a lower bound on any distance label whenever the network contains no negative cycle. Consequently, if we find that the distance label of some node k has fallen below $-nC$, we can terminate any further computation. We can obtain the negative cycle by tracing the predecessor indices starting at node k.

Let us describe yet another negative cycle detection algorithm. This algorithm checks at repeated intervals to see whether the predecessor graph contains a directed cycle. Recall from the illustration shown in Figure 5.4 how the predecessor graph might contain a directed cycle. This algorithm works as follows. We first designate the source node as marked and all other nodes as unmarked. Then, one by one, we examine each unmarked node k and perform the following operation: We mark node k, trace the predecessor indices starting at node k, and mark all the nodes encountered until we reach the first already marked node, say node l. If $k = l$, the predecessor graph contains a cycle, which must be a negative cycle (why?). The reader can verify that this algorithm requires $O(n)$ time to check the presence of a directed cycle in the predecessor graph. Consequently, if we apply this algorithm after every αn distance updates for some constant α, the computations it performs will not add to the worst-case complexity of any label-correcting algorithm.

In general, at the time that the algorithm relabels node j, $d(j) = d(i) + c_{ij}$ for some node i which is the predecessor of j. We refer to the arc (i, j) as a *predecessor arc*. Subsequently, $d(i)$ might decrease, and the labels will satisfy the condition $d(j) \geq d(i) + c_{ij}$ as long as pred$(j) = i$. Suppose that P is a path of predecessor arcs from node 1 to node j. The inequalities $d(k) \geq d(l) + c_{kl}$ for all arcs (k, l) on this path imply that $d(j)$ is at least the length of this path. Consequently, no node j with $d(j) \leq -nC$ is connected to node 1 on a path consisting only of predecessor arcs. We conclude that tracing back predecessor arcs from node j must lead to a cycle, and by Exercise 5.56, any such cycle must be negative.

The FIFO label-correcting algorithm is also capable of easily detecting the presence of a negative cycle. Recall that we can partition the node examinations in the FIFO algorithm into several passes and that the algorithm examines any node at most once within each pass. To implement this algorithm, we record the number of times that the algorithm examines each node. If the network contains no negative cycle, it examines any node at most $(n - 1)$ times [because it makes at most $(n - 1)$ passes]. Therefore, if it examines a node more than $(n - 1)$ times, the network must contain a negative cycle. We can also use the technique described in the preceding paragraph to identify negative cycles.

The FIFO label-correcting algorithm detects the presence of negative cycles or obtains shortest path distances in a network in $O(nm)$ time, which is the fastest available strongly polynomial-time algorithm for networks with nonnegative arc lengths. However, for problems that satisfy the similarity assumption, other weakly polynomial-time algorithms run faster than the FIFO algorithm. These approaches formulate the shortest path problem as an assignment problem (as described in Section 12.7) and then use an $O(n^{1/2}m \log(nC))$ time assignment algorithm to solve the problem (i.e., either finds a shortest path or detects a negative cycle).

5.6 ALL-PAIRS SHORTEST PATH PROBLEM

The all-pairs shortest path problem requires that we determine shortest path distances between every pair of nodes in a network. In this section we suggest two approaches for solving this problem. The first approach, called the *repeated shortest path algorithm*, is well suited for sparse networks. The second approach is a generalization of the label-correcting algorithm discussed in previous sections; we refer to this procedure as the *all-pairs label-correcting algorithm*. It is especially well suited for dense networks. In this section we describe the generic all-pairs label-correcting algorithm and then develop a special implementation of this generic algorithm, known as the *Floyd–Warshall algorithm*, that runs in $O(n^3)$ time.

In this section we assume that the underlying network is strongly connected (i.e., it contains a directed path from any node to every other node). We can easily satisfy this assumption by selecting an arbitrary node, say node s, and adding arcs (s, i) and (i, s) of sufficiently large cost for all $i \in N - \{s\}$, if these arcs do not already exist. For reasons explained earlier, we also assume that the network does not contain a negative cycle. All the algorithms we discuss, however, are capable of detecting the presence of a negative cycle. We discuss situations with negative cycles at the end of this section.

Repeated Shortest Path Algorithm

If the network has nonnegative arc lengths, we can solve the all-pairs shortest path problem by applying any single-source shortest path algorithm n times, considering each node as the source node once. If $S(n, m, C)$ denotes the time needed to solve a shortest path problem with nonnegative arc lengths, this approach solves the all-pairs shortest path problem in $O(n\,S(n, m, C))$ time.

If the network contains some negative arcs, we first transform the network to one with nonnegative arc lengths. We select a node s and use the FIFO label-correcting algorithm, described in Section 5.4, to compute the shortest distances from node s to all other nodes. The algorithm either detects the presence of a negative cycle or terminates with the shortest path distances $d(j)$. In the first case, the all-pairs shortest path problem has no solution, and in the second case, we consider the shortest path problem with arc lengths equal to their reduced arc lengths with respect to the distance labels $d(j)$. Recall from Section 5.2 that the reduced arc length of an arc (i, j) with respect to the distance labels $d(j)$ is $c_{ij}^d = c_{ij} + d(i) - d(j)$, and if the distance labels are shortest path distances, then $c_{ij}^d \geq 0$ for all arcs (i, j) in A [see Property 5.2(c)]. Since this transformation produces nonnegative reduced arc lengths, we can then apply the single-source shortest path algorithm for problems with nonnegative arc lengths n times (by considering each node as a source once) to determine shortest path distances between all pairs of nodes in the transformed network. We obtain the shortest path distance between nodes k and l in the original network by adding $d(l) - d(k)$ to the corresponding shortest path distance in the transformed network [see Property 5.2(b)]. This approach requires $O(nm)$ time to solve the first shortest path problem, and if the network contains no negative cycles, it requires an extra $O(n\,S(n, m, C))$ time to compute the remaining shortest path distances. Therefore, this approach determines all pairs shortest path distances in $O(nm + n\,S(n, m, C)) = O(n\,S(n, m, C))$ time. We have established the following result.

Theorem 5.4. *The repeated shortest path algorithm solves the all-pairs shortest path problem in $O(n\,S(n, m, C))$ time.*

In the remainder of this section we study the generic all-pairs label-correcting algorithm. Just as the generic label-correcting algorithm relies on shortest path optimality conditions, the all-pairs label-correcting algorithm relies on all-pairs shortest path optimality conditions, which we study next.

All-Pairs Shortest Path Optimality Conditions

Let $[i, j]$ denote a pair of nodes i and j in the network. The all-pairs label-correcting algorithm maintains a distance label $d[i, j]$ for every pair of nodes; this distance label represents the length of some directed *walk* from node i to node j and hence will be an upper bound on the shortest path length from node i to node j. The algorithm updates the matrix of distance labels until they represent shortest path distances. It uses the following generalization of Theorem 5.1:

Theorem 5.5 (All-Pairs Shortest Path Optimality Conditions). *For every pair of nodes $[i, j] \in N \times N$, let $d[i, j]$ represent the length of some directed path from node i to node j. These distances represent all-pairs shortest path distances if and only if they satisfy the following all-pairs shortest path optimality conditions:*

$$d[i, j] \le d[i, k] + d[k, j] \qquad \text{for all nodes } i, j, \text{ and } k. \tag{5.3}$$

Proof. We use a contradiction argument to establish that the shortest path distances $d[i, j]$ must satisfy the conditions (5.3). Suppose that $d[i, k] + d[k, j] < d[i, j]$ for nodes i, j, and k. The union of the shortest paths from node i to node k and node k to node j is a directed walk of length $d[i, k] + d[k, j]$ from node i to node j. This directed walk decomposes into a directed path, say P, from node i to node j and some directed cycles (see Exercise 3.51). Since each directed cycle in the network has nonnegative length, the length of the path P is at most $d[i, k] + d[k, j] < d[i, j]$, contradicting the optimality of $d[i, j]$.

