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
4.2 Applications
4.3 Tree of Shortest Paths
4.4 Shortest Path Problems in Acyclic Networks
4.5 Dijkstra's Algorithm
4.6 Dial's Implementation
4.7 Heap Implementations
4.8 Radix Heap Implementation
4.9 Summary

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

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

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

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 clementary 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 $a_1 = (x_1, y_1)$, $a_2 = (x_2, y_2)$, . . . , $a_n = (x_n, y_n)$. Suppose that we have ordered the points so that $x_1 \leq x_2 \leq \cdots \leq x_n$. 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

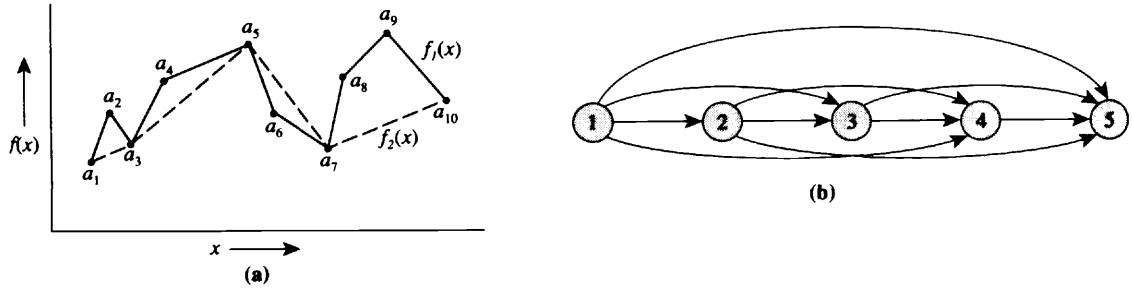

Figure 4.1 Illustrating Applications 4.1: (a) approximating the function $f_1(x)$ passing through 10 points by the function $f_2(x)$; (b) corresponding shortest path problem.

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

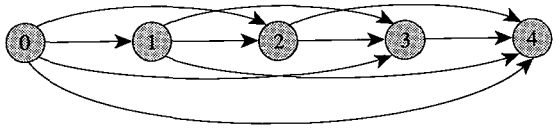

Figure 4.2 Shortest path network associated with the inspection problem.

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

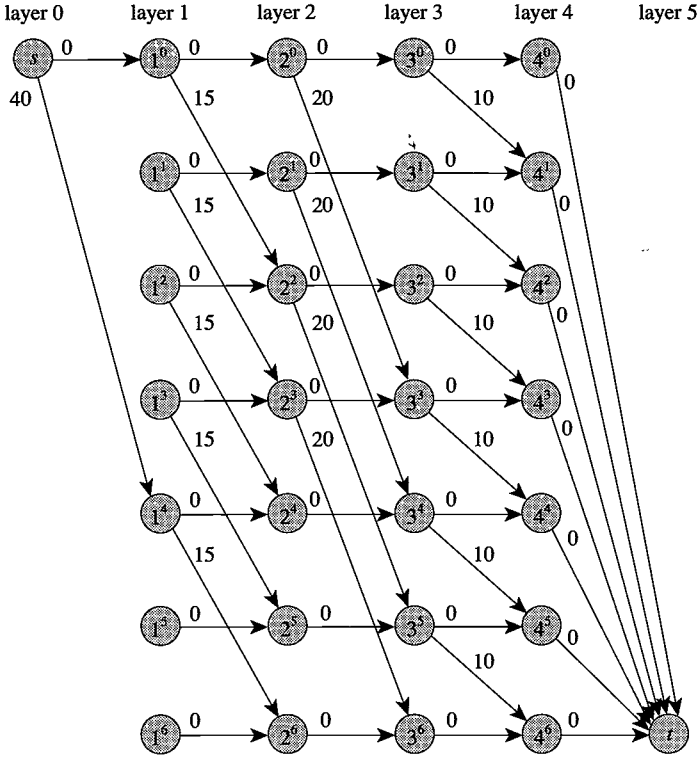

Figure 4.3 Longest path formulation of the knapsack problem.

i^k in the network signifies that the items 1, 2, . . . , 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 $x(j_k) - x(i_k) \leq b(k)$. As an example, consider the following system of constraints whose corresponding graph is shown in Figure 4.4(a):

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

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

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

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

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

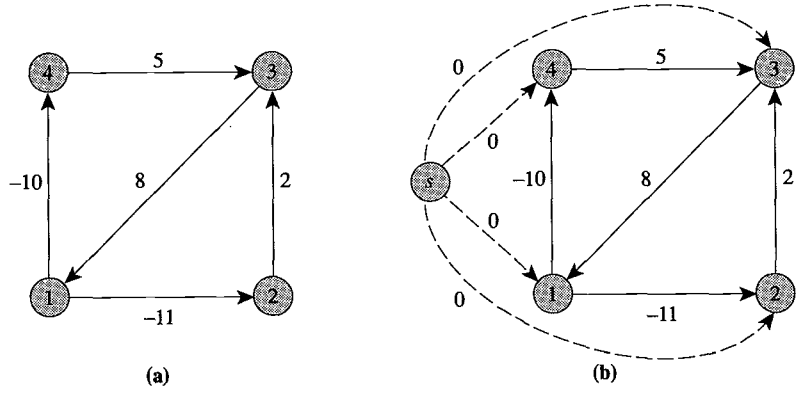

Figure 4.4 Graph corresponding to a system of difference constraints.