We now show that if the distance labels $d[i, j]$ satisfy the conditions in (5.3), they represent shortest path distances. We use an argument similar to the one we used in proving Theorem 5.1. Let P be a directed path of length $d[i, j]$ consisting of the sequence of nodes $i = i_1 - i_2 - i_3 - \cdots - i_k = j$. The condition (5.3) implies that

$$\begin{aligned} d[i, j] = d[i_1, i_k] \le d[i_1, i_2] + d[i_2, i_k] &\le c_{i_1 i_2} + d[i_2, i_k], \\ d[i_2, i_k] &\le c_{i_2 i_3} + d[i_3, i_k], \\ &\vdots \\ d[i_{k-1}, i_k] &\le c_{i_{k-1} i_k}. \end{aligned}$$

These inequalities, in turn, imply that

$$d[i, j] \le c_{i_1 i_2} + c_{i_2 i_3} + \cdots + c_{i_{k-1} i_k} = \sum_{(i,j) \in P} c_{ij}.$$

Therefore, $d[i, j]$ is a lower bound on the length of any directed path from node i to node j. By assumption, $d[i, j]$ is also an upper bound on the shortest path length from node i to node j. Consequently, $d[i, j]$ must be the shortest path length between these nodes which is the derived conclusion of the theorem. ◆

All-Pairs Generic Label Correcting Algorithm

The all-pairs shortest path optimality conditions (throughout the remainder of this section we refer to these conditions simply as the optimality conditions) immediately yield the following generic all-pairs label-correcting algorithm: Start with some distance labels $d[i, j]$ and successively update these until they satisfy the optimality conditions. Figure 5.6 gives a formal statement of the algorithm. In the algorithm we refer to the operation of checking whether $d[i, j] > d[i, k] + d[k, j]$, and if so, then setting $d[i, j] = d[i, k] + d[k, j]$ as a *triple operation.*

```
algorithm all-pairs label-correcting;
begin
    set $d[i, j] := \infty$ for all $[i, j] \in N \times N$;
    set $d[i, i] := 0$ for all $i \in N$;
    for each $(i, j) \in A$ do $d[i, j] := c_{ij}$;
    while the network contains three nodes $i$, $j$, and $k$
        satisfying $d[i, j] > d[i, k] + d[k, j]$ do $d[i, j] := d[i, k] + d[k, j]$;
end;
```

Figure 5.6 Generic all-pairs label-correcting algorithm.

To establish the finiteness and correctness of the generic all-pairs label-correcting algorithm, we assume that the data are integral and that the network contains no negative cycle. We first consider the correctness of the algorithm. At every step the algorithm maintains the invariant property that whenever $d[i, j] < \infty$, the network contains a directed walk of length $d[i, j]$ from node i to node j. We can use induction on the number of iterations to show that this property holds at every step. Now consider the directed walk of length $d[i, j]$ from node i to node j at the point when the algorithm terminates. This directed walk decomposes into a directed path, say P, from node i to node j, and possibly some directed cycles. None of these cycles could have a positive length, for otherwise we would contradict the optimality of $d[i, j]$.

Therefore, all of these cycles must have length zero. Consequently, the path P must have length $d[i, j]$. The distance labels $d[i, j]$ also satisfy the optimality conditions (5.3), for these conditions are the termination criteria of the algorithm. This conclusion establishes the fact that when the algorithm terminates, the distance labels represent shortest path distances.

Now consider the finiteness of the algorithm. Since all arc lengths are integer and C is the largest magnitude of any arc length, the maximum (finite) distance label is bounded from above by nC and the minimum distance label is bounded from below by $-nC$. Each iteration of the generic all-pairs label-correcting algorithm decreases some $d[i, j]$. Consequently, the algorithm terminates within $O(n^3C)$ iterations. This bound on the algorithm's running time is pseudopolynomial and is not attractive from the viewpoint of worst-case complexity. We next describe a specific implementation of the generic algorithm, known as the *Floyd–Warshall algorithm*, that solves the all-pairs shortest path problem in $O(n^3)$ time.

Floyd–Warshall Algorithm

Notice that given a matrix of distances $d[i, j]$, we need to perform $\Omega(n^3)$ triple operations in order to test the optimality of this solution. It is therefore surprising that the Floyd–Warshall algorithm obtains a matrix of shortest path distances within $O(n^3)$ computations. The algorithm achieves this bound by applying the triple operations cleverly. The algorithm is based on inductive arguments developed by an application of a dynamic programming technique.

Let $d^k[i, j]$ represent the length of a shortest path from node i to node j subject to the condition that this path uses only the nodes $1, 2, \ldots, k-1$ as internal nodes. Clearly, $d^{n+1}[i, j]$ represents the actual shortest path distance from node i to node j. The Floyd–Warshall algorithm first computes $d^1[i, j]$ for all node pairs i and j. Using $d^1[i, j]$, it then computes $d^2[i, j]$ for all node pairs i and j. It repeats this process until it obtains $d^{n+1}[i, j]$ for all node pairs i and j, when it terminates. Given $d^k[i, j]$, the algorithm computes $d^{k+1}[i, j]$ using the following property.

Property 5.6. $d^{k+1}[i, j] = min\{d^k[i, j], d^k[i, k] + d^k[k, i]\}$.

This property is valid for the following reason. A shortest path that uses only the nodes $1, 2, \ldots, k$ as internal nodes either (1) does not pass through node k, in which case $d^{k+1}[i, j] = d^k[i, j]$, or (2) does pass through node k, in which case $d^{k+1}[i, j] = d^k[i, k] + d^k[k, j]$. Therefore, $d^{k+1}[i, j] = \min\{d^k[i, j], d^k[i, k] + d^k[k, j]\}$.

Figure 5.7 gives a formal description of the Floyd–Warshall algorithm.

```
algorithm Floyd-Warshall;
begin
    for all node pairs [i, j] ∈ N × N do
            d[i, j] : = ∞ and pred[i, j] : = 0;
    for all nodes i ∈ N do d[i, i] : = 0;
    for each arc (i, j) ∈ A do d[i, j] : = $c_{ij}$ and pred[i, j] : = i;
    for each k : = 1 to n do
            for each [i, j] ∈ N × N do
                if d[i, j] > d[i, k] + d[k, j] then
                begin
                    d[i, j] : = d[i, k] + d[k, j];
                    pred[i, j] : = pred[k, j];
                end;
end;
```

Figure 5.7 Floyd–Warshall algorithm.

The Floyd–Warshall algorithm uses predecessor indices, *pred*$[i, j]$, for each node pair $[i, j]$. The index pred$[i, j]$ denotes the last node prior to node j in the tentative shortest path from node i to node j. The algorithm maintains the invariant property that when $d[i, j]$ is finite, the network contains a path from node i to node j of length $d[i, j]$. Using the predecessor indices, we can obtain this path, say P, from node k to node l as follows. We backtrack along the path P starting at node l. Let $g = \text{pred}[k, l]$. Then g is the node prior to node l in P. Similarly, $h = \text{pred}[k, g]$ is the node prior to node g in P, and so on. We repeat this process until we reach node k.

The Floyd–Warshall algorithm clearly performs n major iterations, one for each k, and within each major iteration, it performs $O(1)$ computations for each node pair. Consequently, it runs in $O(n^3)$ time. We thus have established the following result.

Theorem 5.7. *The Floyd–Warshall algorithm computes shortest path distances between all pairs of nodes in $O(n^3)$ time.* ◆

Detection of Negative Cycles

We now address the issue of detecting a negative cycle in the network if one exists. In the generic all-pairs label-correcting algorithm, we incorporate the following two tests whenever the algorithm updates a distance label $d[i, j]$ during a triple iteration:

1. If $i = j$, check whether $d[i, i] < 0$.
2. If $i \neq j$, check whether $d[i, j] < -nC$.

If either of these two tests is true, the network contains a negative cycle. To verify this claim, consider the first time during a triple iteration when $d[i, i] < 0$ for some node i. At this time $d[i, i] = d[i, k] + d[k, i]$ for some node $k \neq i$. This condition implies that the network contains a directed walk from node i to node k, and a directed walk from node k to node i, and that the sum of the lengths of these two walks is $d[i, i]$, which is negative. The union of these two walks is a closed walk, which can be decomposed into a set of directed cycles (see Exercise 3.51). Since $d[i, i] < 0$, at least one of these directed cycles must be negative.

We next consider the situation in which $d[i, j] < -nC$ for some node pair i and j. Consider the first time during a triple iteration when $d[i, j] < -nC$. At this time the network contains a directed walk from node i to node j of length $-nC$. As we observed previously, we can decompose this walk into a directed path P from node i to node j and some directed cycles. Since the path P must have a length of at least $-(n - 1)C$, at least one of these cycles must be a negative cycle.

Finally, we observe that if the network contains a negative cycle, then eventually $d[i, i] < 0$ for some node i or $d[i, j] < -nC$ for some node pair $[i, j]$, because the distance labels continue to decrease by an integer amount at every iteration. Therefore, the generic label-correcting algorithm will always determine a negative cycle if one exists.

In the Floyd–Warshall algorithm, we detect the presence of a negative cycle simply by checking the condition $d[i, i] < 0$ whenever we update $d[i, i]$ for some node i. It is easy to see that whenever $d[i, i] < 0$, we have detected the presence of a negative cycle. In Exercise 5.37 we show that whenever the network contains a negative cycle, then during the computations we will eventually satisfy the condition $d[i, i] < 0$ for some i.

We can also use an extension of the method described in Section 5.5, using the predecessor graph, to identify a negative cycle in the Floyd–Warshall algorithm. The Floyd–Warshall algorithm maintains a predecessor graph for each node k in the network, which in the absence of a negative cycle is a directed out-tree rooted at node k (see Section 5.3). If the network contains a negative cycle, eventually the predecessor graph contains a cycle. For any node k, the predecessor graph consists of the arcs $\{(\text{pred}[k, i], i) : i \in N - \{k\}\}$. Using the method described in Section 5.5, we can determine whether or not any predecessor graph contains a cycle. Checking this condition for every node requires $O(n^2)$ time. Consequently, if we use this method after every αn^2 triple operations for some constant α, the computations will not add to the worst-case complexity of the Floyd–Warshall algorithm.

Comparison of the Two Methods

The generic all-pairs label-correcting algorithm and its specific implementation as the Floyd–Warshall algorithm are matrix manipulation algorithms. They maintain a matrix of tentative shortest path distances between all pairs of nodes and perform repeated updates of this matrix. The major advantages of this approach, compared to the repeated shortest path algorithm discussed at the beginning of this section, are its simplicity, intuitive appeal, and ease of implementation. The major drawbacks of this approach are its significant storage requirements and its poorer worst-case complexity for all network densities except completely dense networks. The matrix manipulation algorithms require $\Omega(n^2)$ intermediate storage space, which could prohibit its application in some situations. Despite these disadvantages, the matrix manipulation algorithms have proven to be fairly popular computational methods for solving all-pairs shortest path problems.

5.7 MINIMUM COST-TO-TIME RATIO CYCLE PROBLEM

The *minimum cost-to-time ratio cycle problem* is defined on a directed graph G with both a cost and a travel time associated with each arc: we wish to find a directed cycle in the graph with the smallest ratio of its cost to its travel time. The minimum cost-to-time ratio cycle problem arises in an application known as the *tramp steamer problem*, which we defined in Application 4.4. A tramp steamer travels from port to port, carrying cargo and passengers. A voyage of the steamer from port i to port j earns p_{ij} units of profit and requires time τ_{ij}. The captain of the steamer wants to know what ports the steamer should visit, and in which order, in order to maximize its mean daily profit. We can solve this problem by identifying a directed cycle with the largest possible ratio of total profit to total travel time. The tramp steamer then continues to sail indefinitely around this cycle.

In the tramp steamer problem, we wish to identify a directed cycle W of G with the maximum ratio $(\sum_{(i,j)\in W} p_{ij})/(\sum_{(i,j)\in W} \tau_{ij})$. We can convert this problem into a minimization problem by defining the cost c_{ij} of each arc (i, j) as $c_{ij} = -p_{ij}$. We then seek a directed cycle W with the minimum value for the ratio

$$\mu(W) = \frac{\sum_{(i,j)\in W} c_{ij}}{\sum_{(i,j)\in W} \tau_{ij}}.$$

We assume in this section that all data are integral, that $\tau_{ij} \geq 0$ for every arc $(i, j) \in A$, and that $\sum_{(i,j)\in W} \tau_{ij} > 0$ for every directed cycle W in G.

We can solve the minimum cost-to-time ratio cycle problem (or, simply, the minimum ratio problem) by repeated applications of the negative cycle detection algorithm. Let μ^* denote the optimal objective function value of the minimum cost-to-time ratio cycle problem. For any arbitrary value of μ, let us define the length of each arc as $l_{ij} = c_{ij} - \mu\tau_{ij}$. With respect to these arc lengths, we could encounter three situations:

Case 1. *G contains a negative (length) cycle W.*

In this case, $\sum_{(i,j)\in W} (c_{ij} - \mu\tau_{ij}) < 0$. Alternatively,

$$\mu > \frac{\sum_{(i,j)\in W} c_{ij}}{\sum_{(i,j)\in W} \tau_{ij}} \geq \mu^*. \tag{5.4}$$

Therefore, μ is a strict upper bound on μ^*.

Case 2. *G contains no negative cycle, but does contain a zero-length cycle W^*.*

The fact that G contains no negative cycle implies that $\sum_{(i,j)\in W} (c_{ij} - \mu\tau_{ij}) \geq 0$ for every directed cycle W. Alternatively,

$$\mu \leq \frac{\sum_{(i,j)\in W} c_{ij}}{\sum_{(i,j)\in W} \tau_{ij}} \quad \text{for every directed cycle } W. \tag{5.5}$$

Similarly, the fact that G contains a zero-length cycle W^* implies that

$$\mu = \frac{\sum_{(i,j)\in W^*} c_{ij}}{\sum_{(i,j)\in W^*} \tau_{ij}}. \tag{5.6}$$

The conditions (5.5) and (5.6) imply that $\mu = \mu^*$, so W^* is a minimum cost-to-time ratio cycle.

Case 3. *Every directed cycle W in G has a positive length.*

In this case $\sum_{(i,j)\in W} (c_{ij} - \mu\tau_{ij}) > 0$ for every directed cycle W. Alternatively,

$$\mu < \frac{\sum_{(i,j)\in W} c_{ij}}{\sum_{(i,j)\in W} \tau_{ij}} \quad \text{for every directed cycle } W. \tag{5.7}$$

Consequently, μ is a strict lower bound on μ^*.

The preceding case analysis suggests the following search procedure for solving the minimum cost-to-time ratio problem. We guess a value μ for μ^*, define arc lengths as $(c_{ij} - \mu\tau_{ij})$, and apply any shortest path algorithm. If the algorithm identifies a negative cycle, μ exceeds μ^* and our next guess should be smaller. If the algorithm terminates with shortest path distances, we look for a zero-length cycle (as described in Exercise 5.19). If we do find a zero-length cycle W^*, then we stop; otherwise, μ is smaller than μ^*, so our next guess should be larger. To implement this general solution approach, we need to define what we mean by "smaller" and "larger." The following two search algorithms provide us with two methods for implementing this approach.

Sequential search algorithm. Let μ° be a known upper bound on μ^*. If we solve the shortest path problem with $(c_{ij} - \mu^\circ\tau_{ij})$ as arc lengths, we either find a zero-length cycle W or find a negative cycle W. In the former case, W is a minimum ratio cycle and we terminate the search. In the latter case, we chose $\mu^1 = (\sum_{(i,j)\in W} c_{ij})/(\sum_{(i,j)\in W} \tau_{ij})$ as our next guess. Case 1 shows that $\mu^\circ > \mu^1 \geq \mu^*$. Repeating this process, we obtain a sequence of values $\mu^\circ > \mu^1 > \cdots > \mu^k = \mu^*$. In Exercise 5.48 we ask the reader to obtain a pseudopolynomial bound on the number of iterations performed by this search procedure.

Binary search algorithm. In this algorithm we identify a minimum cost-to-time ratio cycle using the binary search technique described in Section 3.3. Let $[\underline{\mu}, \overline{\mu}]$ be an interval that contains μ^*, that is, $\underline{\mu} \leq \mu^* \leq \overline{\mu}$. If $C = \max\{c_{ij} : (i, j) \in A\}$, it is easy to verify that $[-C, C]$ is one such interval. At every iteration of the binary search algorithm, we consider $\mu^\circ = (\underline{\mu} + \overline{\mu})/2$, and check whether the network contains a negative cycle with arc lengths $c_{ij} - \mu^\circ\tau_{ij}$. If it does, $\mu^\circ > \mu^*$ (from Case 1) and we reset $\overline{\mu} = \mu^\circ$, otherwise, $\mu^\circ \leq \mu^*$ (from Case 3) and we reset $\underline{\mu} = \mu^\circ$. At every iteration, we half the length of the search interval. As shown by the following result, after a sufficiently large number of iterations, the search interval becomes so small that it has a unique solution.