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., $x(2) - x(1) \leq -11$, $x(3) - x(2) \leq 2$, and $x(1) - x(3) \leq 8$] are inconsistent because summing these constraints yields the invalid inequality $0 \leq -1$.

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

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

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 op-

timization 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.

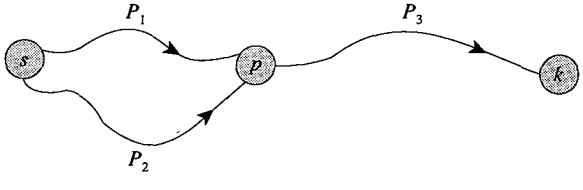

Figure 4.5 Proving Property 4.1.

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 := ∅; S̄ := N;
    d(i) := ∞ for each node i ∈ N;
    d(s) := 0 and pred(s) := 0;
    while |S| < n do
    begin
        let i ∈ S̄ be a node for which d(i) = min{d(j) : j ∈ S̄};
        S := S ∪ {i};
        S̄ := S̄ − {i};
        for each (i, j) ∈ 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.

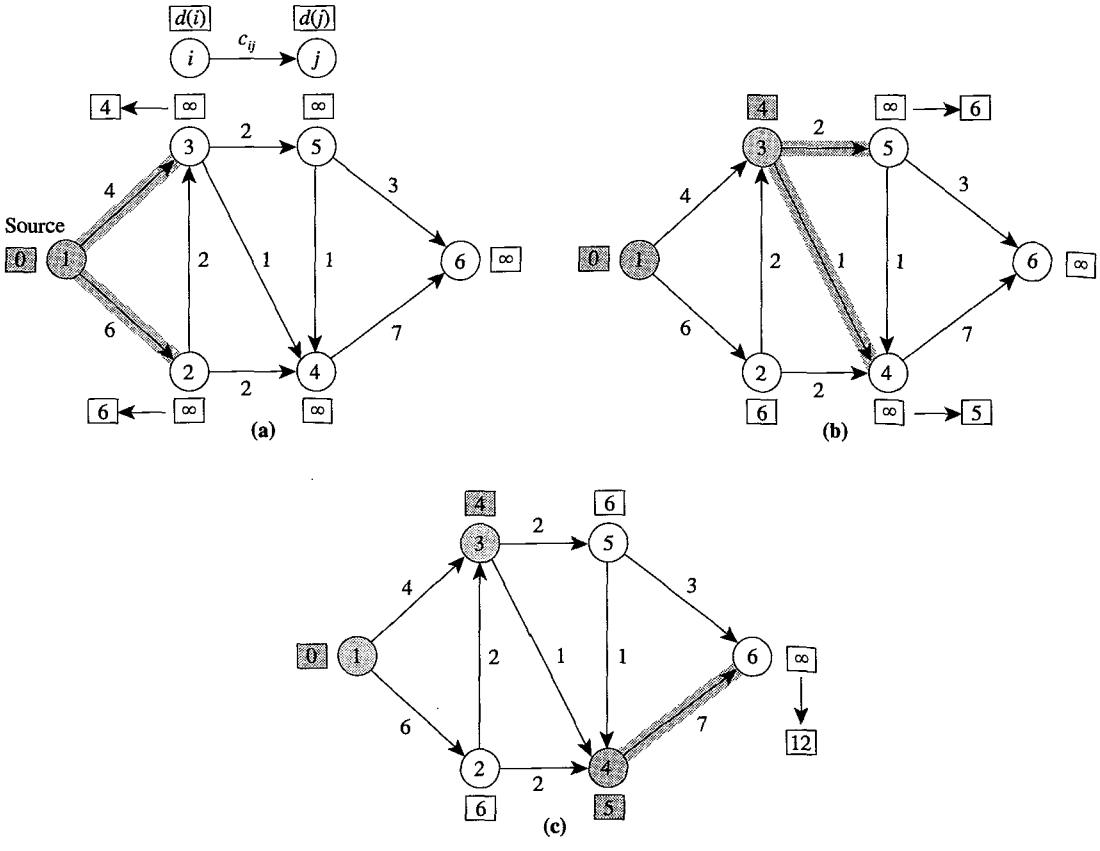

Figure 4.7 Illustrating Dijkstra's algorithm.