Let $c(W)$ and $\tau(W)$ denote the cost and travel time of any directed cycle W of the network G, and let $\tau_0 = \max\{\tau_{ij} : (i, j) \in A\}$. We claim that any interval $[\underline{\mu}, \overline{\mu}]$ of size at most $1/\tau_0^2$ contains at most one value from the set $\{c(W)/\tau(W) : W$ is a directed cycle of the network $G\}$. To establish this result, let W_1 and W_2 be two directed cycles with distinct ratios. Then

$$\left| \frac{c(W_1)}{\tau(W_1)} - \frac{c(W_2)}{\tau(W_2)} \right| \neq 0,$$

or

$$\left| \frac{c(W_1)\tau(W_2) - c(W_2)\tau(W_1)}{\tau(W_1)\tau(W_2)} \right| \neq 0. \tag{5.8}$$

Since the left-hand side of (5.8) is nonzero (and all data are integer), its numerator must be at least 1 in absolute value. The denominator of (5.8) is at most τ_0^2. Therefore, the smallest value of the left-hand side is $1/\tau_0^2$. Consequently, when $(\overline{\mu} - \underline{\mu})$ has become smaller than $1/\tau_0^2$, the interval $[\underline{\mu}, \overline{\mu}]$ must contain at most one ratio of the form $c(W)/\tau(W)$.

Since initially $(\overline{\mu} - \underline{\mu}) = 2C$, after $O(\log(2C\tau_0^2)) = O(\log(\tau_0 C))$ iterations, the length of the interval $[\underline{\mu}, \overline{\mu}]$ becomes less than $1/\tau_0^2$, and we can terminate the binary search. The network then must contain a zero-length cycle with respect to the arc lengths $(c_{ij} - \overline{\mu}\tau_{ij})$; this cycle is a minimum cost-to-time ratio cycle.

Minimum Mean Cycle Problem

The *minimum mean cycle problem* is a special case of the minimum cost-to-time ratio problem obtained by setting the traversal time $\tau_{ij} = 1$ for every arc $(i, j) \in A$. In this case we wish to identify a directed cycle W with the smallest possible *mean cost* $(\sum_{(i,j)\in W} c_{ij})/|W|$ from among all directed cycles in G. The minimum mean cycle problem arises in a variety of situations, such as data scaling (see Application 19.6 in Chapter 19) and as a subroutine in certain minimum cost flow algorithms (see Section 10.5), and its special structure permits us to develop algorithms that are faster than those available for the general minimum cost-to-time ratio cycle problem. In this section we describe an $O(nm)$-time dynamic programming algorithm for solving the minimum mean cycle problem.

In the subsequent discussion, we assume that the network is strongly connected (i.e., contains a directed path between every pair of nodes). We can always satisfy this assumption by adding arcs of sufficiently large cost; the minimum mean cycle will contain no such arcs unless the network is acyclic.

Let $d^k(j)$ denote the length, with respect to the arc lengths c_{ij}, of a shortest directed *walk* containing *exactly* k arcs from a specially designated node s to node j. We can choose any node s as the specially designated node. We emphasize that $d^k(j)$ is the length of a directed walk to node j; it might contain directed cycles. We can compute $d^k(j)$ for every node j and for every $k = 1, \ldots, n$, by using the following recursive relationship:

$$d^k(j) = \min_{\{i:(i,j)\in A\}} \{d^{k-1}(i) + c_{ij}\}. \tag{5.9}$$

We initialize the recursion by setting $d^0(j) = \infty$ for each node j. Given $d^{k-1}(j)$ for all j, using (5.9) we compute $d^k(j)$ for all j, which requires a total of $O(m)$ time. By repeating this process for all $k = 1, 2, \ldots, n$, within $O(nm)$ computations we determine $d^k(j)$ for every node j and for every k. As the next result shows, we are able to obtain a bound on the cost μ^* of the minimum mean cycle in terms of the walk lengths $d^k(j)$.

Theorem 5.8

$$\mu^* = \min_{j\in N} \max_{0\le k\le n-1} \left[\frac{d^n(j) - d^k(j)}{n-k}\right]. \tag{5.10}$$

Proof. We prove this theorem for two cases: when $\mu^* = 0$ and $\mu^* \neq 0$.

Case 1. $\mu^* = 0$. In this case the network does not contain a negative cycle (for otherwise, $\mu^* < 0$), but does contain a zero cost cycle W. For each node $j \in N$, let $d(j)$ denote the shortest path distance from node s to node j. We next replace each arc cost c_{ij} by its reduced cost $c_{ij}^d = c_{ij} + d(i) - d(j)$. Property 5.2 implies that as a result of this transformation, the network satisfies the following properties:

1. All arc costs are nonnegative.
2. All arc costs in W are zero.
3. For each node j, every arc in the shortest path from node s to node j has zero cost.
4. For each node j, the shortest path distances $d^k(j)$, for any $1 \le k \le n$, differ by a constant amount from their values before the transformation.

Let $\bar{d}^k(j)$ denote the length of the shortest walk from node s to node j with respect to the reduced costs c_{ij}^d. Condition 4 implies that the expression (5.10) remains valid even if we replace $d^n(j)$ by $\bar{d}^n(j)$ and $d^k(j)$ by $\bar{d}^k(j)$. Next, notice that for each node $j \in N$,

$$\max_{1\le k\le n-1} [\bar{d}^n(j) - \bar{d}^k(j)] \ge 0, \tag{5.11}$$

because for some k, $\bar{d}^k(j)$ will equal the shortest path length $\bar{d}(j)$, and $\bar{d}^n(j)$ will be at least as large. We now show that for some node p, the left-hand side of (5.11) will be zero, which will establish the theorem. We choose some node j in the cycle W and construct a directed walk containing n arcs in the following manner. First, we traverse the shortest path from node s to node j and then we traverse the arcs in W from node j until the walk contains n arcs. Let node p be the node where this walk ends. Conditions 2 and 3 imply that this walk from node s to node p has a zero length. This walk must contain one or more directed cycle because it contains n arcs. Removing the directed cycles from this walk gives a path, say of length $k \leq n - 1$, from node s to node p of zero length. We have thus shown that $\bar{d}^n(p) = \bar{d}^k(p) = 0$. For node p the left-hand side of (5.11) is zero, so this node satisfies the condition

$$\mu^* = \max_{0 \leq k \leq n-1} \left[\frac{d^n(p) - d^k(p)}{n - k}\right] = 0,$$

as required by the theorem.

Case 2. $\mu^* \neq 0$. Suppose that Δ is a real number. We study the effect of decreasing each arc cost c_{ij} by an amount Δ. Clearly, this change in the arc costs reduces μ^* by Δ, each $d^k(j)$ by $k\Delta$, and therefore the ratio $(d^n(v) - d^k(v))/(n - k)$, and so the right-hand side of (5.10), by an amount Δ. Consequently, translating the costs by a constant affects both sides of (5.10) equally. Choosing the translation to make $\mu^* = 0$ and then using the result of Case 1 provides a proof of the theorem. ◆

We ask the reader to show in Exercise 5.55 that how to use the $d^k(j)$'s to obtain a minimum mean cycle.

5.8 SUMMARY

In this chapter we developed several algorithms, known as the *label-correcting algorithms*, for solving shortest path problems with arbitrary arc lengths. The shortest path optimality conditions, which provide necessary and sufficient conditions for a set of distance labels to define shortest path lengths, play a central role in the development of label-correcting algorithms. The label-correcting algorithms maintain a distance label with each node and iteratively update these labels until the distance labels satisfy the optimality conditions. The generic label-correcting algorithm selects any arc violating its optimality condition and uses it to update the distance labels. Typically, identifying an arc violating its optimality condition will be a time-consuming component of the generic label-correcting algorithm. To improve upon this feature of the algorithm, we modified the algorithm so that we could quickly select an arc violating its optimality condition. We presented two specific implementations of this *modified label-correcting algorithm*: A FIFO implementation improves on its running time in theory and a dequeue implementation improves on its running time in practice. Figure 5.8 summarizes the important features of all the label-correcting algorithms that we have discussed.

The label-correcting algorithms determine shortest path distances only if the network contains no negative cycle. These algorithms are, however, capable of de-

Algorithm	Running Time	Features
Generic label-correcting algorithm	$O(\min\{n^2mC, m2^n\})$	1. Selects arcs violating their optimality conditions and updates distance labels. 2. Requires $O(m)$ time to identify an arc violating its optimality condition. 3. Very general: most shortest path algorithms can be viewed as special cases of this algorithm. 4. The running time is pseudopolynomial and so is unattractive.
Modified label-correcting algorithm	$O(\min\{nmC, m2^n\})$	1. An improved implementation of the generic label-correcting algorithm. 2. The algorithm maintains a set, LIST, of nodes: whenever a distance label $d(j)$ changes, we add node j to LIST. The algorithm removes a node i from LIST and examines arcs in $A(i)$ to update distance labels. 3. Very flexible since we can maintain LIST in a variety of ways. 4. The running time is still unattractive.
FIFO implementation	$O(nm)$	1. A specific implementation of the modified label-correcting algorithm. 2. Maintains the set LIST as a queue and hence examines nodes in LIST in first-in, first-out order. 3. Achieves the best strongly polynomial running time for solving the shortest path problem with arbitrary arc lengths. 4. Quite efficient in practice. 5. In $O(nm)$ time, can also identify the presence of negative cycles.
Dequeue implementation	$O(\min\{nmC, m2^n\})$	1. Another specific implementation of the modified label-correcting algorithm. 2. Maintains the set LIST as a dequeue. Adds a node to the front of dequeue if the algorithm has previously updated its distance label, and to the rear otherwise. 3. Very efficient in practice (possibly, linear time). 4. The worst-case running time is unattractive.

Figure 5.8 Summary of label-correcting algorithms.

tecting the presence of a negative cycle. We described two methods for identifying such a situation: the more efficient method checks at repeated intervals whether the predecessor graphs (i.e., the graph defined by the predecessor indices) contains a directed cycle. This computation requires $O(n)$ time.

To conclude this chapter we studied algorithms for the all-pairs shortest path problem. We considered two basic approaches: a repeated shortest path algorithm and an all-pairs label-correcting algorithm. We described two versions of the latter approach: the generic version and a special implementation known as the Floyd–Warshall algorithm. Figure 5.9 summarizes the basic features of the all-pairs shortest path algorithms that we studied.

Algorithm	Running Time	Features
Repeated shortest path algorithm	$O(nS(n,m,C))$	1. Preprocesses the network so that all (reduced) arc lengths are nonnegative. Then applies Dijkstra's algorithm n times with each node $i \in N$ as the source node. 2. Flexible in the sense that we can use an implementation of Dijkstra's algorithm. 3. Achieves the best available running time for all network densities. 4. Low intermediate storage.
Floyd–Warshall algorithm	$O(n^3)$	1. Corrects distance labels in a systematic way until they represent the shortest path distances. 2. Very easy to implement. 3. Achieves the best available running time for dense networks. 4. Requires $\Omega(n^2)$ intermediate storage.

Figure 5.9 Summary of all pairs shortest path algorithms. [$S(n, m, C)$ is the time required to solve a shortest path problem with nonnegative arc lengths.]

REFERENCE NOTES

Researchers, especially those within the operations research community, have actively studied label-correcting algorithms for many years; much of this development has focused on designing computationally efficient algorithms. Ford [1956] outlined the first label-correcting algorithm for the shortest path problem. Subsequently, several researchers, including Moore [1957] and Ford and Fulkerson [1962], studied properties of the generic label-correcting algorithms. Bellman's [1958] dynamic programming algorithm for the shortest path problem can also be viewed as a label-correcting algorithm. The FIFO implementation of the generic label-correcting algorithm is also due to Bellman [1958]. Although Bellman developed this algorithm more than three decades ago, it is still the best strongly polynomial-time algorithm for solving shortest path problems with arbitrary arc lengths.

In Section 12.7 we show how to transform the shortest path problem into an assignment problem and then solve it using any assignment algorithm. As we note in the reference notes of Chapter 12, we can solve the assignment problem in $O(n^{1/2}m \log(nC))$ time using either the algorithms reported by Gabow and Tarjan [1989a] or the algorithm developed by Orlin and Ahuja [1992]. These developments show that we can solve shortest path problems with arbitrary arc lengths in $O(n^{1/2}m \log(nC))$ time. Thus the best available time bound for solving the shortest path problem with arbitrary arc lengths is $O(\min\{nm, n^{1/2}m \log(nC)\})$: The first bound is due to Bellman [1958], and the second bound is due to Gabow and Tarjan [1989a] and Orlin and Ahuja [1992].

Researchers have exploited the inherent flexibility of the generic label-correcting algorithm to design algorithms that are very efficient in practice. Pape's implementation, described in Section 5.4, is based on an idea due to D'Esopo that was later refined and tested by Pape [1974]. Pape [1980] gave a FORTRAN listing of this algorithm. Pape's algorithm runs in pseudopolynomial time. Gallo and Pallottino [1986] describe a two-queue implementation that retains the computational efficiency of Pape's algorithm and still runs in polynomial time. The papers by Glover, Klingman, and Phillips [1985] and Glover, Klingman, Phillips, and Schneider [1985] have described a variety of specific implementations of the generic label-correcting algorithm and studied their theoretical and computational behavior. These two papers, along with those by Hung and Divoky [1988], Divoky and Hung [1990], and Gallo and Pallottino [1984, 1988], have presented extensive computational results of label-setting and label-correcting algorithms. These studies conclude that for a majority of shortest path problems with nonnegative or arbitrary arc lengths, the label-correcting algorithms, known as *Thresh X1* and *Thresh X2*, suggested by Glover, Klingman, and Phillips [1985], are the fastest shortest path algorithms. The reference notes of Chapter 11 provide references for simplex-based approaches for the shortest path problem.

The generic all-pairs label-correcting algorithm, discussed in Section 5.3, is a generalization of the single source shortest path problem. The Floyd–Warshall algorithm, which was published in Floyd [1962], was based on Warshall's [1962] algorithm for finding transitive closure of graphs.

Lawler [1966] and Dantzig, Blattner, and Rao [1966] are early and important references on the minimum cost-to-time ratio cycle problem. The binary search algorithm described by us in Section 5.7 is due to Lawler [1966]. Dantzig, Blattner, and Rao [1966] presented a primal simplex approach that uses the linear programming formulation of the minimum ratio problems; we discuss this approach in Exercise 5.47. Meggido [1979] describes a general approach for solving minimum ratio problems, which as a special case yields a strongly polynomial-time algorithm for the minimum cost-to-time ratio cycle problem.

The $O(nm)$-time minimum mean cycle algorithm, described in Section 5.7, is due to Karp [1978]. Several other algorithms are available for solving the minimum mean cycle problem: (1) an $O(nm \log n)$ parametric network simplex algorithm proposed by Karp and Orlin [1981], (2) an $O(n^{1/2}m \log(nC))$ algorithm developed by Orlin and Ahuja [1992], and (3) an $O(nm + n^2 \log n)$ algorithm designed by Young, Tarjan, and Orlin [1990]. The best available time bound for solving the minimum mean cycle problem is $O(\min\{nm, n^{1/2}m \log(nC))$: The two bounds contained in this expression are due to Karp [1978] and Orlin and Ahuja [1992]. However, we believe that the parametric network simplex algorithm by Karp and Orlin [1981] would prove to be the most efficient algorithm empirically. We describe an application of the minimum mean cycle problem in Application 19.6. The minimum mean cycle problem also arises in solving minimum cost flow problems (see Goldberg and Tarjan [1987, 1988]).

EXERCISES

5.1. Select a directed cycle in Figure 5.10(a) and verify that it satisfies Property 5.2(a). Similarly, select a directed path from node 1 to node 6 and verify that it satisfies Property 5.2(b). Does the network contain a zero-length cycle?

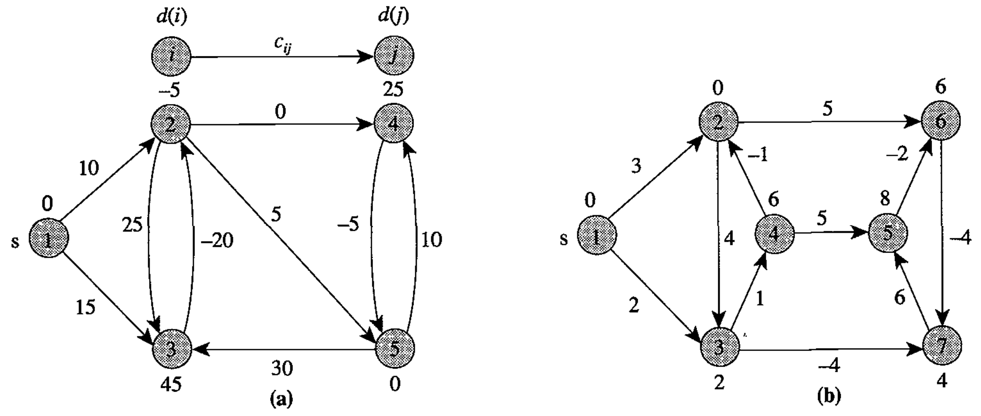

Figure 5.10 Examples for Exercises 5.1 to 5.5.

5.2. Consider the shortest path problems shown in Figure 5.10. Check whether or not the distance label $d(j)$ given next to each node j represents the length of some path. If your answer is yes for every node, list all the arcs that do not satisfy the shortest path optimality conditions.