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

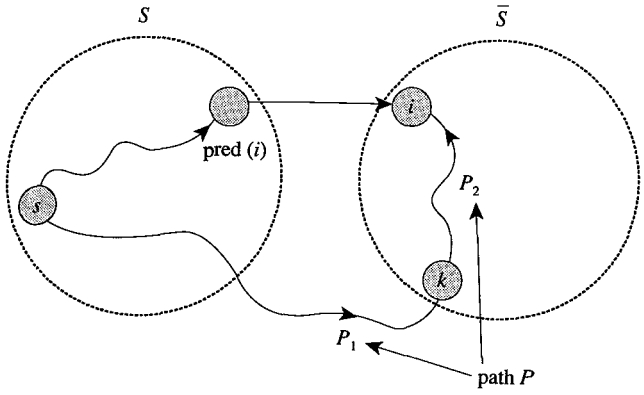

Figure 4.8 Proving Dijkstra's algorithm.

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 $\overline{S}'$, 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 $\overline{S}$ (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 $\overline{S}$ and a node in $\overline{S}'$ 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.

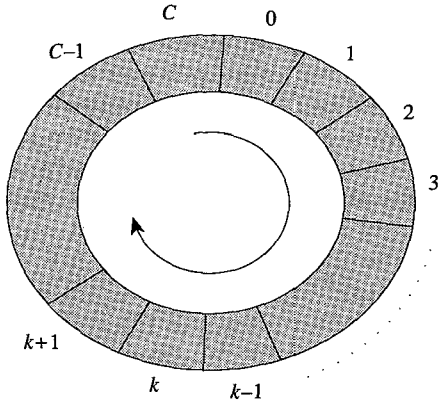

Figure 4.9 Bucket arrangement in Dial's algorithm.

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) : = ∞ for all j ∈ N;
    d(s) : = 0 and pred(s) : = 0;
    insert(s, H);
    while H ≠ ∅ do
    begin
        find-min(i, H);
        delete-min(i, H);
        for each (i, j) ∈ A(i) do
        begin
            value : = d(i) + c_ij;
            if d(j) > value then
                if d(j) = ∞ 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) = $\emptyset$. 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

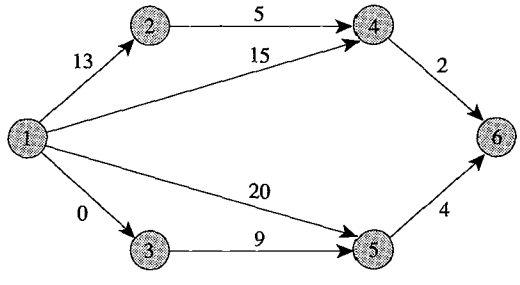

Figure 4.11 Shortest path example.

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}	$\emptyset$	$\emptyset$	$\emptyset$	{2, 4}	{5}	$\emptyset$	

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)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{2, 4, 5}	$\emptyset$	$\emptyset$	$\emptyset$

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 $j \in \text{content}(k)$ 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 $d(j) \notin \text{range}(k)$, 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 outtree 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 (Ex-

ercise 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 $x(j_k) - x(i_k) \leq b(k)$. Suppose, instead, that some constraints are of the form $x(j_k) \leq b(k)$ or $x(i_k) \geq b(k)$. 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.

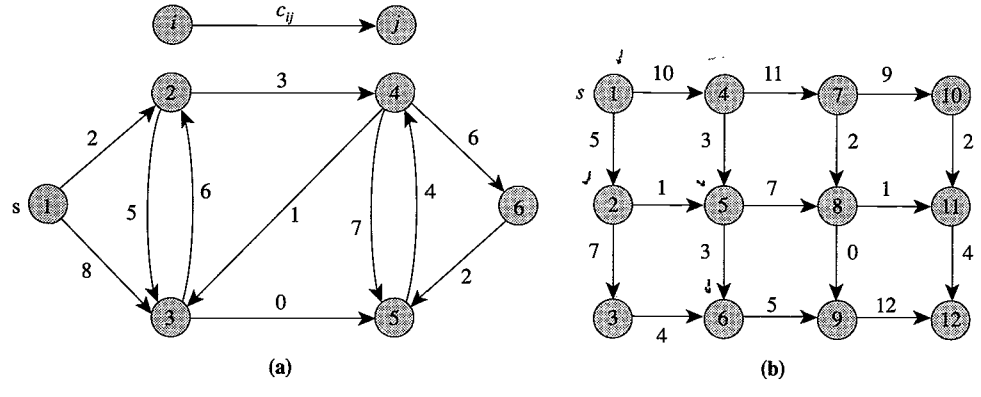

Figure 4.15 Some shortest path networks.

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.

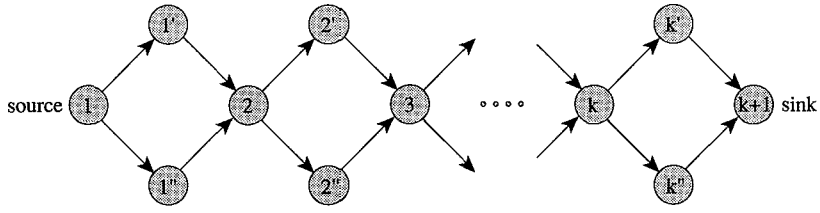

Figure 4.16 Network for Exercise 4.17.

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 $(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)$ 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 $0 < \mu_{ij} \leq 1$ 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 $h(i) \leq h(j) + c_{ij}$ 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.