5.3. Apply the modified label-correcting algorithm to the shortest path problem shown in Figure 5.10(a). Assume that the adjacency list of each node is arranged in increasing order of the head node numbers. Always examine a node with the minimum number in LIST. Specify the predecessor graph after examining each node and count the number of distance updates.

5.4. Apply the FIFO label-correcting algorithm to the example shown in Figure 5.10(b). Perform two passes of the arc list and specify the distance labels and the predecessor graph at the end of the second pass.

5.5. Consider the shortest path problem given in Figure 5.10(a) with the modification that the length of arc (4, 5) is -15 instead of -5. Verify that the network contains a negative cycle. Apply the dequeue implementation of the label-correcting algorithm; after every three distance updates, check whether the predecessor graph contains a directed cycle. How many distance updates did you perform before detecting a negative cycle?

5.6. Construct a shortest path problem whose shortest path tree contains a largest cost arc in the network but does not contain the smallest cost arc.

5.7. **Bellman's equations**

(a) Show that the shortest path distances $d(\cdot)$ must satisfy the following equations, known as *Bellman's equations*:

$$d(j) = \min\{d(i) + c_{ij} : (i, j) \in A(i)\} \qquad \text{for all } j \in N.$$

(b) Show that if a set of distance labels $d(i)$'s satisfy Bellman's equations and the network contains no zero-length cycle, these distance labels are shortest path distances.

(c) Verify that for the shortest path problem shown in Figure 5.11, the distance labels

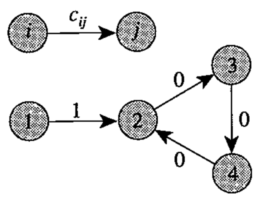

Figure 5.11 Example for Exercise 5.7.

$d = (0, 0, 0, 0)$ satisfy Bellman's equations but do not represent shortest path distances. This example shows that in the presence of a zero-length cycle, Bellman's equations are not sufficient for characterizing the optimality of distance labels.

5.8. Our termination argument of the generic label-correcting algorithm relies on the fact that the data are integral. Suppose that in the shortest path problem, some arc lengths are irrational numbers.
(a) Prove that for this case too, the generic label-correcting algorithm will terminate finitely. (*Hint*: Use arguments based on the predecessor graph.)
(b) (Gallo and Pallottino [1986]). Assuming that the network has no negative cost cycles, show the total number of relabels is $O(2^n)$. (*Hint*: Show first that if the algorithms uses the path 1-2-3-4 to label node 4, then it never uses the path 1-3-2-4 to label node 4. Then generalize this observation.)
(c) Show that the generic label correcting algorithm requires $O(2^n)$ iterations.

5.9. In Dijkstra's algorithm for the shortest path problem, let S denote the set of permanently labeled nodes at some stage. Show that for all node pairs $[i, j]$ for which $i \in S, j \in N$ and $(i, j) \in A$, $d(j) \le d(i) + c_{ij}$. Use this result to give an alternative proof of correctness for Dijkstra's algorithm.

5.10. We define an *in-tree of shortest paths* as a directed in-tree rooted at a sink node t for which the tree path from any node i to node t is a shortest path. State a modification of the generic label-correcting algorithm that produces an in-tree of shortest paths.

5.11. Let $G = (N_1 \cup N_2, A)$ be a bipartite network. Suppose that $n_1 = |N_1|$, $n_2 = |N_2|$ and $n_1 \le n_2$. Show that the FIFO label-correcting algorithm solves the shortest path problem in this network in $O(n_1 m)$ time.

5.12. Let $d^k(j)$ denote the shortest path length from a source node s to node j subject to the condition that the path contains at most k arcs. Consider the $O(nm)$ implementation of the label-correcting algorithm discussed in Section 5.4; let $D^k(j)$ denote the distance label of node j at the end of the kth pass. Show that $D^k(j) \le d^k(j)$ for every node $j \in N$.

5.13. In the shortest path problem with nonnegative arc lengths, suppose that we know that the shortest path distance of nodes $i_1, i_2, \ldots, i_n$ are in nondecreasing order. Can we use this information to help us determine shortest path distances more efficiently than the algorithms discussed in Chapter 4? If we allow arc lengths to be negative, can you solve the shortest path problem faster than $O(nm)$ time?

5.14. Show that in the FIFO label-correcting algorithm, if the kth pass of the arc list decreases the distances of at least $n - k + 1$ nodes, the network must contain a negative cycle. (*Hint*: Use the arguments required in the complexity proof of the FIFO algorithm.)

5.15. **Modified FIFO algorithm** (Goldfarb and Hao [1988]). This exercise describes a modification of the FIFO label-correcting algorithm that is very efficient in practice. The generic label-correcting algorithm described in Figure 5.1 maintains a predecessor graph. Let $f(j)$ denote the number of arcs in the predecessor graph from the source node to node j. We can easily maintain these values by using the update formula $f(j) = f(i) + 1$ whenever we make the distance label update $d(j) = d(i) + c_{ij}$. Suppose that in the algorithm we always examine a node i in LIST with the minimum value of $f(i)$. Show that the algorithm examines the nodes with nondecreasing values of $f(\cdot)$ and that it examines no node more than $n - 1$ times. Use this result to specify an $O(nm)$ implementation of this algorithm.

5.16. Suppose after solving a shortest path problem, you realize that you underestimated each arc length by k units. Suggest an $O(m)$ algorithm for solving the original problem with the correct arc lengths. The running time of your algorithm should be independent of the value of k (*Hint*: Use Dial's implementation described in Section 4.6 on a modified problem.)

5.17. Suppose that after solving a shortest path problem, you realize that you underestimated some arc lengths. The actual arc lengths were $c'_{ij} \ge c_{ij}$ for all $(i, j) \in A$. Let $L = \sum_{(i,j)\in A}(c'_{ij} - c_{ij})$. Suggest an $O(m + L)$ algorithm for reoptimizing the solution obtained for the shortest path problem with arc lengths c_{ij}. (*Hint*: See the hint for Exercise 5.16.)

5.18. Suppose that after solving a shortest path problem, you realize that you underestimated some arc lengths and overestimated some other arc lengths. The actual arc lengths are c'_{ij} instead of c_{ij} for all $(i, j) \in A$. Let $L = \sum_{(i,j)\in A} | c_{ij} - c'_{ij} |$. Suggest an $O(mL)$ algorithm for reoptimizing the shortest path solution obtained with the arc lengths c_{ij}. (*Hint*: Apply the label-correcting algorithm on a modified problem.)

5.19. Identifying zero-length cycles. In a directed network G with arc lengths c_{ij}, let $d(j)$ denote the shortest path distance from the source node s to node j. Define reduced arc lengths as $c^d_{ij} = c_{ij} + d(i) - d(j)$ and define the *zero-residual network* G^0 as the subnetwork of G consisting only of arcs with zero reduced arc lengths. Show that there is a one-to-one correspondence between zero-length cycles in G and directed cycles in G^0. Explain how you can identify a directed cycle in G^0 in $O(m)$ time.

5.20. Enumerating all shortest paths. Define the zero-residual network G^0 as in Exercise 5.19, and assume that G^0 is acyclic. Show that a directed path from node s to node t in G is a shortest path if and only if it is a directed path from node s to node t in G^0. Using this result, describe an algorithm for enumerating all shortest paths in G from node s to node t. (*Hint*: Use the algorithm in Exercise 3.44.)

5.21. Professor May B. Wright suggests the following method for solving the shortest path problem with arbitrary arc lengths. Let $c_{\min} = \min\{c_{ij} : (i, j) \in A\}$. If $c_{\min} < 0$, add $| c_{\min} |$ to the length each arc in the network so that they all become nonnegative. Then use Dijkstra's algorithm to solve the shortest path problem. Professor Wright claims that the optimal solution of the transformed problem is also an optimal solution of the original problem. Prove or disprove her claim.

5.22. Describe algorithms for updating the shortest path distances from node s to every other node if we add a new node $(n + 1)$ and some arcs incident to this node. Consider the following three cases: (1) all arc lengths are nonnegative and node $(n + 1)$ has only incoming arcs; (2) all arc lengths are nonnegative and node $(n + 1)$ has incoming as well as outgoing arcs; and (3) arc lengths are arbitrary, but node $(n + 1)$ has only incoming arcs. Specify the time required for the reoptimization.

5.23. Maximum mnltiplier path problem. The *maximum multiplier path problem* is an extension of the maximum reliability path problem that we discussed in Exercise 4.39, obtained by permitting the constants μ_{ij} to be arbitrary positive numbers. Suppose that we are not allowed to use logarithms. State optimality conditions for the maximum multiplier path problem and show that if the network contains a positive multiplier directed cycle, no path can satisfy the optimality conditions. Specify an $O(nm)$ algorithm for solving the maximum multiplier path problem for networks that contain no positive multiplier directed cycles.

5.24. Sharp distance labels. The generic label-correcting algorithm maintains a predecessor graph at every step. We say that a distance label $d(i)$ is *sharp* if it equals the length of the unique path from node s to node i in the predecessor graph. We refer to an algorithm as *sharp* if every node examined by the algorithm has a sharp distance label. (A sharp algorithm might have nodes with nonsharp distances, but the algorithm never examines them.)

(a) Show by an example that the FIFO implementation of the generic label-correcting algorithm is not a sharp algorithm.

(b) Show that the dequeue implementation of the generic label correcting is a sharp algorithm. (*Hint*: Perform induction on the number of nodes the algorithm examines. Use the fact that the distance label of a node becomes nonsharp only when the distance label of one of its ancestors in the predecessor graph decreases.)

5.25. Partitioning algorithm (Glover, Klingman, and Phillips [1985]). The *partitioning algorithm* is a special case of the generic label-correcting algorithm which divides the set LIST of nodes into two subsets: NOW and NEXT. Initially, NOW = $\{s\}$ and NEXT = $\varnothing$. When examining nodes, the algorithm selects *any* node i in NOW and adds to NEXT any node whose distance label decreases, provided that the node is not already in NOW or NEXT. When NOW becomes empty, the algorithm transfers all the nodes from NEXT to NOW. The algorithm terminates when both NOW and NEXT become empty.

(a) Show that the FIFO label-correcting algorithm is a special case of the partitioning algorithm. (*Hint*: Specify rules for selecting the nodes in NOW, adding nodes to NEXT, and transferring nodes from NEXT to NOW.)

(b) Show that the partitioning algorithm runs in $O(nm)$ time. (*Hint*: Call the steps between two consecutive replenishments of NOW a *phase*. Extend the proof of the FIFO label-correcting algorithm to show that at the end of the kth phase, the algorithm determines optimal distances for all nodes whose shortest paths have no more than k arcs.)

5.26. **Threshold algorithm** (Glover, Klingman, and Phillips [1985]). The threshold algorithm is a variation of the partitioning algorithm discussed in Exercise 5.25. When NOW becomes empty, the threshold algorithm does not transfer all the nodes from NEXT to NOW; instead, it transfers only those nodes i for which $d(i) \le t$ for some threshold value t. At each iteration, the algorithm choses the threshold value t to be at least as large as the minimum distance label in NEXT (before the transfer), so it transfers all those nodes with the minimum distance label, and possibly other nodes as well, from NEXT to NOW. (Note that we have considerable flexibility in choosing t at each step.)

(a) Show that if all arc lengths are nonnegative, the threshold algorithm runs in $O(nm)$ time. (*Hint*: Use the proof of Dijkstra's algorithm.)

(b) Show that if all arc lengths are nonnegative and the threshold algorithm transfers at most five nodes from NEXT to NOW at each step, including a node with the minimum distance label, then it runs in $O(n^2)$ time.

5.27. **Pathological instances of the label-correcting algorithm** (Pallottino [1991]). We noted in Section 5.4 that the dequeue implementation of the generic label-correcting algorithm has excellent empirical behavior. However, for some problem instances, the algorithm performs an exponential number of iterations. In this exercise we describe a method for constructing one such pathological instance for every n. Let $G = (N, A)$ be an acyclic graph with n nodes and an arc (i, j) for every node pair i and j satisfying $i > j$. Let node n be the source node. We define the cost of each arc (i, j) as $c_{ij} = 2^{i-2} - 2^{j-1} \ge 0$. Assume that the adjacency list of each node $i \in N - \{n\}$ is arranged in decreasing order of the head nodes and the adjacency list of the source node n is arranged in the increasing order of the head nodes.

(a) Verify that for $n = 6$, the method generates the instance shown in Figure 5.12.

(b) Consider the instance shown in Figure 5.12. Show that every time the dequeue implementation examines any node (other than node 1), it updates the distance label of node 1. Show that the label of node 1 assumes all values between 15 and 0.

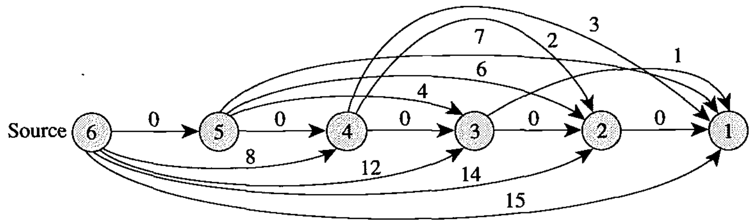

Figure 5.12 Pathological example of the label-correcting algorithm.

5.28. Using induction arguments, show that for an instance with n nodes constructed using the method described in Exercise 5.27, the dequeue implementation of the label-correcting algorithm assigns to node 1 all labels between $2^{n-2} - 1$ to 0 and therefore runs in exponential time.

5.29. Apply the first three iterations (i.e., $k = 1, 2, 3$) of the Floyd–Warshall algorithm to the all-pairs shortest path problems shown in Figure 5.13(a). List four triplets (i, j, k) that violate the all-pairs shortest path optimality conditions at the conclusion of these iterations.

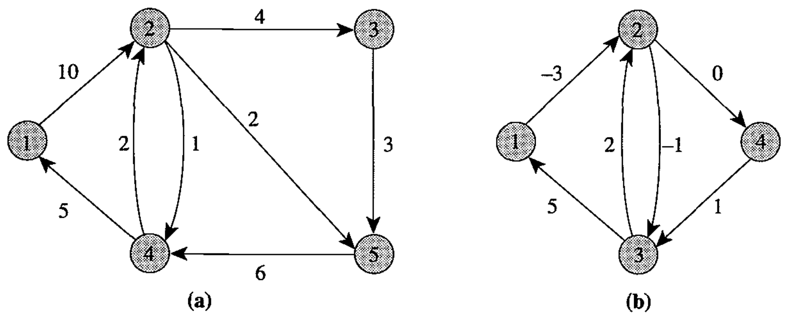

Figure 5.13 Example for Exercises 5.29 to 5.31.

5.30. Solve the all-pairs shortest path problem shown in Figure 5.13(b).

5.31. Consider the shortest path problem shown in Figure 5.13(b), except with c_{31} equal to 3. What is the least number of triple operations required in the Floyd–Warshall algorithm before the node pair distances $d^k[i,j]$ satisfy one of the negative cycle detection conditions?

5.32. Show that if a network contains a negative cycle, the generic all-pairs label-correcting algorithm will never terminate.

5.33. Suppose that the Floyd–Warshall algorithm terminates after detecting the presence of a negative cycle. At this time, how would you detect a negative cycle using the predecessor indices?

5.34. In an all-pairs shortest path problem, suppose that several shortest paths connect node i and node j. If we use the Floyd–Warshall algorithm to solve this problem, which path will the algorithm choose? Will this path be the one with the least number of arcs?

5.35. Consider the maximum capacity path problem defined in Exercise 4.37. Modify the Floyd–Warshall algorithm so that it finds maximum capacity paths between all pairs of nodes.

5.36. Modify the Floyd–Warshall all-pairs shortest path algorithm so that it determines maximum multiplier paths between all pairs of nodes.

5.37. Show that if we use the Floyd–Warshall algorithm to solve the all-pairs shortest path problem in a network containing a negative cycle, then at some stage $d^k[i, i] < 0$ for some node i. [*Hint*: Let i be the least indexed node satisfying the property that the network contains a negative cycle using only nodes 1 through i (not necessarily all of these nodes).]

5.38. Suppose that a network G contains no negative cycle. Let $d^{n+1}(i, j)$ denote the node pair distances at the end of the Floyd–Warshall algorithm. Show that $\min\{d^{n+1}[i, i] : 1 \leq i \leq n\}$ is the minimum length of a directed cycle in G.

5.39. In this exercise we discuss another dynamic programming algorithm for solving the all-pairs shortest path problem. Let d_{ij}^k denote the length of a shortest path from node i to node j subject to the condition that the path contains no more than k arcs. Express d_{ij}^k in terms of d_{ij}^{k-1} and the c_{ij}s and suggest an all-pairs shortest path algorithm that uses this relationship. Analyze the running time of your algorithm.

5.40. Sensitivity analysis. Let d_{ij} denote the shortest path distances between the pair $[i, j]$ of nodes in a directed network $G = (N, A)$ with arc lengths c_{ij}. Suppose that the length of one arc (p, q) changes to value $c'_{pq} < c_{pq}$. Show that the following set of statements finds the modified all-pairs shortest path distances:

```
if $d_{qp} + c'_{pq} < 0$, then the network has a negative cycle
else
  for each pair $[i, j]$ of nodes do
    $d_{ij} := \min\{d_{ij}, d_{ip} + c'_{pq} + d_{qj}\}$;
```

5.41. In Exercise 5.40 we described an $O(n^2)$ method for updating shortest path distances between all-pairs of nodes when we decrease the length of one arc (p, q). Suppose that we increase the length of the arc (p, q). Can you modify the method so that it reoptimizes the shortest path distances in $O(n^2)$ time? If your answer is yes, specify an algorithm for performing the reoptimization and provide a justification for it; and if your answer is no, outline the difficulties encountered.

5.42. Arc addition. After solving an all-pairs shortest path problem, you realize that you omitted five arcs from the network G. Can you reoptimize the shortest path distances with the addition of these arcs in $O(n^2)$ time? (*Hint*: Reduce this problem to the one in Exercise 5.40.)

5.43. Consider the reallocation of housing problem that we discussed in Application 1.1.

(a) The housing authority prefers to use short cyclic changes since they are easier to handle administratively. Suggest a method for identifying a cyclic change involving the least number of changes. (*Hint*: Use the result of one of the preceding exercises.)

(b) Suppose that the person presently residing in a house of category i desperately wants to move to his choice category and that the chair of the housing authority wants to help him. Can the chair identify a cyclic change that allocates the person to his choice category or prove that no such change is possible? (*Hint*: Use the result of one of the preceding exercises.)

5.44. Let $G = (N, A)$ denote the road network of the greater Boston area. Four people living in the suburbs form a car pool. They drive in separate cars to a common meeting point and drive from there in a van to a common point in downtown Boston. Suggest a method for identifying the common meeting point that minimizes the total driving time of all the participants. Also, suggest a method for identifying the common meeting point that minimizes the maximum travel time of any one person.

5.45. Location problems. In a directed $G = (N, A)$ with arc lengths c_{ij}, we define the distance between a pair of nodes i and j as the length of the shortest path from node i to node j.

(a) Define the *radial distance* from node i as the length of the distance from node i to the node farthest from it. We say that a node p is a *center* of the graph G if node p has as small a radial distance as any node in the network. Suggest a straightforward polynomial-time algorithm for identifying a center of G.

(b) Define the *star distance* of node i as the total distance from node i to all the nodes in the network. We refer to a node q as a *median* of G if node q has as small a star distance as any node in the network. Suggest a straightforward polynomial-time algorithm for identifying a median of G.

5.46. Suppose that a network $G = (N, A)$ contains no negative cycle. In this network, let f_{ij} denote the maximum amount we can decrease the length of arc (i, j) without creating any negative cycle, assuming that all other arc lengths remain intact. Design an efficient algorithm for determining f_{ij} for each arc $(i, j) \in A$. (*Hint*: Use the all-pairs shortest path distances.)

5.47. Consider the following linear programming formulation of the minimum cost-to-time ratio cycle problem:

$$\text{Minimize } z = \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{5.12a}$$

subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = 0 \qquad \text{for all } i \in N, \tag{5.12b}$$

$$\sum_{(i,j)\in A} \tau_{ij}x_{ij} = 1, \tag{5.12c}$$

$$x_{ij} \geq 0 \qquad \text{for all } (i, j) \in A. \tag{5.12d}$$

Show that each directed cycle in G defines a feasible solution of (5.12) and that each feasible solution of (5.12) defines a set of one or more directed cycles with the same ratio. Use this result to show that we can obtain an optimal solution of the minimum cost-to-time ratio problem from an optimal solution of the linear program (5.12).

5.48. Obtain a worst-case bound on the number of iterations performed by the sequential search algorithm discussed in Section 5.7 to solve the minimum cost-to-time ratio cycle problem.

5.49. In Section 5.7 we saw how to solve the minimum cost-to-time ratio cycle problem efficiently. This development might lead us to believe that we could also determine efficiently a minimum ratio directed path between two designated nodes s and t (i.e., a path P for which $(\sum_{(i,j)\in P} c_{ij})/(\sum_{(i,j)\in P} \tau_{ij})$ is minimum). This assertion is not valid. Outline the difficulties you would encounter in adapting the algorithm so that it would solve the minimum ratio path problem.

5.50. Use the minimum mean cycle algorithm to identify the minimum mean cycle in Figure 5.13(b).

5.51. **Bit-scaling algorithm** (Gabow [1985]). The bit-scaling algorithm for solving the shortest path problem works as follows. Let $K = \lceil \log C \rceil$. We represent each arc length as a K-bit binary number, adding leading zeros if necessary to make each arc length K bits long. The problem P_k considers the length of each arc as the k leading bits (see Section 3.3). Let d_k^* denote the shortest path distances in problem P_k. The bit-scaling algorithm solves a sequence of problems $P_1, P_2, \ldots, P_k$, using the solution of problem P_{k-1} as the starting solution of problem P_k.

(a) Consider problem P_k and define reduced arc lengths with respect to the distances $2d_{k-1}^*$. Show that the network contains a path from the source node to every other node whose reduced length is at most n. (*Hint*: Consider the shortest path tree of problem P_{k-1}.)

(b) Show how to solve each problem P_k in $O(m)$ time. Use this result to show that the bit-scaling algorithm runs in $O(m \log C)$ time.

5.52. **Modified bit-scaling algorithm.** Consider Exercise 5.51 but using a base β representation of arc cost c_{ij} in place of the binary representation. In problem P_k we use the k leading base β digits of the arc lengths as the lengths of the arcs. Let d_{k-1}^* denote the shortest path distances in Problem P_{k-1}.

(a) Show that if we define reduced arc lengths in problem P_k with respect to the distances βd_{k-1}^*, the network contains a path from the source to every other node whose reduced length is at most $\beta\, n$.

(b) Show how to solve each problem P_k in $O(m + \beta n)$ time and, consequently, show that the modified bit-scaling algorithm runs in $O((m + \beta n) \log_\beta C)$ time. What value of β achieves the least running time?

5.53. **Parametric shortest path problem.** In the parametric shortest path problem, the cost c_{ij} of each arc (i, j) is a linear function of a parameter λ (i.e., $c_{ij} = c_{ij}^\circ + \lambda c_{ij}^*$) and we want to obtain a tree of shortest paths for all values of λ from 0 to $+\infty$. Let T^λ denote a tree of shortest paths for a specific value of λ.

(a) Consider T^λ for some λ. Show that if $d^\circ(j)$ and $d^*(j)$ are the distances in T^λ with respect to the arc lengths c_{ij}° and c_{ij}^*, respectively, then $d^\circ(j) + \lambda d^*(j)$ are the distances with respect to the arc lengths $c_{ij}^{\circ} + \lambda c_{ij}^{*}$ in T^{λ}. Use this result to describe a method for determining the largest value of λ, say $\bar{\lambda}$, for which T^{λ} is a shortest path tree for all λ, $1 \le \lambda \le \bar{\lambda}$. Show that at $\lambda = \bar{\lambda}$, the network contains an alternative shortest path tree. (*Hint*: Use the shortest path optimality conditions.)

(b) Describe an algorithm for determining T^{λ} for all $0 \le \lambda \le \infty$. Show that T^{∞} is shortest path tree with the arc lengths as c_{ij}^{*}.

5.54. Consider a special case of the parametric shortest path problem in which each $c_{ij}^{*} = 0$ or 1. Show that as we vary λ from 0 to $+\infty$, we obtain at most n^2 trees of shortest paths. How many trees of shortest paths do you think we can obtain for the general case? Is it polynomial or exponential? [*Hint*: Let $f(j)$ denote the number of arcs with $c_{ij}^{*} = 1$ in the tree of shortest paths from node s to node j. Consider the effect on the potential function $\Phi = \sum_{j \in N} f(j)$ of the changes in the tree of shortest paths.]

5.55. Let $d^k(j)$ denote the length of the shortest path from node s to node j using at most k arcs in a network G. Suppose that $d^k(j)$ are available for all nodes $j \in N$ and all $k = 1, \ldots, n$. Show how to determine a minimum mean cycle in G. (*Hint*: Use some result contained in Theorem 5.8.)

5.56. Show that if the predecessor graph at any point in the execution of the label-correcting algorithm contains a directed cycle, then the network contains a negative cycle.